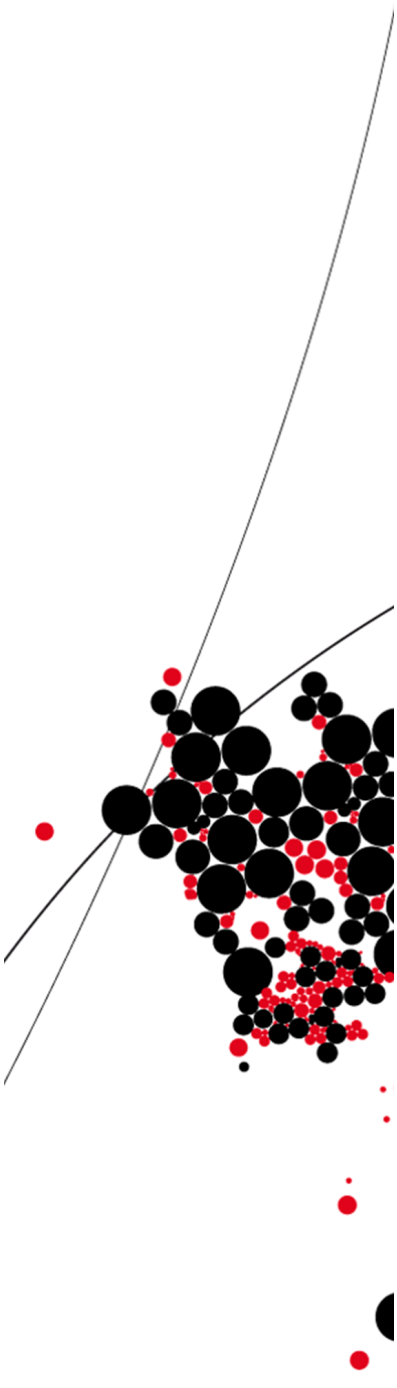
UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Exploring the potential use of FaaS within an iPaaS infrastructure.

Julian M. Elsten
(s1796755)
research@elstenit.nl

Master Thesis
Business Information Technology
July 2023



Supervisors:
Dr.ir. M.J. Van Sinderen
Dr.ir. J.M. Moonen
Business Supervisor:
S. Kaya (eMagiz)

Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

1 Executive Summary

Enterprises increasingly adopt more systems and software to execute their daily activities. In order to connect all these systems and software, a middleware solutions needs to be implemented in order to create interoperability. A specific type of middleware that fills this gap are iPaaS platforms. These integration Platform as a Service products create easy integrations between different applications and systems by providing a platform where a user can design, test and deploy the integration. The platforms can be run on-premise but are nowadays mostly ran in the public cloud.

In this research, we will be taking a closer look at the way of running these platforms in the public cloud. Currently, customer environments containing the integrations are mostly ran in virtual private cloud environments where one or more dedicated virtual machines are running the integrations. This brings multiple problems due to the nature of these environments. First, the machine needs to be scaled by investigating the maximal workload of an integration. This automatically means that the machines are most of the time oversized for their average tasks resulting in higher cloud infrastructure costs than actually needed. Secondly, in case of a growing integration a machines needs to be resized which is currently manual devops work for the iPaaS platforms.

Serverless is often suggested as a flexible technology that saves on manual cloud configurations and has many benefits for changing environments such as auto-scaling. In this research, we will be looking at possible Function as a Service technologies that are suiting for a iPaaS platform in order to improve scalability and reduce infrastructure costs.

In order to come up with a proposed solution, requirements were discussed with experts in the field. These requirements were tested against a list of 10 commercial and open-source FaaS solutions. This resulted in a final two frameworks to be tested in a prototype. To create the prototype, the architecture of a typical iPaaS solution was investigated resulting in the baseline architecture as prevailing architecture. Based on this baseline architecture, a target architecture was constructed with the implementation of the FaaS framework. By comparing these two architecture, a gap analysis and implementation plan is created which serves as input for the prototype.

With the prototype, the last requirements were validated such as checking the auto-scaling capabilities, the costs, the configuration and maintenance needs and the security & isolation. It was found that the open-source frameworks OpenFaaS and OpenWhisk fulfilled the most requirements as set through the interviews. Finally, OpenFaas was chosen as the most suitable FaaS framework for implementation on an iPaaS. However, as additional information needs to be researched still in future research it was not yet recommend to the business of the case study to go all in for implementing this technology but to start preparing the existing environment for future adoption.

This research contributed in multiple ways to practice and science by producing the following artefacts: a baseline architecture of an iPaaS platform, a target architecture of an iPaaS platform implementing opens-source FaaS technology. Recommendations for business in the case study concerning easing the focus on vendor lock-in and preparing for future adoption of FaaS technology.

2 Acknowledgements

I would like to thank Dr.ir. M.J. Van Sinderen (Marten) and Dr.ir. J.M. Moonen (Hans) for their extensive support and feedback during the production of this Master Thesis. A word of thanks also goes to the eMagiz Team which helped me out with information and candidates for interviews when needed. A special word of thanks goes out to Samet Kaya, who onboarded me to the eMagiz Team and helped me greatly by keeping the research on the right track.

3 Table of contents

Contents

4 Table of Figures

List of Figures

5 Table of Tables

List of Tables

6 List of acronyms

Acronyms

AI Artificial Intelligence. 1,

API Application Programming Interface. 1, 2, 13, 20, 24, 26, 43, 44, 46,

AWS Amazon Web Services. 10, 13–15, 21, 23, 30, 57, 59,

BaaS Backend-as-a-Service. 12, 13, 54,

CI/CD Constant Integration / Constant Delivery. 51,

CLI Command Line Interface. 13, 20, 21, 34, 42,

CPU Central Processing Unit. 21, 29, 48,

CRM Customer Relations Management. 1, 8,

CSP Cloud Service Providers. 2, 10, 13–15, 19, 20, 30, 54, 57, 59,

CTO Chief Technical Officer. 16, 17, 23, 38,

DNS Domain Name Service. 25,

EKS Elastic Kubernetes Service. 30,

ERP Enterprise Resource Planning. 1,

FaaS Function-as-a-Service. 12–14, 17, 18, 29, 54, 55, 57–59,

FR Functional Requirements. 16, 19,

GKE Google Kubernetes Engine. 49, 50,

GUI Graphical User Interface. 30,

HTTP Hypertext Transfer Protocol. 13,

IaaS Infrastructure-as-a-Service. 8, 10, 54,

IDE Integrated Development Environment. 34,

IOT Internet-Of-Things. 1,

IP Internet Protocol. 48,

iPaaS integration-Platform-as-a-Service. 1, 2, 8, 9, 23, 25, 27, 38, 53–55, 58,

IT Information Technology. 17,

JSON JavaScript Object Notation. 42,

MS Microsoft. 14, 57,

NFR Non-Functional Requirements. 16, 19,

NIST National Institute of Standards & Technology. 8,

OS Operating System. 10, 54,

PaaS Platform-as-a-Service. 2, 8, 9, 54, 58,

RBAC role-based authentication control. 51,

RPS Requests Per Second. 44,

SaaS Software-as-a-Service. 1, 8, 9, 54, 59,

SQS Simple Queue Service. 21,

SSI semi-structured interview. 16,

SSL Secure Sockets layer. 51,

TLS Transport Layer Security. 51,

URL Uniform Resource Locator. 13, 34,

VM Virtual Machine. 2, 23, 29,

VPC Virtual Private Cloud. 24, 25, 28, 29, 55,

XML Extensible Markup Language. 38, 39,

7 Introduction

7.1 Background

When looking back in history we see a constant demand for innovation in industry. These are historically marked by the industrial revolutions. The first industrial revolution was represented by the introduction of steam and water power. The second industrial revolution at the beginning of the 20th century introduced electrical energy for running industry. The third industrial revolution is based around automation with the aid of electronics and internet technology. Currently we are still working on the 4th industrial revolution which was initially started in 2011. Companies were seeking for more efficiency which encompasses technologies such as Enterprise Resource Planning (ERP), Internet-Of-Things (IOT), cloud based manufacturing and social product development. [7]. In the paper of [7] it is mentioned that "The goals of Industry 4.0 are to achieve a higher level of operational efficiency and productivity, as well as a higher level of automatization". One leading aspect of the aspects of Industry 4.0 is data and industrial integration [8]. This is where middleware solutions and iPaaS platforms come into play.

The trend of ongoing digitalization is currently shifting from Industry 4.0 into Industry 5.0 with more tasks for Artificial Intelligence (AI) and integration of computers into the tasks of people [9]. As a result, more companies face the need for data integrations amongst different applications and domains such as Customer Relations Management (CRM), Enterprise Resource Planning, IOT systems, Smart manufacturing systems, Smart grid systems, Smart building systems etc. These applications include Software-as-a-Service solutions but also legacy on-premise solutions. "Interoperability between information sources is a first condition for meeting the challenges of data value chain management" [10]. Due to this constantly growing complexity of different software solutions adopted by companies it becomes increasingly hard to exchange data amongst different applications. The number of Point to Point connections becomes exponentially higher with each addition of an application. As a result, middleware solutions are on the rise to lessen the amount of needed connections and its appurtenant complexity. Possible components which can be considered middleware are Enterprise Business Bus, Application Programming Interface (API), Message Queues, Data Streaming and many more. By offering these types of middleware solutions into one low-code platform, a company is able to setup data integrations amongst their applications in a low-code environment provided on an As-a-service basis. Additionally, it is possible to change and adapt data to further increase the value of such a solution compared to traditional middleware solutions. An example of such a platform is eMagiz, an iPaaS provider based in The Netherlands.

7.2 eMagiz challenges

As explained in the previous section, there are many developments ongoing in current industry when looking at digitization. With this ongoing digitization comes an increased workload and expectation of customers. Currently, it is difficult to scale the existing and new customer's environments due to a traditional architecture based on virtual machines. Besides scaling is difficult, another aspect is becoming increasingly more important which is the costs for the cloud infrastructure. Currently, we are experiencing a so-called energy crisis which even further pushes up energy costs of computational power. This is not only caused by demographic disturbances such as the war between Russia and Ukraine but also through an increased focus on electrifying houses and businesses. We see more and more business that are being denied a connection to the grid [11] making it important to use power as efficiently as possible. Lastly, speed of development is an important aspect too. By facilitating developers in easily deploying new features to the cloud infrastructure, the overall time it takes before a feature is taken to production becomes lower, improving innovation and lowering the time to market for new features. Taking these aspects into consideration it is important to look at newer and state-of-the art cloud architectures for running the integration integration-Platform-as-a-Service (iPaaS).

eMagiz is currently exploring cloud architectures for running their iPaaS solution as there are many ongoing developments in this field (micro-services, containerization etc.). Due to the varying workload of the integration platform solution current architectures based on Virtual Machine (VM) are not an ideal solution. This is mainly because the size of the VM's is decided on once beforehand and cannot be easily changed without manual intervention and migration to a bigger or smaller machine. Previous research has been done on whether or not it would be possible to run the iPaaS solution in a Containerized architecture using Kubernetes [12]. A new architecture tailored towards future change, scalability and further reduction of infrastructure costs is based on serverless functions. Serverless runs a function on request and shuts down afterwards, eliminating the need to administer the infrastructure needed to run traditional VM's or Docker images. This architecture holds benefits such as cost reduction as there is no need to pay for idle time and a fast time-to-market as there is less need to work on configuration files of the underlying infrastructure.

Therefore, in this paper an investigation into the possibility to increase flexibility and scalability while reducing costs and preserving the current reliability, security and the ability to operate cloud agnostic of the platform is conducted. Research into whether or not serverless would be possible for use within an low-code iPaaS solution, and if so in which form this would be, needs to be done.

7.3 Problem Statement

iPaaS platforms have underlying technologies such as event streaming, API gateways or messaging. These technologies often encounter heavy traffic spikes and drops [13]. When traffic is not there, ideally the machines running the processes to work with this traffic are not running and therefore not costing any money.

Unfortunately, using traditional architectures based on VM's or even more recent technologies such as containerization (for example Kubernetes) never scale to zero and are therefore inducing costs for running the platform. Serverless, a recent cloud technology, has been described to tackle this problem as it can scale to absolutely zero yet can still scale up automatically in case of traffic spikes [14] [13].

However, as serverless is a recent development introduced in 2015 [15] and only has been picking up popularity in practice in recent years, a lot is still unknown and lacking in scientific research. As a result, research needs to be done on whether serverless technology would be suiting to handle typical iPaaS processes such as messaging and event streaming. Additionally, there are a lot of variations within serverless technologies such as Open-source vs Commercial Cloud Service Providers (CSP). Every variation has advantages and disadvantages which are effecting the applicability in a certain use-case.

In recent years a lot more scientific research has been done on serverless technologies. However, no research has been done on the applicability of serverless technologies in a Platform-as-a-Service (PaaS) or specifically an integration-Platform-as-a-Service (iPaaS) scenario. This research aims to identify which functions within a iPaaS solution can run serverless if there are any, which requirements are in place for running serverless and how to implement a serverless solution in such an architecture.

7.4 Research Questions

Following the problem as stated in the problem statement section, research questions have been defined.
The main research question is formulated:

MQ1. How can serverless be implemented for an iPaaS solution in order to improve scalability and reduce infrastructure costs?

In order to come to a concise answering of the main research question, sub-questions have been formulated as follows:

SQ1. What cloud models are currently in use and what defines them? (SaaS, PaaS (iPaaS) and IaaS)?

SQ2. What are the state-of-the-art serverless technologies/frameworks that currently exist and in what way do they differ from each other?

SQ3. Which functional and non-functional requirements are in place for a serverless architecture to ensure similar business functionality, a decrease in costs and DevOps but increase scalability for an iPaaS solution?

SQ4. How do the state-of-the-art serverless technologies/frameworks fulfill the elected requirements?

SQ5. What is the prevailing architecture of an iPaaS infrastructure?

SQ6. How can the state-of-the-art serverless technologies/frameworks be implemented into the current typical iPaaS infrastructure?

SQ7. What measurable improvements on scalability and costs of infrastructure does the new architecture for implementing serverless technologies into iPaaS bring?

7.5 Structure

In this chapter 7 the **Introduction, motivation, problem statement and Research Questions** are described.

In the upcoming chapters, the following information is described:

- In chapter 8 the **State of the art** is presented as a result of the literature analysis performed prior to the design study. The answers of the literature research questions SQ1 - SQ2 can be found and are concluded in this chapter.
- In chapter 9 the **functional and non-functional requirements** are gathered through literature research and expert interviews. These requirements serve as input for the solution design and will be answering SQ3.
- In chapter 10 the **state-of-the-art serverless solutions** are investigated in depth to serve as input and decision tool for the solutions design. This chapter will answer SQ4 partially for requirements that can be answered through literature.
- In chapter 11 the baseline and target**architecture** of the infrastructure of the iPaaS platform are modelled. Additionally, a gap analysis between these two architectures is described. The chapter is concluded with the steps to take to create the architecture architecture which serve as input for the solution design. This chapter answers SQ5 and SQ6.
- In chapter 12 the process of creating the **prototype** is described in order to serve as input for the validation chapter.
- In chapter 13 the prototype is used **to validate** the remaining requirements and open research questions in order to be able to draw a conclusion.
- In chapter 14 the final results are presented and **a conclusion is drawn** answering the main research question.

This results in the following table where all chapters contributing to a specific research question can be found:

Research Question:	**In which chapter is it answered?**
SQ1.	In chapter 8 State of the Art
SQ2.	In chapter 8 State of the Art
SQ3.	In chapter 9 Requirements
SQ4.	In chapter 10 Serverless Solutions and 13 Validation
SQ5.	In chapter 11 Architecture
SQ6.	In chapter 11 Architecture and 12 Prototype
SQ7.	In chapter 13 Validation and 12 Prototype
MQ1.	In chapter 14 Conclusion

Table 1: Summary of which chapter answers which research question.

8 State of the art

In this chapter, an in-depth literature research is conducted in order to gain a better insight in the current state-of-the-art of serverless technologies and Cloud Model Architectures. Additionally, important concepts are explained since many terms concerning serverless are mistakenly interchanged and used. By conducting the background research, enough knowledge is acquired in order to conduct a thorough design research in the later stage while using correct concepts and methods with state-of-the-art information.

8.1 Method Literature Review

In order to conduct a thorough literature review, a structured literature review in conducted. In this section, the procedure and search criteria including the results will be shown. For this research, serverless and cloud models will be considered part of the field of Software Engineering. As a result, the procedure as described by [16] will be used as a guideline for the systematic literature review. As part of this procedure, a first step is to define and document the search strategy used.

For all the literature research questions, different search strategies are described underneath. Overall, the database from Scopus is used which links to other databases accordingly. In the tables in sections 8.2.1 and 8.2.2 the used search term and the results and descriptions leading to the search are exactly described. The results found are from the query processed in August 2022. At a different point in time, more or less results can be found as articles are added to this field of research constantly.

Besides the use of standard literature it is also important to scan so-called "grey literature" or other sources such as conference proceedings, white-papers and technical reports to validate certain outcomes and prevent systematic bias in systematic reviews [16]. Including Grey Literature into the research yields many additional benefits as described by [17] these include "gaining significant knowledge from practitioners in addition to academic articles, reducing publication bias where studies only report positive findings and a way to address topics that are missing from conventional academic sources." Especially in the field of software engineering, many state-of-the-art technologies are available through grey literature instead of traditional peer-reviewed literary sources.

In the study of [17] it is found that around 50% of the grey literature sources are unavailable on the internet through servers or pages being taken offline. As this is not completely unavoidable but this does not contribute to the replicability of this research, we attempt to use high-quality blog articles and websites that are generally more likely to be around for a longer period.

This method will be used to fully answer SQ1. and SQ2. as well as partly SQ3. In order to complement the answering of SQ3, the method of expert interviewing is used which is explained in the following section.

8.2 Search Strategies

In this section, the search strategies used to gather scientific research papers is described.

8.2.1 SQ 1. (What cloud models are currently in use and what defines them? (SaaS, PaaS (iPaaS) and IaaS)?)

Initial search:	TITLE-ABS-KEY (cloud AND models)	143892 results.
Only results that include information on PaaS models as well:	(TITLE-ABS-KEY (cloud AND models)) AND (PaaS)	2227 results.
Limit to information from the last years 2017+:	(TITLE-ABS-KEY (cloud AND models)) AND (PaaS) AND (LIMIT-TO (PUBYEAR , 2023) OR LIMIT-TO (PUBYEAR , 2022) OR LIMIT-TO (PUBYEAR , 2021) OR LIMIT-TO (PUBYEAR , 2020) OR LIMIT-TO (PUBYEAR , 2019) OR LIMIT-TO (PUBYEAR , 2018))	998 results.
Only results that were within the field of Computer Science and include information on SaaS and Iaas services to get a good comparison:	(TITLE-ABS-KEY (cloud AND models)) AND (((PaaS)) AND (iaas)) AND (SaaS) AND (LIMIT-TO (SUBJAREA , "COMP")) AND (LIMIT-TO (PUBYEAR , 2023) OR LIMIT-TO (PUBYEAR , 2022) OR LIMIT-TO (PUBYEAR , 2021) OR LIMIT-TO (PUBYEAR , 2020) OR LIMIT-TO (PUBYEAR , 2019) OR LIMIT-TO (PUBYEAR , 2018) OR LIMIT-TO (PUBYEAR , 2017))	368 results.
Filter to English and final verisons only:	(TITLE-ABS-KEY (cloud AND models)) AND (((PaaS)) AND (iaas)) AND (SaaS) AND (LIMIT-TO (SUBJAREA , "COMP")) AND (LIMIT-TO (PUBYEAR , 2023) OR LIMIT-TO (PUBYEAR , 2022) OR LIMIT-TO (PUBYEAR , 2021) OR LIMIT-TO (PUBYEAR , 2020) OR LIMIT-TO (PUBYEAR , 2019) OR LIMIT-TO (PUBYEAR , 2018)) AND (LIMIT-TO (LANGUAGE , "English")) AND (LIMIT-TO (PUBSTAGE , "final"))	280 results.

The results are ordered on 'relevance' and after reading multiple abstracts, articles which might contain useful information for answering this research question are analyzed in-depth resulting in the section of 8.3. In the end a total amount of 8 articles were fully read to find relevant information for use in this paper.

8.2.2 SQ2. What are the state-of-the-art serverless technologies/frameworks that currently exist and in what way do they differ from each other?)

Initial search:	TITLE-ABS-KEY (serverless)	1384 results.
Filtering to recent years to confirm up-to-date information:	TITLE-ABS-KEY (serverless) AND (LIMIT-TO (PUBYEAR , 2022) OR LIMIT-TO (PUBYEAR , 2021) OR LIMIT-TO (PUBYEAR , 2020) OR LIMIT-TO (PUBYEAR , 2019))	1065 results.
Filtering to only English paper that are final and within the field of computer science:	TITLE-ABS-KEY (serverless) AND (LIMIT-TO (PUBYEAR , 2022) OR LIMIT-TO (PUBYEAR , 2021) OR LIMIT-TO (PUBYEAR , 2020) OR LIMIT-TO (PUBYEAR , 2019)) AND (LIMIT-TO (PUBSTAGE , "final")) AND (LIMIT-TO (SUBJAREA , "COMP")) AND (LIMIT-TO (LANGUAGE , "English"))	949 results.
Add 'Provider' and 'Open Source' to the keywords in order to find papers focusing on the current solutions'	(TITLE-ABS-KEY (serverless)) AND ((open AND source)) AND (provider) AND (LIMIT-TO (PUBYEAR , 2022) OR LIMIT-TO (PUBYEAR , 2021) OR LIMIT-TO (PUBYEAR , 2020) OR LIMIT-TO (PUBYEAR , 2019)) AND (LIMIT-TO (PUBSTAGE , "final")) AND (LIMIT-TO (SUBJAREA , "COMP")) AND (LIMIT-TO (LANGUAGE , "English"))	107 results.

Again, the resulting articles are ordered on relevance and selected for further investigation based on reading through abstracts of suitable titled articles. In the end a total amount of 10 articles were fully read to find relevant information for use in this paper.

8.3 Cloud Models

Nowadays, a lot of software, platforms and infrastructure is marketed on a 'as a service' basis. According to [18] the National Institute of Standards & Technology (NIST) describes the most common types SaaS, PaaS and Iaas as:

- "Software-as-a-Service (SaaS). The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user specific application configuration settings.
- Platform-as-a-Service (PaaS). The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment. A specific kind of PaaS is the type of iPaaS or Integration Platform as a Service.
- Infrastructure-as-a-Service (IaaS). The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)."

8.3.1 SaaS

Software as a service is software delivered over the internet where customers and users only pay for usage of the application. As mentioned before, the user does not need to bother with installing the software within on premises servers or any underlying infrastructure. As a result, the customer does not need to be bothered by maintenance or updates of the software [19]. Secondly, scaling of the application and the underlying hardware resources is done by the SaaS provider.

Some well-known example of SaaS applications are Salesforce CRM system [20], Google Apps such as Gmail, Google Docs and Google Calendar [21] and Freshbooks.com invoicing software [22].

As SaaS can be considered as a blackbox technology where it is not clear where the data and software is ran physically it can be hard to validate compliance in case of strict business policies. Therefore alternatives such as IaaS or PaaS might be a better fit for certain use cases.

8.3.2 PaaS

Compared to SaaS, PaaS provides the platform to run software without the actual application and data. These differences are visualized in figure 2. This makes that a user or client does not need to bother with runtimes, operating systems and the underlying infrastructure. On the other side, they are responsible for the software running on the platform and its associated updates and data. These platforms are often provided by the CSP where the Infrastructure is managed too, such as AWS Elastic Beanstalk. But it is also possible for other vendors to develop a platform and sell it on a As-a-service basis where the underlying infrastructure is managed by them. Certain use cases for a PaaS solution include integration solutions (iPaaS such as eMagiz [23] and Boomi [24]).

We will have a closer look at these specific type of Platform as a Service in the following section on iPaaS.

8.3.3 iPaaS

An integration Platform as a service is a specific kind of PaaS. As the name suggests, it delivers a platform where companies can setup business integration's amongst their different pieces of software. iPaaS is defined as "suite[s] of cloud services enabling development, execution and governance of integration flows connecting any combination of on-premises and cloud-based processes, services, applications and data within individual, or across multiple, organizations" by [25]. As can be derived from this definition, an iPaaS can be used to connect different systems and pieces of software together in order to increase productivity and create business value. It does not matter where these systems or software are running. This can be Software as a Service (Section: 8.3.1) systems or legacy on-premise systems. In earlier times, point-to-point interfaces were used for inter-application communication and data sharing [6]. As more and more companies transitioned into cloud computing and SaaS applications, the need for an alternative became increasingly bigger resulting in the rise of the iPaaS solutions.

An iPaaS solution makes complex integration tasks more easy by creating a low-code environment where complex message and data operations are compiled in understandable graphical data flows. Some scenarios for usage of iPaaS solutions can be found in table 2. By using iPaaS as a middleware solution, a lot of additional connections between all the different applications are eliminated as seen in figure 1. This makes the whole integration process easier to handle an manage as not every point-to-point connection needs to be monitored 24/7. In literature iPaaS providers are also described as "As an intermediary, the iPaaS provider thus connects the market side of the SaaS providers and the businesses." [26].

Scenario:	Description:	Example:
1. Cloud to cloud	Integration between purely cloud-based applications	User profiles in different social networks are synchronized with contact data of a cloud-based CRM.
2. Cloud to on-premise	Integration of cloud-based applications with existing on-premise applications.	A cloud-based CRM system is connected to a legacy ERP system to synchronize customer data.
3. On-premise to on-premise	Integration solely between on-premise applications.	Airlines of an alliance synchronize their passenger information systems.

Table 2: Scenarios for use of iPaaS [6]

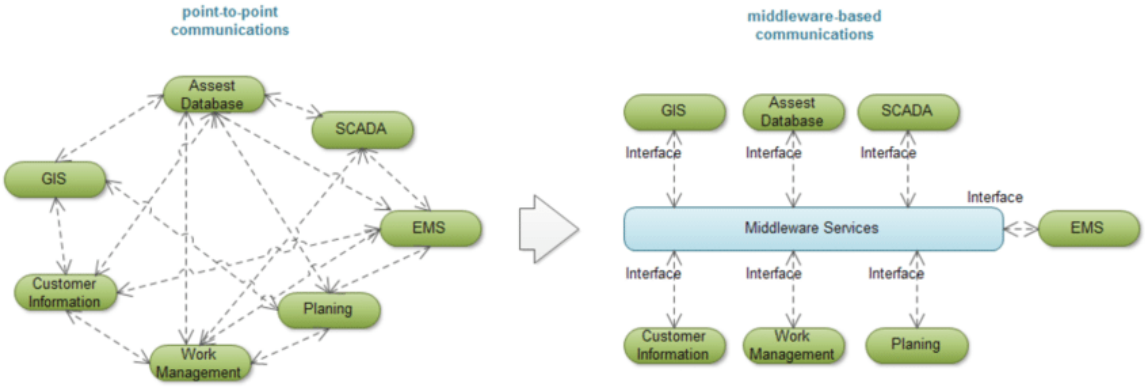


Figure 1: P2P Vs. Middleware Integrations [1]

8.3.4 Iaas

The least managed version of the As a service types is the Infrastructure-as-a-Service (IaaS). In this type of service only the underlying infrastructure (such as Networking, Storage, Servers and Virtualization) is managed by the CSP and the platform on top as well as the software running (Operating System (OS), Middleware, Runtime, Data and Applications) is managed by the client/user. This can be seen in figure 2.

The provider provides the virtualized computing resources over the internet. These resources include storage, severs and the connecting network facilities [27]. A client can access these resources through a system of the CSP where they van access the hypervisor to install virtual machines with specific OS's or other applications such as middleware applications (e.g. databases). By doing so, the client has full control over their software stack without the need to bother with hardware resources as needed in an on-premise solution.

Benefits of such an architecture include a faster, easier and more cost-efficient deployment as there is no need to buy, upgrade, manage and support the underlying infrastructure. This is especially suiting to changing workloads or experimental deployments as no long-term investments have to be made and additional resources are easily added or scaled up.

Common CSP's that provide IaaS services are Amazon's AWS, Microsoft Azure, Google Cloud, Digital Ocean and AliBaba Cloud.

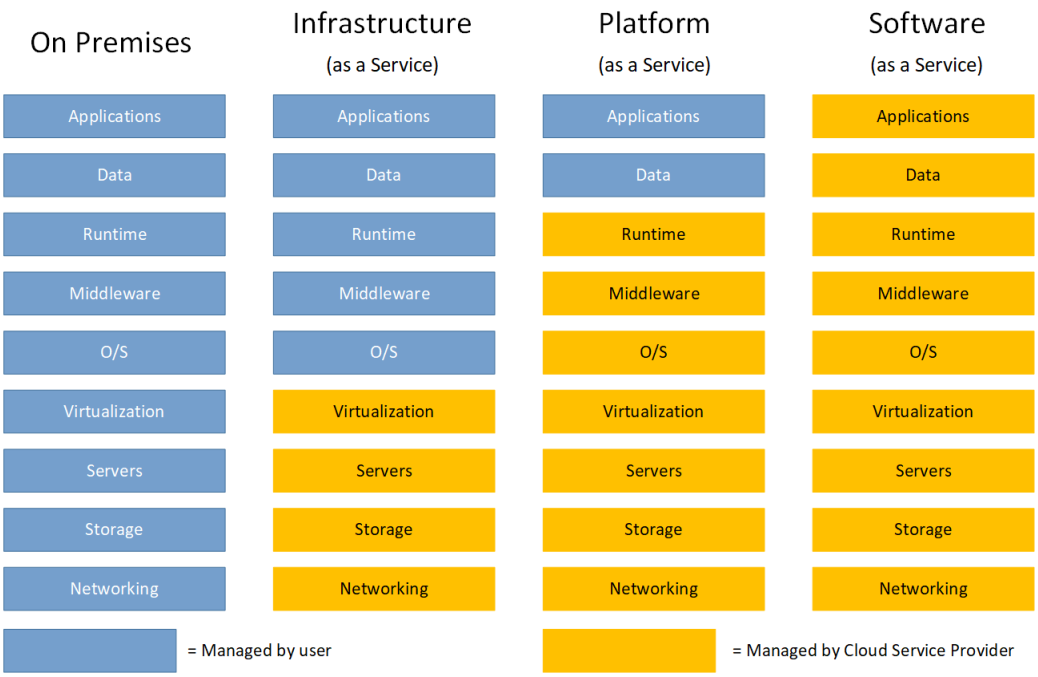


Figure 2: Possible Cloud Models and On Premises compared to each other, based on [2].

8.4 Serverless Technology

In this section, serverless functions and commonly used terms and applications of severless technologies are explained including a brief analysis on the different cloud service providers (CSP) and Open-Source Frameworks providing serverless services.

8.4.1 What is serverless?

Serverless technology is actually not a good name for the technology as you will still need servers at places where calculations are made or storage is provided. Serverless is the term used for on-demand execution of functions or other application where a client does not need to bother with any of the platform provisioning. This yields some benefits as described in [28] "Serverless computing puts multiplexing and scalability to the next level by allowing providers to commit just the required amount of resources to a particular application and utilize the resources for just the time needed to execute an invoked function. Resources are scaled dynamically to meet the demand of user requests. Unlike traditional cloud deployment models, where a number of computing instances are deployed well in advance, serverless computing achieves nearly zero resource cost when there is no demand, and scales to as many instances as needed to meet the traffic demand. Thus, serverless computing could be both scalable and cost effective." As can be seen in figure 4, Function-as-a-Service (FaaS) takes away a lot of the infrastructure management tasks needed in legacy infrastructure. Using serverless, there is the need to develop only the function. In legacy infrastructure, server selection, security, schedulers, transaction managers need to be configured and deployed before an app can be deployed.

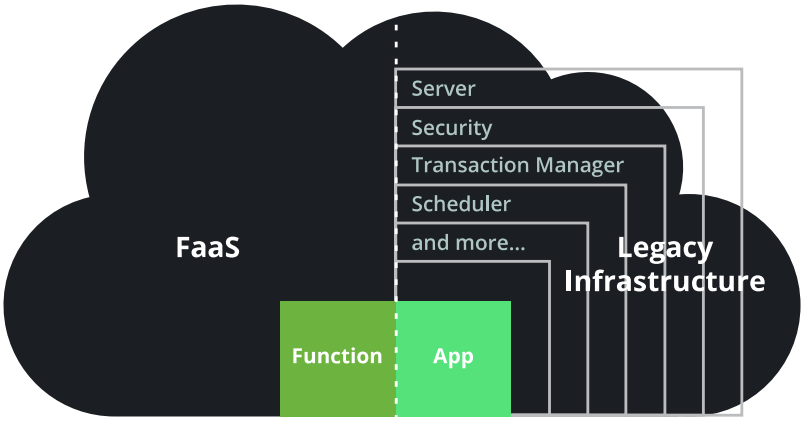


Figure 3: Serverless Infrastructure Vs. Legacy Infrastructure [3]

Serverless is often seen as the successor of virtual machines in the field of virtualization [4] [29]. When comparing serverless with containerization, we see that these technologies are somewhat intertwined. Especially when looking at the open-source framework that are often working on top of a containerization technology as described in section 8.3. Each have their own benefits and use-cases.

Serverless is the overall technology of not dealing with underlying infrastructure. Two applications of serverless technology are Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS) or Backend-as-a-Service and Function-as-a-service respectively. In the following sections, the difference between these two concepts will be further explained.

8.4.2 FaaS

Another term which is often used with serverless is Function-as-a-Service (FaaS). Even though they are often used interchangeably, this is not correct according to [30]. FaaS can be considered a serverless technology but this does not mean that every serverless technology is concerned with specifically functions only. In case of FaaS, a developer can focus solely on the application logic as the code is executed on request (Message queue, Hypertext Transfer Protocol (HTTP) request etc.). Most CSP FaaS Solutions do offer compatibility with multiple well-known high-level programming languages such as Java, Go, Python, Node.JS, C#, Ruby and many more (Compatible with Amazon Web Services (AWS)) The general process of running a serverless application consists of two phases: Function Programming and Function Serving [4]. During **Function Programming** a function is developed by the rules of the specific cloud service provider or framework. The function is deployed to the platform over Command Line Interface (CLI) after the function is saved in a database and the runtimes of the function are pushed to a repository. Lastly, a Uniform Resource Locator (URL) is returned of the functions that can be used to invoke them. The invocation of a function is handled during **Function Serving**. The URL provided by the function programming process can be called manually or through another triggering service such as a message queue or API. When the function is called, the load balancer or scheduler from the serverless platform fetches the function from the database and prepares the environment to run the function, this environment is called a *Sandbox*. After the platform loads the function specific files such as class files, the function is executed. This process is graphically illustrated in figure 4. Big FaaS platforms that are well-known include AWS Lambda, Azure Functions and Google Functions.

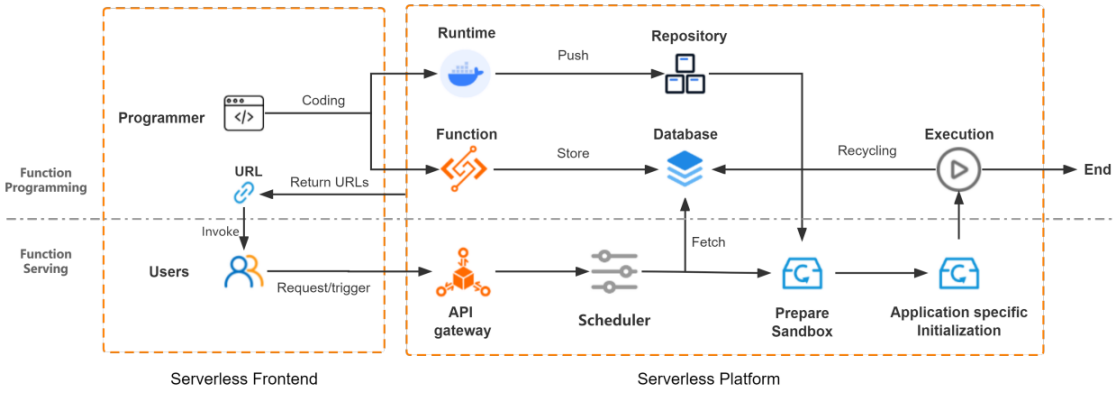


Figure 4: The process of running a serverless function [4]

8.4.3 BaaS

Backend-as-a-Service (BaaS) is another technology based on serverless technology. As described in the previous section FaaS is the actual running of functions on request. On the other hand, application-dependent services deployed by Cloud Service Providers can be considered as BaaS applications. These include services such as Databases and Object Storage Services. In case of BaaS, these services will automatically scale up and down based on the needed resources. By doing so, there is no need to choose resources and storage as a developer.

8.4.4 (Possible) Drawbacks

Of course, serverless technology does not only have benefits. There are a couple of drawbacks which need to be taken into consideration when looking at implementing a serverless platform. The most mentioned possible drawback is the cold start problem of serverless functions. When scaling to absolutely 0, there is no pod or machine running to immediately start the function in case a request comes is. Therefore, frameworks and serverless services came up with the concept of a warm start. Using this concept, a pod or machine is kept running constantly to immediately execute a function when it comes in. This opposes the strategy of serverless where no resources are costing money when scaling to zero, therefore the task and the requirements of the application of serverless need to be well decided upon to prevent additional costs or high latency's. Additionally, it is important to validate that the connecting databases and storage services should be able to handle large amount of concurrent connections when the serverless technology scales up.

8.4.5 Commercial Cloud Service Providers with Serverless services

The most well-known application of Serverless technology is through commercial Cloud Service providers facilitating serverless functions to work on their platform. The most popular CSP's that provide these kind of services are AWS Lambda, Microsoft (MS) Azure Functions, Google Cloud Functions. These platforms are included in the research and comparison of [31]. The three CSP's can be found in table 3.

AWS Lambda	https://docs.aws.amazon.com/lambda/
Google Cloud Functions	https://cloud.google.com/functions/docs
MS Azure Functions	https://docs.microsoft.com/en-us/azure/azure-functions/

Table 3: Commercial CSP Serverless Solutions

8.4.6 Open-Source Serverless Frameworks

Besides the commercial cloud service providers that are offering FaaS services and their associated drawbacks such as vendor lock-in and the heavy reliance on the services of the CSP (storage, message queuing and database) [28], there are many Open-Source Serverless frameworks that provide serverless functionalities to a certain degree. In this section we will be looking at the most popular ones and compare and contrast the functionalities and technologies used by the frameworks. In current literature, studies exist that compare serverless platforms. Articles that are used as input for this section are [28], [31]. We will be comparing the in these articles mentioned serverless frameworks on their properties. In the article of [31], the Open-Source FaaS platforms reviewed can be found in table 4.

Apache Openwhisk	https://openwhisk.apache.org/
Fission	https://fission.io/
Fn	https://fnproject.io/
Knative	https://knative.dev/docs/
Kubeless	https://github.com/vmware-archive/kubeless
Nuclio	https://nuclio.io/
OpenFaaS	https://www.OpenFaaS.com/

Table 4: Open-source Serverless Solutions

In the paper of [28] 4 open-source serverless frameworks are analyzed on their characteristics. These are Knative, Kubeless, Nuclio and OpenFaaS which are all taken into consideration in the paper of [31] too.

8.4.7 Interoperability across cloud vendors

In order to ensure interoperability across multiple cloud vendors and by doing so preventing vendor lock-in the code needs to be written in a compatible way or through a specific framework such as Spring Boot [3] or one of the open-source frameworks. Serverless services of commercial Cloud Service Providers such as AWS Lambda or Google Functions often are made to work with additional cloud services of the corresponding Cloud Service Providers (CSP).

9 Requirements

In this chapter, requirements for the target architecture using serverless are defined. Requirements are obtained through literature research and expert interviews. The chapter will be concluded with a summary on the requirements.

9.1 Method Expert Interviews

In order to come up with an effective design, it is important to integrate the requirements of stakeholders into the design [32]. Besides the literature research, interviews are used to gather an in-depth insight into experiences and expert knowledge. There are multiple possibilities for conducting an interview. On the one hand we have the surveys or questionnaires with mostly closed-ended questions. On the other hand, we have focus groups where people are engaged in brainstorming to enquire extended knowledge on a certain subject or design [33]. A third approach falls between these two and is well-known as a semi-structured interview (SSI).

The process of setting up a semi-structured interview has 3 main phases. The first step is selecting respondents and arranging the interviews. Because of time constraints and not a very large sample group, it is decided to select a group of people based on their function profile and the associated knowledge on a certain subject.

Secondly, the interview script is created. In this script it is important to state the goal of the interview, how the answers are collected, how the answers are stored and who is allowed to see them in the introduction. After this introduction, the subjects and some directions for questions are written to guide the interview and make sure agenda points are not skipped leaving questions unanswered.

The interview script can be found in appendix 14.5. The interview is carried out with experts in the field. A total of 4 interviewees from Dutch iPaaS provider eMagiz are questioned. The first interviewee is a Software Delivery Manager responsible for the delivery of software. Secondly, the Chief Technical Officer (CTO) is interviewed. Thirdly, an external Cloud Architect working at eMagiz but also at other Cloud Projects is interviewed before we concluded the interviews with the Cloud Engineer of eMagiz.

Lastly, the answers of the interview need to be analyzed and converted into meaningful input for the Functional Requirements (FR) and Non-Functional Requirements (NFR) sections 9. This is done through selecting the mentioned and described requirements and plotting them on a MoSCoW prioritization table. MoSCoW is the most cited method as used in software requirements prioritization research and therefore a good fit for this specific research [34].

After the individual interviews, a group discussion took place with the interviewees to discuss the prioritization of all the identified requirements. This resulted in the prioritized requirements as can be found in chapter 9.

9.2 Interviewees

The expert interviews are all conducted with employees of a Dutch iPaaS platform called eMagiz. This company is based in Enschede with over 20 years of experience in integration software.

The first expert interview was conducted with a delivery manager of the iPaaS platform. Being a delivery manager, he is responsible and accountable for the development team. Additionally, the finances and procurement concerning the Cloud Service Costs are managed and checked by this person.

The second interview was conducted with the CTO of the iPaaS platform. As a CTO, this person is responsible for the overall direction where the platform is heading. Another aspect, the CTO is concerned with, is the licensing of the software and the licenses of the used frameworks and packages.

The third interview was conducted with a Cloud Engineer being responsible for the cloud infrastructure of the iPaaS platform.

The fourth interview was conducted with a Cloud Architect working on the iPaaS environment. The architect is responsible for the in-depth configuration of the cloud services needed to run the iPaaS environment. This person is working in Information Technology (IT) for over 25 years and therefore has been through many technology transitions (on-premise to Cloud, rise of the internet etc.) making him a very valuable expert in the field.

Since every interviewee has different field of knowledge and expertise, not all questions from the interview script were asked to the same people. Based on their role and expertise, questions touching upon this knowledge is asked and documented from every interview.

This leads to 5 main categories of requirements: Costs, Legal, Security, Usability and workload.

9.3 Costs

During the interviews we could see that 3 out of the 4 interviewees are familiar with cloud costs to a certain degree. One person is especially responsible for all the cloud costs within the iPaaS company. As a result, a good insight into requirements on costs could be obtained during the interviews. These result in the requirement that an estimation on the costs of the new architectures needs to be constructed in the prototype phase and that the new architecture should lower or match the current costs of the infrastructure. The new architecture cannot exceed the costs of the current architecture.

9.4 Licenses

As with most software engineering requirements there are legal and regulatory requirements too. During the interviews specific concerns about licenses and certificates were expressed. When looking at licenses, it is important that the solution, whether this is a commercial or open-source solution, can be used in a commercial product. When looking at software licenses, certain licenses do not allow for commercial use making it unusable for this specific use-case. Additionally, literary sources such as [31] state that "The business view of our FaaS Platforms Classification Framework comprises categories and dimensions of interest for project managers aiming to identify the FaaS platforms complying with the high-level project requirements. These include, for instance, the license under which a FaaS platform is released and whether the platform can be installed on-premise or not". This confirms the importance of such as requirement in this case.

9.5 Security

Since the iPaaS solutions can be used to transform and move sensitive data of the customers, security is an very important aspect. Often customers demand certain certifications for a solution when it comes to security. In the case of our interviewees, this is no different as they are ISO 27001 and SOC2 compliant. In order to adhere to these compliance's the requirements of isolation, availability and security come up. Specific requirements concerning security and isolation would focus on data being securely moved within the iPaaS platform and not accessible for the outside world or other customer environments. Especially in the case of serverless where resources are often shared between different environments to benefit the most from the scaling feature, attention needs to be paid to this security aspect.

9.6 Usability

In order for the solution to lead to a successful implementation, it is important that the developers and cloud engineers can work with it without too much of a hassle compared to the current workflow. Therefore it is important that the solution is compatible with a currently used language such as Java but also with current infrastructure products and frameworks such as kubernetes or Spring Boot. In addition to this, the framework should be well-maintained in order to prove a certain quality and believe in it from the community. Certain factors that can be used to measure this can include Github stars, commits and contributors but also big companies that did adopt the technology. In the article of [31] they say the following about this aspect: "The Community category enables to classify FaaS platforms based on the size, activity, and popularity of their development community."

9.7 Workload

In 2 interviews, it was mentioned that the current workload of the cloud team (responsible for running and developing the cloud infrastructure) is substantial. This is mainly due to the developing and managing of a kubernetes cluster for running the new generation of the iPaaS solution. It is mentioned that this should decrease when the cluster is completed but as the iPaaS platform continues to grow and more customers are using it, it is expected to be a fairly large workload for this team. Running certain components within the iPaaS platform on a FaaS framework should decrease the amount of work involved in managing infrastructure and thus decreasing the workload for this team. For now it is not expressed as a very high priority in the interviews conducted with the experts.

9.8 Summary

The interviews and literature research lead to a list of requirements which are listed below. We distinguish between Functional Requirements (FR) and Non-Functional Requirements (NFR). The MoSCoW prioritization has been applied to these requirements after a group discussion with the interviewees as described in the method.

9.8.1 Functional Requirements

FR1. The new architecture with serverless ***should*** reduce the workload for DevOps teams where infrastructure needs to be configured and maintained.

FR2. Based on the workload of the system, the infrastructure ***must*** scale up to accommodate the increased workload.

FR3. When there is no demand for a certain function to be ran, the machine(s) ***should*** scale to zero.

FR4. For the new architecture, a cost estimation ***must*** be made as the new solution ***should*** lead to a similar or lower cost than the current architecture.

9.8.2 Non-Functional Requirements

NFR1. The serverless solution ***should*** have a license that allows an iPaaS provider to use it in a commercial product.

NFR2. The serverless solution ***must*** be well-maintained / actively developed.

NFR3. The serverless solution ***should*** be used by some big companies in order to prove a certain magnitude within the field.

NFR4. The serverless solution ***should*** natively support the programming language (Java) as often used by standard iPaaS solutions.

NFR5. The new architecture ***must*** take security and isolation (of client environments) into consideration.

NFR6. The chosen solution ***should*** be open-source to provide CSP-agnostic capabilities.

NFR7. The chosen solution ***must*** be compatible with Kubernetes container orchestration framework.

NFR8. The chosen solution ***should*** be compatible with iPaaS typical triggers.

NFR9. The chosen solution ***could*** be having a business support in order to work through high-impact problems.

10 Serverless Solutions

In this chapter, different open-source serverless frameworks are discussed and compared taking into account the requirements resulting from the expert interviews and literature research in chapter 9.

In order to create a first selection of possible candidates for further research, we look at the "must" requirements which can be answered through literature research. This leads to the requirements: 1. "The serverless solution must be well-maintained / actively developed." (NFR2.), 2. "The chosen solution must be compatible with Kubernetes container orchestration framework."(NFR7.) and 3. "Based on the workload of the system, the infrastructure must scale up to accommodate the increased workload." (FR2.).

When eliminating solutions that do not fulfill requirement NFR2. We see that frameworks such as Kubeless and Fn Project are not actively maintained anymore and therefore do not need any further investigation.

When we apply the 2nd requirement (NFR 7.) concerning Kubernetes compatibility to the resulting candidate solutions, we find that all 5 frameworks do work with Kubernetes and have support for it. This results in OpenWhisk, Fission, Knative, Nuclio and OpenFaaS.

Lastly, the 3rd requirement (FR2.) about automatically scaling up is applied. Only Nuclio has no support for this feature and therefor is not taken into consideration for this investigation.

Finally, 4 resulting frameworks are further investigated as a possible solution for use within an iPaaS platform; OpenWhisk, OpenFaaS, Knative and Fission. We will take a closer look at these 4 and compare and contrast them in the following section.

10.1 OpenWhisk

The first developments for OpenWhisk are dating back to 2015 when a small team of researchers within IBM research started working on serverless technology. A year later, the framework became open-source on GitHub. IBM continued in the serverless scene with the development of IBM Cloud Functions (A commercial CSP product). OpenWhisk became part of the Apache Software Foundation in December 2016 [35].

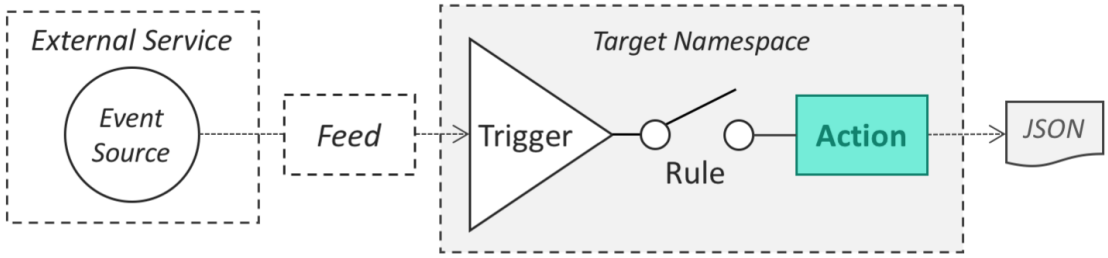


Figure 5: The process of invoking an action within OpenWhisk [5]

OpenWhisk has a wide range of supported programming languages. The following runtimes can be used with OpenWhisk .Net, Go, Java, JavaScript, PHP, Python, Ruby and Swift. Other language which are not natively supported, can be ran using the Docker SDK on the Docker Runtime.

OpenWhisk actions (stateless functions) can be called in multiple ways. Actions van be invoked using the OpenWhisk CLI, the OpenWhisk REST API, user-created API's or automated by triggers (classes or events send by event sources). This process is shown in figure 5.

OpenWhisk can be ran on an cloud-based Kubernetes cluster. Additionally it runs on serverless providers hosting Apache OpenWhisk and support the OpenWhisk CLI such as IBM Cloud Functions.

Website: https://openwhisk.apache.org/

10.2 OpenFaaS

OpenFaaS is one of the oldest framework around and launched in 2016 where it was initially founded by Alex Ellis. Besides the free community edition of OpenFaaS, OpenFaaS Ltd - the company behind OpenFaaS offers enterprise support for production environments making it an overall mature framework. In 2020 they held the second position with a 10% user base. OpenFaas is also the most popular framework on GitHub with 22.3K stars (November 2022). OpenFaaS has the ability to do auto scaling as well as scale to zero. The function repositories and the framework itself can be managed through the faas-cli tool as well as a through a GUI in from of a web interface. Several programming languages are supported, these include NodeJS, Python, Java, Ruby, PHP, Go and C#.

Although OpenFaaS can scale down to zero, they recommend to keep one function replica available to cope with the cold start problem. This can be a limitation for functions that need a fast response time.

OpenFaaS has multiple triggers that it can work with. These include HTTP/Webhooks, NATS Streaming and CLI. The Pro version of OpenFaaS had additional trigger services which include: Apache Kafka, Postgres, AWS SQS, Cron Connector, MQTT Connector, Minio / S3, NATS Pub/-Sub, AWS SNS, CloudEvents and RabbitMQ.

OpenFaaS has an extensive list of adopters on their github including companies like VMware, Citrix, DigitalOcean and Bulletproof.

Website: https://www.openfaas.com/

10.3 Knative

Knative is a younger framework compared to the previous 2. Knative started in 2018. It can do auto scaling and additionally scale to zero too. Knative works on the basis of serverless containers in cooperation with Kubernetes. By doing this, it creates a serverless like development pattern but in the end it cannot be considered a real serverless framework like the other 3 candidates. As it might still be a good candidate for our use-case, it will be left in the comparison for now. Knative is trusted by multiple big companies to run their serverless functions. These companies include VMWare, IBM, Red Hat, Google and TriggerMesh [36].

Website: https://knative.dev/docs/

10.4 Fission

Fission works with Kubernetes too. The first release of Fission on Github dates back to 2017 and the last updates and new versions are still released in 2022 being actively maintained and developed. Python, NodeJS, Go, C#, PHP are officially supported programming languages. It does support auto scaling based on Central Processing Unit (CPU) usage (it mentions that in the future, other scaling metrics are implemented as well) and has a very fast Cold-Start of typically 100msec making it stand out from other solutions. Fission mentioned many big users on their Github Page which include companies as Apple and Unilever.

Website: https://fission.io/

10.5 Summary on remaining serverless frameworks

In this section, a comparison amongst the 4 chosen serverless frameworks is made and the results are summarized in table 5 to create a concise overview for helping making a choice on the suitable framework for use within an iPaaS solution.

	OpenWhisk	OpenFaas	Knative	Fission
Supported Programming Languages:	.Net, Go, Java, JavaScript, PHP, Python, Ruby, Swift.	.Net, C#, Go, Java, JavaScript, Python, Ruby	C#, Go, Java, JavaScript, PHP, Pyton, Ruby, Rust	C#, Go, NodeJS, PHP, Python, Ruby, Bash
Supported Triggers:	HTTP/Webhooks, Timers(Alarm)	HTTP/Webhooks, NATS Streaming, CLI, Apache Kafka, Postgres, AWS SQS, Cron Connector, MQTT Connector, Minio / S3, NATS Pub-/Sub, AWS SNS, CloudEvents, RabbitMQ	HTTP/Webhooks, Message Queus, Github, GitLab, PingSource (Timers), RedisSource, RabbitMQ, Diverse 3rd Party apps to work with Amazon, Azure and Google cloud services.	HTTP/Webhooks, Message Queues, Timers, Kubernetes Events
License:	Apache V2.0	MIT	Apache V2.0	Apache V2.0
Used by big companies:	IBM (Apache Foundation)	VMware, Citrix, DigitalOcean and Bulletproof and many more smaller companies.	VMWare, IBM, Red Hat, Google and TriggerMesh	Fareye, Apple, iQuanti, Gadget, CinnamonAI, Armo, The Social Audience, KubeML, Unilever, BD, Biofourmis, Babylon

Table 5: Summary of important properties of the resulting serverless frameworks.

11 Architecture

In this chapter, a model of the baseline architecture will be described. This will act as a starting point for the target architecture which implements the previous research of this thesis in order the come up with a solution for the main problem at hand. Then, a gap analysis is conducted in order to find out where and how certain architectural changes are contributing to solving the problem. This gives an input for the implementation steps which need to be taken in order to convert from the baseline to the target architecture. This will serve as input for the next chapter, the prototype.

11.1 Method Solution Design/Architecture

In an effort to create a design that suites the current architecture of the eMagiz iPaaS platform it is important to get an insight in the current architecture of the platform. In order to do this, we will be using Archimate Modelling to create a baseline architecture. In this baseline architecture, current business processes and the application and technologies that enable these business processes are modelled. This is done through researching images, models and presentations of eMagiz. After a first modelling session, the model is discussed and fine-tuned with employees of eMagiz (CTO, Cloud Engineer and Delivery Manager). When the as-is situation is made clear through the baseline architecture model, we will continue with the to-be situation through the target architecture. This will be the architecture which still supports current business processes but with the implementation of the serverless solution as decided on in the literature research. When the baseline and target architecture are clear and approved by the company, we can start identifying how to bridge the gap between these two architecture. This will result in a gap analysis which reports the main differences and changes that need to be made in order to transform the baseline architecture into the target architecture. Finally, this will result in an implementation plan which will be tested by actually developing a prototype of a serverless iPaaS platform.

11.2 Baseline Architecture of iPaaS

In order to find out where a serverless implementation would make sense and how to do this, the 'as is' situation needs to be modelled in a baseline architecture.

In order to come up with an iPaaS baseline architecture, a couple assumptions are made: All iPaaS providers host their cloud environments at AWS, Azure or Google Cloud. [37] [38].

We will be looking at two main services of an iPaaS platform. The setup of a new customer environment after an integration has been designed and the actual customer environment that receives, converts and sends messages across the platform and to other applications or endpoints. We chose these two services as these are the mostly dealing with the infrastructure where the iPaaS in running and therefore will probably be impacted the most by our implementation.

Only the parts of the architecture that are relevant to the scope of this investigation are modelled.

11.2.1 The business processes

The first process which we will describe and model is the process of launching a new integration within the iPaaS environment. A customer can do this for example when a new piece of software is added to their internal stack and they want the data to be synchronized with already existing systems to improve data quality and reliability. The new integration is send to developers to test that the integration is working correctly and yields the correct results. A Virtual Machine (VM) is chosen which runs the integration. For now this machine's size is based on experience of the developers by other integrations. After testing and validating the integration, it is send to the portal and activated (deployed) in order to start processing the messages.

This deployment process is described in figure 6. When we start at the left side of the model we find that a customer or a partner of a customer starts an implementation by determining technical

requirements. This includes figuring out the formatting of the data, how to connect to existing external services etc. Currently, when the technical requirements are clear, eMagiz developers choose a adequately sized machine that is able to run the chosen integration. As auto-scaling is no option in the current architecture, this is an important process in order to create a good customer experience and deliver a qualitative service. Developing, testing and deploying the integration are business processes which are taken care of by the platform itself after it has received a correctly configured Virtual Private Cloud (VPC) environment. This whole process contributes to the main business process of the platform, namely providing an integration platform. Other processes that complete this main process are described in the following sections.

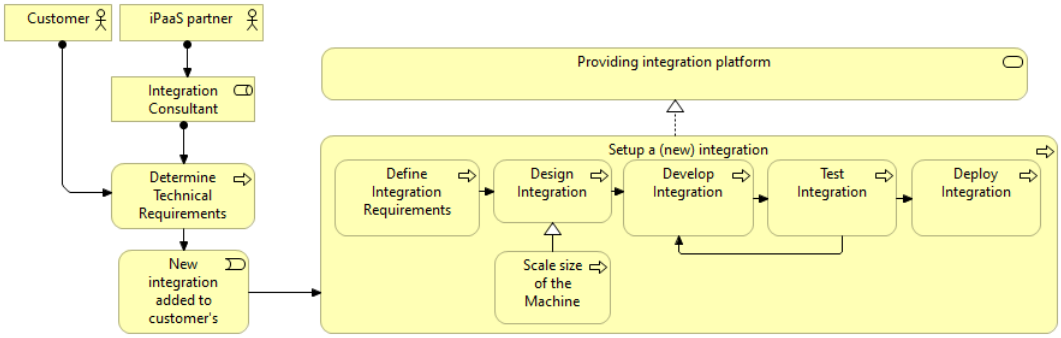


Figure 6: The deployment process in the current architecture.

Of course the service of providing an integration platform is not limited to just deploying an integration environment but also consists of managing and actually providing the integration services for the customer. Therefore two more business processes are added to the architecture; Processing the customer's data and managing the integration. In order to process data of the customer, three main components or products that make up an typical iPaaS stack are described as API Management, Event Streaming and Messaging.

In order to run the integration successfully, management of the integration is important in order to act upon distortions and incidents. This is done by gathering metrics from the running integrations and by implementing certain thresholds, create alerts and send these to developers and support teams in order to mitigate possible incidents. Additionally, customers can often set thresholds and alerts themselves as well to manage certain incidents which are dependent on their own software and processes.

The total business processes architecture can be found in figure 7. The 3 processes together make up the business service 'Providing an integration platform'. Additionally, managing an incident can include changing or fixing a integration, hence the relation between these two processes.

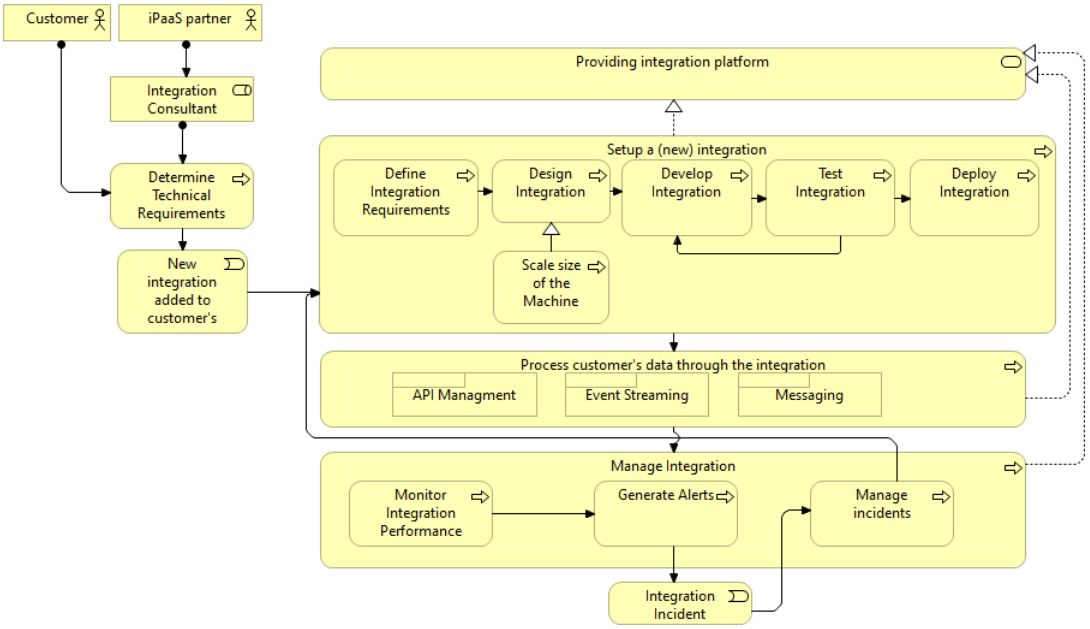


Figure 7: The business processes that are conducted by an iPaaS provider.

11.2.2 Applications and Technology

Now the business processes of a iPaaS provider are clear, we need an underlying technology and application layer to actually support these business processes. A couple of assumptions are made in order to create a suitable model. Technology and Applications assumptions:

1. iPaaS providers run some sort of portal where a customer can create certain message- and data pipelines. These pipelines are deployed to a Cloud Service Provider when finished.

2. Every customer has it's own Virtual Private Cloud environment sized to their needs.

3. iPaaS providers have an 'Infra Cloud' that takes care of general tasks that are needed by every customer and are not customer specific. These services include a general Domain Name Service (DNS) Resolver, Metrics and Logging and Image Storage.

In the technology view as visible in figure 8 we can observe one main application process and two main application components. The process deploying an integration consists of two functions being creating and building the image and setting up or provisioning this image to the infra cloud Image Storage. The building process itself is initiated from the Portal Service where customers create or code their specific flows. When a specific image is built and deployed, we can observe the customer VPC being deployed to a cloud slot within the cloud platform. This VPC is than specifically configured and scaled for a specific customer. Infra Clouds are machine configured by the iPaaS providers to support multiple processes of an iPaaS provider that are not customer specific. This included application services such as the main DNS Resolver, Metrics and Logging storage and Image Storage.

In order to process messages and data send to the platform, customers connect their systems over the internet to the main DNS Resolver in the Infra Cloud. Based on certain rules, this resolver knows which type of messages or from a certain destination belong to a certain customer and than routes this data to the correct Customer VPC. When messages arrive at the VPC, an internal Resolver

and Loadbalancer distribute the messages to the correct workers to start handling them. Depending on the configuration of a customer, these workers can do actions such as translate formats, post to a bus or queue or act as an API gateway.

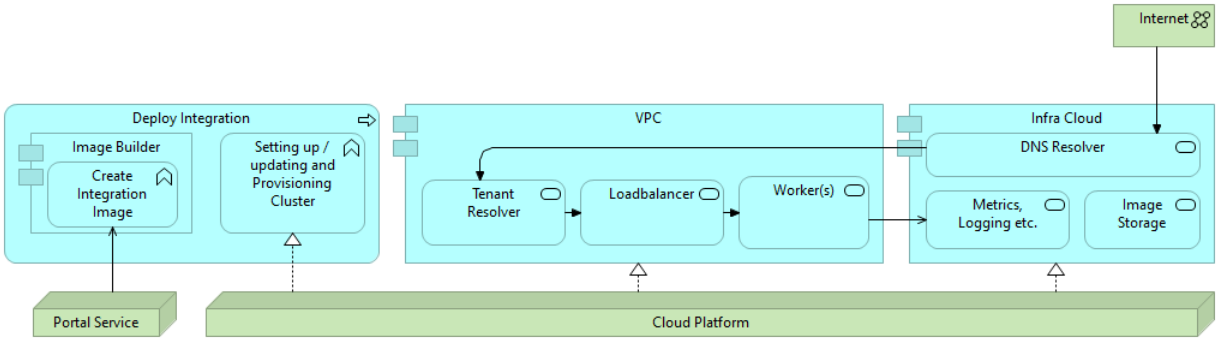


Figure 8: Technology and application of the baseline architecture.

11.2.3 Overview of baseline architecture

When we combine the business, application and technology layer and connect the application services and components to the business processes we can observe which part of the application layer is responsible for serving which business process. The total overview of the baseline architecture for current iPaaS providers can be seen in figure 9. This will be the starting point for the target architecture where the improvements are added based on our previous research as described in the previous chapters.

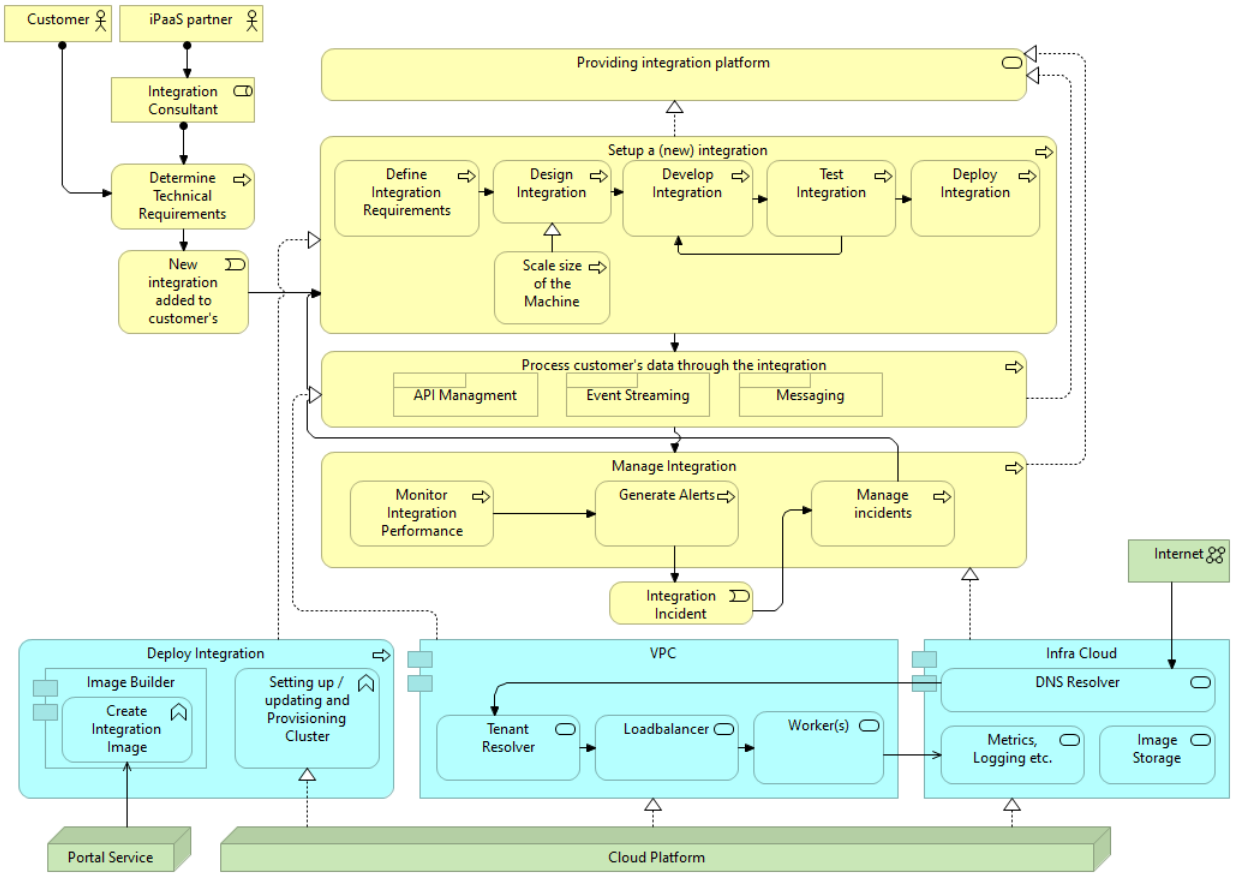


Figure 9: Baseline Architecture Overview

11.3 Target Architecture of iPaaS

In order to improve the problems as identified in the previous section, a to-be or target architecture is modelled. The main architectural problem of making the architecture more scalable ready is addressed by moving from a VPC based architecture to a kubernetes cluster with an serverless framework running on it.

It is important to note that the business processes do not change. From the viewpoint of an end-user there are no immediate changes visible in the software. Most changes are applied "under the hood" in the architecture of the infrastructure to optimize the use of cloud infrastructure and therefore decrease costs.

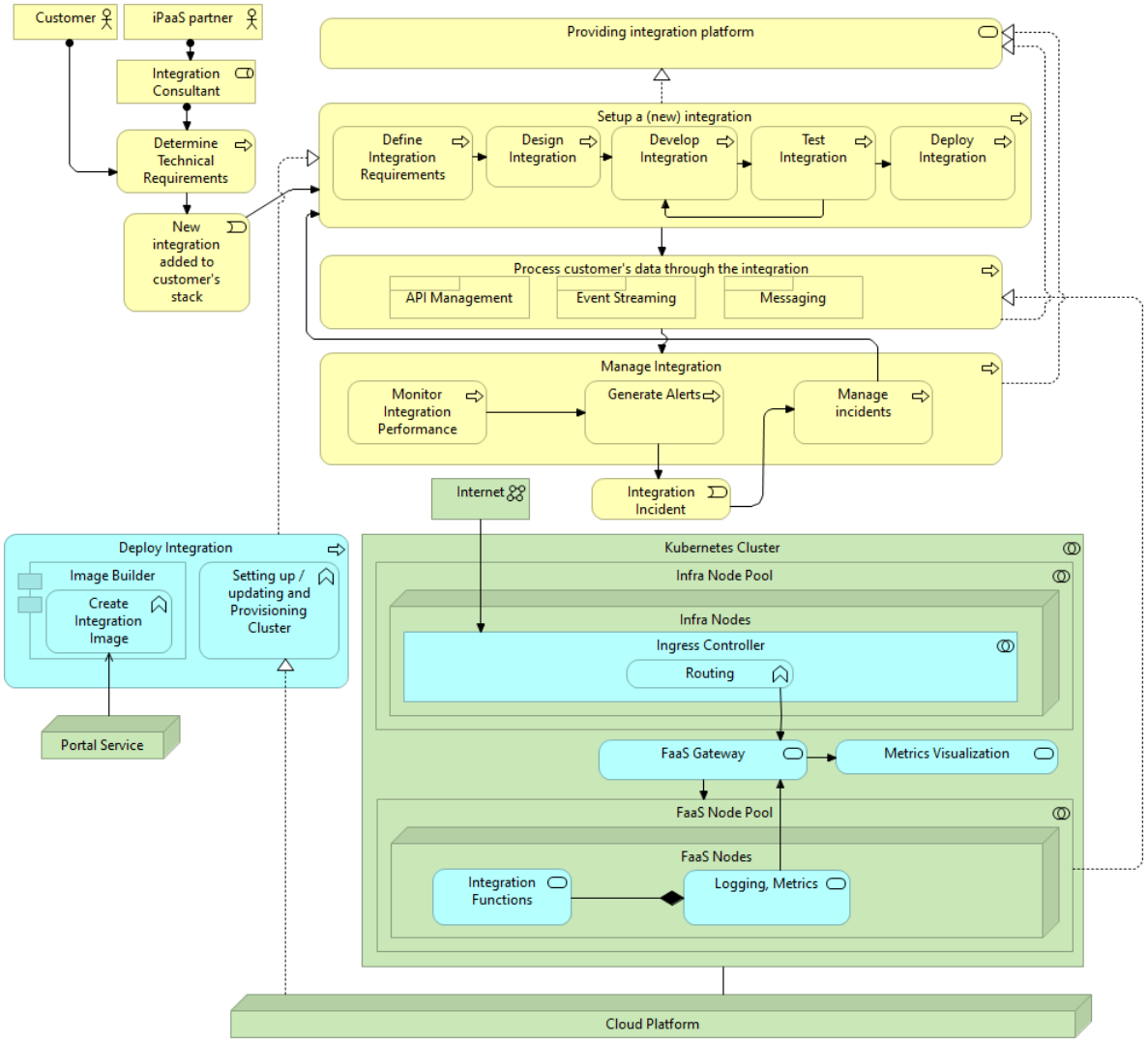


Figure 10: Target Architecture Overview

11.4 Gap Analysis

In this section we will be investigating the steps that need to be taken in order to move from the as-is situation with the baseline architecture to the to-be situation with the target architecture. In order to structure this section, we will be looking at the problems that may arise in the baseline architecture and how that will be improved with the target architecture.

11.4.1 Scalability

In the baseline architecture we find that because of the use of VPC's for every customer's environment, scaling is difficult. Especially when a customer's environment has a lot of scheduled batch processes that cause spikes in traffic and workload for the environment, it is possible that the VPC runs out of resources. When this happens, environments can become unstable and slow or even can lead to possible loss of data. As this can be disastrous to the customer this should be avoided. In the target architecture we find that a customer's environment is no longer based on non-scaling Virtual Private Cloud but on a scaling FaaS framework. By doing so, the customer's environment will be able to scale in order to accommodate rises and spikes in workload and traffic to handle. Additionally, when there is not a lot of workload going through the environment it will scale down to a minimum as well. The functionality is called auto scaling and can be configured on specific metrics such as requests per second or CPU and Memory usage.

11.4.2 Resource Efficiency / Cost optimization

Every VPC is sized to accommodate peaks to a certain degree in the customers' integration environment. By doing this, it automatically results in most VPC's being oversized for their workload under normal circumstances. This leads to inefficient use of resources of Cloud Providers. By moving towards the target architecture we have a scaling architecture that optimized resource usage by running multiple environments and functions on a shared pool of resources. As a result, machines need less overhead to run tasks and not every environment needs its fully own VM. As less resources are needed to run the same task, cloud infra costs are likely to decrease too.

11.4.3 Resiliency

Another benefit of the target architecture over the baseline architecture concerns the aspect of resiliency. Virtual machines are prone to failure in case of software issues or network disruptions. Of course in the target architecture with a FaaS framework running on Kubernetes these issues are still possible but it has built-in tools to accommodate these errors. For example when a function in a FaaS environment becomes unresponsive or fails, the workload is taken by other instances to guarantee correct handling of the workload. The same holds for the infra side of the target architecture where services are restarted and auto-healed by Kubernetes mechanisms in case of a networking problem.

11.4.4 DevOps Automation

In current architecture, a lot of manual steps are involved when looking at the business process of setting up a (new) integration. Especially when looking at the process of designing the integration, manual input from developers in needed in order to correctly size the Virtual Private Cloud (VPC) environment of the customer. In the new architecture, this sizing and manually buying of resources is no longer needed as the environment automatically scales based on the workload of the client environments

11.4.5 Overview of required steps

In order to create the target architecture starting from the identified baseline architecture, certain steps need to be taken. These will be briefly explained here.

1. The first step is to create an cloud environment capable of running the serverless Framework and it's functions. In order to do this, I would advice to look at the current Cloud Service Provider and create a cluster in the Kubernetes service of this CSP. In case of eMagiz this will be the AWS Elastic Kubernetes Service (EKS). By doing this with the same CSP as currently used in the baseline there is no additional change in GUI or for example billing structure.

2. The second step will be to install the serverless framework on the Kubernetes Cluster and configure it to scale correctly based on the specific function it will carry out. Some functions cannot scale to zero as they need to quickly react while others can scale to zero and work with a slight delay.

3. Additional services needed to run the new environments need to be installed. These include but are not limited to the metrics, dashboarding applications, loadbalancers, ingress services, elastic storage services etc.

4. The most influential step will probably be the one that takes care of adapting the Java applications into "serverless ready Java Functions". This will need some rethinking of how the applications are currently working and how they can be divided into separate functions. A lot of testing needs to be done in this step to confirm correct working before moving it into development environments.

5. After the testing and confirmation of correct implementation of the serverless version of the Java applications we can start to connect to external services such as Kafka Event Streaming busses and the Graphical User Interface (GUI) portal where customers create the flows. We can see if changes in the portal reflect correctly in the serverless environment.

6. Different customers have different requirements and use different services in their stack. Therefore, for most implementation environments an implementation planning should be considered to check whether every environment will successfully run in the new infrastructures. As a result, there should be a transitioning period where customers move over to the new architecture in staged periods. By doing this, environments can be reverted back to the old environment in case something does not work as anticipated and there should be only minor disruptions for the customers.

7. Finally, when customers are gradually moving over to the new infrastructure with serverless technology, it opens up new doors to move to a more pay-per-use way of pricing instead of the one size fits all pricing per license model which is currently in place.

12 Prototype

In order to validate the remaining requirements and to test whether or not our target architecture addresses the identified problems, a prototype is created. In this prototype we will be installing two open source serverless frameworks on a kubernetes cluster deployed on a public cloud provider. We will be deploying two java functions (one synchronous and one asynchronous) to these frameworks in order to test if they would work with typical workloads as used by eMagiz iPaaS.

12.1 Prototype

The last step before validation can be started is the actual production and development of the prototype environment. According to Arnowitz et al. (2007) [39] a prototype in software making can be useful to determine economical feasibility, evaluate stakeholder's response to the product and if the design solution or idea will actually work in the set environment. In order to create the prototype, the scrum method is applied as agile framework. The total prototype will be developed in 4 1-week sprints. After a basic prototype is delivered, small adjustments can still be made in order to validate certain requirements. Such changes can for example be adding logging and metrics capabilities in order to obtain quantitative data to validate certain requirements.

12.2 Cloud Provider

The first step of the prototype was to choose a public cloud provider that support a Kubernetes cluster. The most well-known providers are Amazon Web Services (AWS) Google Cloud and Microsoft Azure Cloud. All of these have a specific service for running Kubernetes clusters, namely Amazon Elastic Kubernetes Service (EKS) [40], Google Kubernetes Engine (GKE) [41] and Azure Kubernetes Service (AKS) [42] respectively. After installation of the cluster, managing the cluster is mostly done through the CLI tool Kubectl and through the specific serverless framework CLI's. As a result it does not matter where we deploy our cluster when looking at functionality. Therefore we started looking at the costs of the different cloud providers as we want to be the prototype as cheap as possible. On GitHub there is a list with Free Kubernetes trials and/or credits [43] which we used to select our Cloud Service Provider for this Prototype. Google's GKE offers a free trial with $300 credit for 90 days. When you sign up and validate with a business mail address (@business.com) you get an additional $100 credit. Microsoft Azure (AKS) has a similar offering where a $200 credit can be used for a 30 days period to run a kubernetes cluster. AWS does not have a free offering as their free tier does not cover their EKS and needed EC2 instances. Taking these offerings into consideration, the decision to work with Google Kubernetes Engine (GKE) for this prototype was an easy one. Because pricing is an important requirement in this investigation, a check was done on pricing across other Cloud Service Providers such as AWS and Azure. It was found that all three of them have comparable prices for running an Kubernetes Cluster.

12.3 Kubernetes Cluster

In order to create a kubernetes cluster for every serverless framework two different Google Cloud accounts are created with the trial credits added to them. OpenWhisk and OpenFaas both have different system requirements when it comes to the underlying Kubernetes Cluster. In order to fulfill these requirements we setup the following two Kubernetes Cluster in Google Kubernetes Engine as found in table 6. As we are working from Europe it would have made more sense to choose a European zone for the clusters to run in but as the America zones are cheaper, it is decided to go with these instead for the prototype.

	OpenFaas:	OpenWhisk:
Control plane and Default Node zones:	us-central1-a	us-central1-a
Version:	1.25.7-gke.1000	1.25.7-gke.1000
Total size:	2	2
Machine type:	e2-medium	e2-medium

Table 6: Kubernetes Clusters on GKE

The clusters are provisioned through the command line interface (CLI) of Google Cloud. By using this tool, it is possible to use a set of "gcloud" commands to configure the clusters through code and therefore prevent possible difference between the both install when we would be using the GUI. The following command is ran in the CLI to install the clusters as specified in table 6:

```
gcloud container clusters create openwhisk \
    --cluster-version=${k8s_version} \
    --zone=us-central1-a \
    --num-nodes=2 \
    --machine-type=e2-medium \
    --no-enable-cloud-logging \
    --disk-size=30 \
    --enable-autorepair \
    --enable-network-policy \
    --scopes=gke-default,compute-rw,storage-rw
```

When having a closer look at this command we can see that the cluster name is set in the first line, in this case OpenWhisk but we will be creating one for OpenFaas as well. On the first line the kubernetes version of the cluster can be specified. At the time of writing, the default stable version is 1.25.7-gke.1000 on Google Cloud which we will be using. The zone specifies in which geographical zone we want our cluster to be running. In order to save costs we will be using a US zone instead of an EU zone for this prototype. The number of nodes in the cluster will be 2 in order to start off with. This can be re-scaled up and down in a later stadium. The machine type is e2-medium (consisting of 2vCPUs and 4GB memory). This is the minimal recommend size for OpenWhisk. OpenFaas can run on less as well but in order to make sure both installations are running on comparable clusters, we decided on the e2-medium machine type. We will not be using logging of the Google Cloud environment. Disk size is 30GB as recommended to run OpenFaas and OpenWhisk. When a node stops or breaks, we want it to automatically restart en repair the cluster. Therefore, we enable the auto-repair variable. The last two lines apply some policies to the cluster to make sure it has the correct rights to work correctly.

After some minutes waiting, both clusters are up and running and we are ready for installing the FaaS platforms as described in the next section.

12.4 OpenFaas installation

After both clusters are configured and running in the Google Cloud, it is time to start installing the chosen FaaS frameworks on it. In order to install the packages needed for OpenFaas we will be using a package manager for Kubernetes. In this case this is HELM [44]. We can install the HELM package manager with the following command:

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/
    ↪ scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

At the time of writing, we are using helm version v3.11.3. which is the most recent stable release. After the HELM is installed we can install OpenFaas using the helm charts.

Specifically for OpenFaaS there is another piece of software that helps installing and provides many other useful pieces of software such as kubectl. This is called Arkade and is from the same developer as OpenFaas. We install Arkade in a similar fashion as helm:

```
$ curl -sLS https://get.arkade.dev | sudo sh
```

After Arkade is installed, we use it to install OpenFaas as well:

```
arkade install openfaas --load-balancer
```

When we look briefly over this command we can find that we want to install the OpenFaas in the namespace ”openfaas”. We want to use a LoadBalancer to expose the OpenFaas services to the world so we pass an extra flag. Other options for this flag include using an NodePort or Cluster IP with port-forwarding. This is mostly used in development environments and less suitable for our prototype as it needs to connect to other endpoints and does not support TLS.

Now that OpenFaas sussecfully installed on our cluster we need to obtain some information about the installation. In order to connect to OpenFaas we will need to have the IP address of the external load balancer that exposes OpenFaas to the internet. We obtain this information through the following commands:

```
kubectl get svc -o wide gateway-external -n openfaas
```

When accessing the IP through a web browser we are prompted to enter the username and password which can be obtained through the following command:

```
kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-password
    ↪ }" | base64 --decode; echo
```

When entering all the information we can see a successful deployment of OpenFaas on our cluster and we can start deploying some example functions through the OpenFaas ”function store”. This can be found in figure 11.

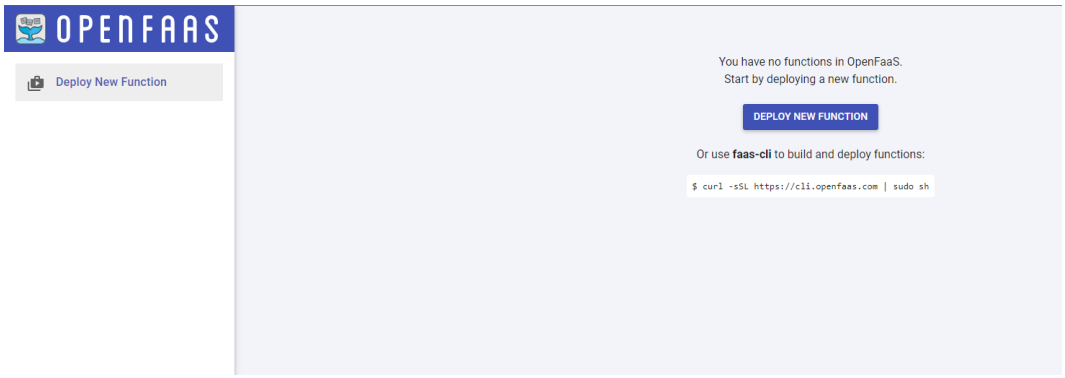


Figure 11: The OpenFaas Web interface after successful deployment.

On this page a tip to use the faas-cli can already by found. faas-cli is the Command Line Interface (CLI) tool to manage and deploy functions on OpenFaas. We will not be installing this on the Google Cloud CLI but on the computer where the functions are developed. In this way, functions can easily be build and deployed from the Integrated Development Environment (IDE) to the OpenFaas environment.

We install the faas-cli with the following command:

```
curl -sSL https://cli.openfaas.com | sudo sh
```

By default, faas-cli will try to connect to an OpenFaas instance running as localhost on the same machine. Since we are working with a cloud hosted OpenFaas instance we need to tell faas-cli where we want it to connect to. We can do that by specifying the URL:

```
export OPENFAAS_URL= "IP of OpenFaas Instance"
```

Now that the tool knows where the OpenFaas instance is located we need to login to it in order to execute commands. We use a command that logs in and saves a file to /.openfaas/config.yml for following sessions:

```
echo -n "Password of the installation here" | faas-cli login --username admin --
    ↪ password-stdin
```

That is it for the OpenFaas installation for now. In the upcoming sections we will be looking at extending this installation with logging, metrics and dashboarding as well as deploying functions to the framework.

12.5 OpenWhisk installation

In a similar fashion to the installation of OpenFaas we will be using HELM again to install tha chart for OpenWhisk. Where the installation of OpenWhisk differs from OpenFaas is at the step of installation where OpenFaas uses Arkade. For OpenWhisk we will be using so called YAML files as OpenWhisk is not part of the Arkade environment and does not have a similar tool.

YAML is human-readable data serialization language sued to write configuration files. In this file we can apply certain variables to the environment which we would like to configure. For this specific case we will be using the following configuration:

```
   whisk:
 affinity:
   enabled: false
 toleration:
   enabled: false
 invoker:
   options: "-Dwhisk.kubernetes.user-pod-node-affinity.enabled=false"
 ingress:
   apiHostName: openwhisk.elstenit.nl
   apiHostPort: 443
   apiHostProto: https
   type: Standard
   domain: openwhisk.elstenit.nl
   tls:
     enabled: true
     secretenabled: true
     createsecret: true
     secretname: openwhisk-ingress-tls-secret
     secrettype: kubernetes.io/tls
     crt: GENERATED KEY
     key: GENERATED KEY
   annotations:
     kubernetes.io/ingress.class: nginx
     kubernetes.io/tls-acme: true
     nginx.ingress.kubernetes.io/proxy-body-size: 0
```

There are a couple of line in this file which need some explanation such as the term ”affinity”, ”toleration” and the ”invoker”line. As we want an initial version for this prototype with only one worker node. This saves us the configuration of worker nodes and control plane nodes for our OpenWhisk environment. In the future we can still change these parameters as our environment our prototype grows. The other lines are mostly concerned with configuring the ingress for accessing our environment. This is done through the elstenit.nl domain as we already have access to this domain but it would work with every other name too.

In order to create a value for the ”crt” and ”key” we van use a command to create hem in base64:

```
openssl req -newkey rsa:2048 -nodes -keyout tls.key -x509 -days 365 -out tls.crt
cat tls.key | base64
cat tls.crt | base64
```

12.6 Logging, Metrics and Dashboarding

In order to make sure that the platform is performing as expected and delivering a qualitative integration it is important to monitor and log the performance of the services, cluster and the functions. In the current architecture this is done through the open-source logging software Prometheus [45]. As Prometheus is compatible with Kubernetes as well as OpenFaas and OpenWhisk this is considered as a good candidate for logging metrics.

Now that we have a tool for logging our metrics we want an additional piece of software in order to transform these metrics into useful insights. This is done by creating a performance dashboard through the open-source software Grafana [46]. In the following sections we will have a closer look at the installation and setup of these frameworks.

12.6.1 Logging and metrics OpenFaas

With the basic installation of OpenFaas as described in section 12.4 comes Prometheus pre-installed and configured to collect logs. This saves us the hassle of manually configuring it. Therefore we will dive straight into the installation of Grafana dashboarding. In order to install Grafana we will use the kubectl run command to deploy a container image to a Kubernetes pod on our cluster:

```
kubectl -n openfaas run \
--image=stefanprodan/faas-grafana:4.6.3 \
--port=3000 \
grafana
```

We are using a pre-made docker image which works with and is configured to OpenFaaS. In this specific command we are specifying that the service should be available at port 3000 but this can be changed as well. To access this container with the Grafana service we need to expose it to the internet through a nodeport or loadbalancer. For ease of use and to save us the hassle of port-forwarding we will be using the loadbalancer:

```
kubectl -n openfaas expose pod grafana \
--type=LoadBalancer \
--name=grafana
```

In order to test the Grafana dashboard we need to find the IP-address it is accessible on:

```
kubectl -n openfaas get svc grafana
```

When accessing this IP with the corresponding Grafana Port we set earlier, it will display the login field. When we login with the standard credentials, we can find our first Grafana dashboard displaying our OpenFaas metrics. This dashboard can be seen in figure 12.

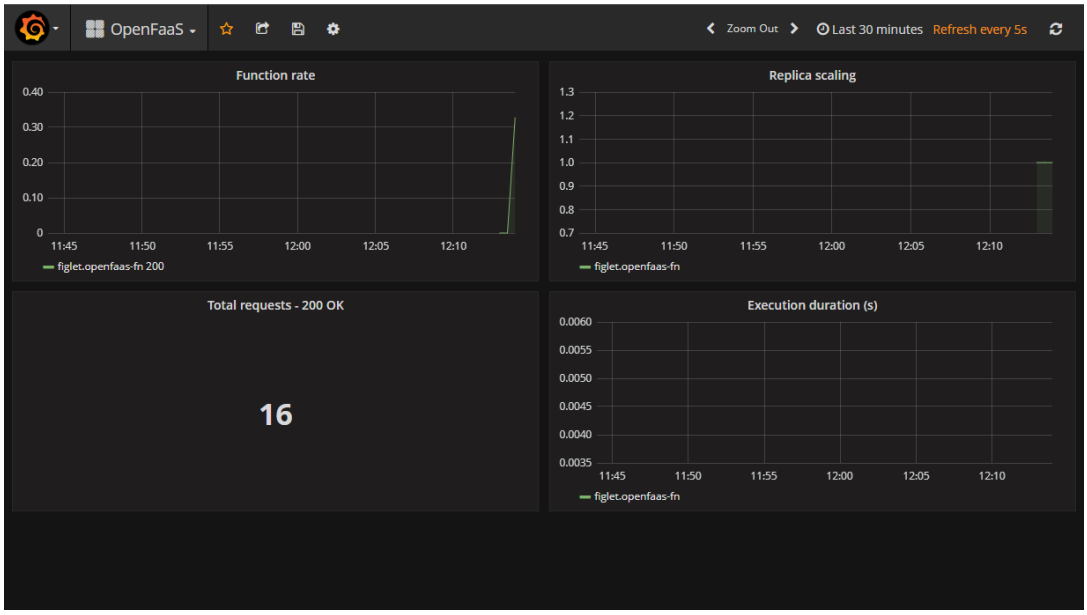


Figure 12: The OpenFaas Grafana dashboard.

12.7 Adapting the Java Application

In order to run Java on the serverless environments, certain changes need to be made to the code. OpenFaas and OpenWhisk both provide Java Templates that can be used to quickly develop an serverless compatible application. In order to check whether or not the iPaas components are able to be ran on the serverless frameworks we will be using a basic template that has some specific iPaaS dependencies. The template is provided by the CTO of eMagiz and specifically for this project has been stripped of unnecessary actions. The following code will be ran on the two test environments:

```
package com.emagiz.boot.application;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ImportResource;

@SpringBootApplication
@ImportResource("classpath:emagiz-flows/*.xml")
public class eMagiz {

	public static void main(final String[] args) {
		SpringApplication.run(eMagiz.class, args);
	}
}
```

The interesting part in this Java excerpt is that it uses an external Extensible Markup Language (XML) file to add content to the program. By doing this, content of the code can change dynamically based on the design of an integration by the end-user in the portal. In this example we will be using the following XML file content that has bee built as a Hello World kind of application:

```
   <?xml version="1.0" encoding="UTF-8"?>
<beans xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
    ↪ springframework.org/schema/beans/spring-beans.xsd
                    http://www.springframework.org/schema/integration http://www
                        ↪ .springframework.org/schema/integration/spring-
                        ↪ integration.xsd
                    http://www.springframework.org/schema/integration/http http
                        ↪ ://www.springframework.org/schema/integration/http/
                        ↪ spring-integration-http.xsd"
            xmlns="http://www.springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:si="http://www.springframework.org/schema/integration"
            xmlns:http="http://www.springframework.org/schema/integration/http">

      <http:inbound-gateway id="backend.receive.http" request-channel="backend.
          ↪ channel.request-json" reply-channel="backend.channel.reply-json"
                    convert-exceptions="false" extract-reply-payload="true"
                        ↪ mapped-request-headers="(none)" mapped-response-
                        ↪ headers="(none)"
                    payload-expression="'{}'" supported-methods="GET" path="/
                        ↪ hello">
            <http:header name="contentType" expression="'application/json'"/>
            <http:header name="param_name" expression="#requestParams.getFirst('
                ↪ name')"/>
```

```
        </http:inbound-gateway>

        <si:channel id="backend.channel.request-json" fixed-subscriber="false"/>

        <si:json-to-object-transformer id="backend.transform.request" input-channel
            ↪ ="backend.channel.request-json" output-channel="backend.channel.
            ↪ request-object"/>

        <si:channel id="backend.channel.request-object" fixed-subscriber="false"/>

        <si:enricher id="backend.transform.hello-world" input-channel="backend.
            ↪ channel.request-object" output-channel="backend.channel.reply-object
            ↪ ">
                <si:property name="greeting" expression="'Hello, ' + (headers['
                    ↪ param_name'] ?: 'world') + '!'"/>
        </si:enricher>

        <si:channel id="backend.channel.reply-object" fixed-subscriber="false"/>

        <si:object-to-json-transformer id="backend.transform.reply" input-channel="
            ↪ backend.channel.reply-object" output-channel="backend.channel.reply-
            ↪ json"/>

        <si:channel id="backend.channel.reply-json" fixed-subscriber="false"/>

</beans>
```

Additionally there will be an infra XML file to handle the authentication of users so not everyone can use the application:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
    ↪ springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/security http://www.
                            ↪ springframework.org/schema/security/spring-security.
                            ↪ xsd"
                xmlns="http://www.springframework.org/schema/beans"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:sec="http://www.springframework.org/schema/security">

        <sec:http>
                <sec:intercept-url pattern="/**" access="isAuthenticated()"/>
                <sec:csrf disabled="true"/>
                <sec:http-basic/>
        </sec:http>

        <sec:authentication-manager>
                <sec:authentication-provider>
                        <sec:user-service><!-- password for all users is: 1 -->
                                <sec:user name="user1" password="{pbkdf2}
                                    ↪ d9ab16838c7ed10e92a9216d1df621027325c88f6ea3bbb2207cb766d59a245f
                                    ↪ " authorities="ROLE_USER"/>
```

```
                    </sec:user-service>
                </sec:authentication-provider>
        </sec:authentication-manager>

</beans>
```

In this specific prototype we will be using the IDE IntelliJ since that is an IDE which is widely used and the author is familiar with. Depending on which of the two framework we want to deploy to, the procedure is a little different so we will be describing them both briefly here:

12.7.1 Deploying a function to OpenFaaS

In order to deploy a basic Java function to OpenFaaS, certain changes need to be made to the function in order for it to be compatible with OpenFaaS. OpenFaas has templates available for many programming languages including Java. We can pull these in to find out what OpenFaaS specific elements need to be in the application. We will be using the faas-cli tool for this:

```
$ faas-cli template pull
```

Now we can create a project while using these templates:

```
$ faas-cli new --lang java11 emagiz-test --prefix="<your-docker-username-here
    ↪ >"
```

This command will create a new project including necessary files. One of these files will be a YAML file with OpenFaaS configuration parameters. By filling in the prefix parameter in the previous command, the imagename will be prefixed with your Docker username needed for compiling the image. There are other elements in the YAML file as well:

```
version: 1.0
provider:
  name: openfaas
  gateway: <ip-of-openfaas-cluster-here>
functions:
  emagiz-test:
    lang: java11
    handler: ./emagiz-test
    image: elstenit/emagiz-test:latest
    labels:
      com.openfaas.scale.min: "1"
      com.openfaas.scale.max: "5"
      com.openfaas.scale.factor: "20"
```

Most parameters in this file are pretty self-explanatory but the labels at the end are specifically interesting. These will limit the scaling of an OpenFaas Application with the minimal and maximal amount of replicas for a certain function and the factor by which it should scale. We will have a closer look at this in the validation section.

For the sake of testing we can deploy this function to our OpenFaas Cluster by the following command:

```
$ faas-cli up -f emagiz-test.yml
```

This command combines the build, push and deploy function into one. Since we are using local docker for building, docker needs to be running and logged in via:

```
$ docker login
```

After successful deployment, the functions van be invoked through the web-interface or the CLI.

12.7.2 Deploying a function to OpenWhisk

In order to deploy a function to the Open-Whisk framework, a similar method is used as with OpenFaas. OpenWhisk has a CLI tool called wsk for interacting with OpenWhisk services. The tool can be downloaded from the GitHub repository and by adding the folder to the system PATH it is accessible from all CLI tools on a computer.

In order to connect to our cluster it is important to tell wsk where it is and authenticate with it. This can be done through the following command where we add the correct IP address of our OpenWhisk installation and a authentication token:

```
$ wsk property set --apihost <master_node_public_ip>:31001
$ wsk property set --auth <TOKEN>
```

In order to test if the connection is set up successfully, we can list the installed packages within the OpenWhisk installation by the following command:

```
$ wsk -i package list /whisk.system
```

When building Java functions for deployment to OpenWhisk, certain requirements need to be met before the Java function van run on OpenWhisk. OpenWhisk is built around the JSON Object as thus needs parameters parsed as a JavaScript Object Notation (JSON) object and it will will return a response in the form of a JSON Object. In order to do this is uses the library: *com.google.gson.JsonObject*.

An example of such a Java file correctly configured for use with OpenWhisk can look like:

```
import com.google.gson.JsonObject;

public class Hello {
    public static JsonObject main(JsonObject args){
        String name;

        try {
            name = args.getAsJsonPrimitive("name").getAsString();
        } catch(Exception e) {
            name = "stranger";
        }

        JsonObject response = new JsonObject();
        response.addProperty("greeting", "Hello " + name + "!");
        return response;
    }
}
```

In order to deploy our own functions to OpenWhisk we need to create so-called actions. We can compile the Java file for that as follows:

```
$ javac Hello.java
$ jar cvf hello.jar Hello.class
```

And finally, the specific OpenWhisk action is created by the following command:

```
$ wsk action create helloJava hello.jar --main Hello
```

Finally, the function can be invoked through wsk by the following command:

```
$ wsk action invoke helloJava --result
```

13 Validation

In this chapter we will be using the prototype to test our remaining requirements. Therefore we will be using two main methods: Quantitative research and qualitative research. First the quantitative research by comparing the metrics and data generated by the prototype to the baseline situation based on non-scaling environments. Requirements that are tested by this method:

- FR2. Based on the workload of the system, the infrastructure must scales up to accommodate the increased workload.
- FR4. For the new architecture, a cost estimation must be made as the new solution should lead to a similar or lower cost than the current architecture.

Additionally, an interview will be conducted with the product manager and developer to gather a qualitative insight into the requirements that are hard to validate using purely quantitative data. This is an interview with both the product manager and the developer in order to encourage discussions in case of a disagreement on a statement. The requirements tested by this method are:

- FR1. The new architecture with serverless should reduce the workload for DevOps teams where infrastructure needs to be configured and maintained.
- NFR5. The new architecture must take security and isolation (of client environments) into consideration.

13.1 Scaling en Costs

In order to test remaining requirements as defined in chapter 9 and which could not be answered by literature research a quantitative research is conducted. The research is done by building the prototype as described in chapter 12 and setting up particular tests on it for auto-scaling (section 13.1.1) and the costs (section 13.1.2). These tests and their results are explained in the following sections:

13.1.1 Autoscaling

First, requirement **FR2. Based on the workload of the system, the infrastructure must scale up to accommodate the increased workload.** is tested. This is done by deploying an example function to the OpenFaaS and OpenWhisk deployments in the Google Cloud and firing a certain amount of API calls to it. When looking at the OpenFaaS installation, we are using the CE (Community Edition) for prototyping. This version only supports one type of autoscaling [47]. This way of auto-scaling reads the usage of a function in RPS (Requests Per Second) from the Prometheus Metrics service in order to know when it needs to fire an alert to the API gateway and start a new instance. Synchronous as well as asynchronous calls to a function count towards this method of auto-scaling. The rule when this alert is fired is configured in the Prometheus config file of OpenFaaS:

```
alert.rules.yml: |
   groups:
     - name: openfaas
       rules:
       - alert: APIHighInvocationRate
         expr: sum(rate(gateway_function_invocation_total{code="200"}[10s])) BY (
             ↪ function_name) > 5
         for: 5s
```

```
        labels:
          service: gateway
          severity: major
        annotations:
          description: High invocation total on "{{ "{{" }}$labels.function_name
              ↪ {{ "}}" }}"
          summary: High invocation total on "{{ "{{" }}$labels.function_name{{
              ↪ "}}" }}"
```

We can see here that if the amount of requests for a certain function exceeds 5 for 10s an alert is fired to the API Gateway. Based on these alerts, we can configure rules how to handle. The amount of instances serving the request should grow towards the maximal amount as configured in the YAMl file of the example application. The scale factor is the amount of instances by which the system should be scaled up. For this test we configured the YAML file for OpenFaas as follows:

```
version: 1.0
provider:
  name: openfaas
  gateway: <ip-of-openfaas-cluster-here>
functions:
  emagiz-test:
    lang: java11
    handler: ./emagiz-test
    image: elstenit/emagiz-test:latest
    labels:
      com.openfaas.scale.min: "1"
      com.openfaas.scale.max: "5"
      com.openfaas.scale.factor: "20"
```

This means that per alert coming from the alertmanager we will be scaling with 1 extra instance to a maximal amount of 5. The pro version of OpenFaas does support scaling to zero (it does not leave any 'warm' replicas to immediately start processing the request) but as the Community Edition does not we do not have to worry about that for now and we will keep a minimum of 1 replica active all the time.

In order to monitor the scaling, we will be using the Grafana dashboard as described in section 12.6 which shows the amount of replicas per function over a certain amount of time. The variable we are interested in for the replicas is generated by the API gateway and called *"gateway_service_count"*.

In order to fire a big amount of requests to the function, we will be using the Postman API platform [48]. By configuring a so-called "runner" we can iterate a request multiple times over a certain period. We configure the runners to send 2000 iterations of a request to our serverless function. We set the delay to 0 to create as many requests as possible in a short timespan. The request per runner are not send in bulk at once but after each other as soon as a response is received back. Another benefit of using this method is that it will calculate the average response time of the function which is of interest for testing our prototype as well. For loading our function we will be running 5 runners with 2000 iterations each sending a total of 10000 requests to one function. After the runners finish they will tell how long they have been running so we can calculate the Requests Per Second (RPS) and see if the function scales accordingly.

The results are as follows, the 5 times 2000 iterations of the Postman Runner are all started within 300ms from each other and take all 5 minutes (4 minute 56 seconds to 4 minute 58 seconds) to fully complete the 2000 iterations of each runner. This means that each runner fired about 2000 requests in 297 seconds resulting in around 6.74 request per second per runner. Multiplying this by 5 results in a total of around 34 request per second fired at the function. This can also be seen

in figure 13 retrieved from our Grafana dashboard connected to the OpenFaaS Prometheus. We can see that the rate scales up and when all 5 of the runners are running constantly the increase flattens down. As the framework is configured the launch 1 additional replica per alert, we expect a steady increase in replicas as soon as the rate increases. This graph can be seen in figure 14 and we find that the framework scales to our maximal amount of replicas of 5. As soon as the workload drops, the replicas are scaled back down to our minimum of 1. As a result we can conclude that the OpenFaas framework successfully passes the requirement "to automatically scale in case of increased workload.

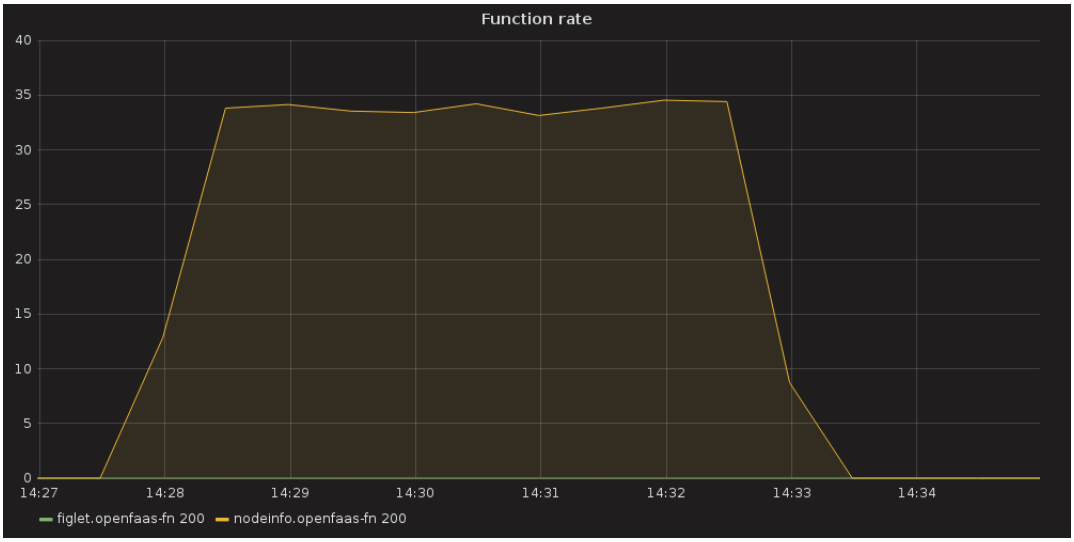


Figure 13: Function Rate when testing OpenFaaS Scaling.

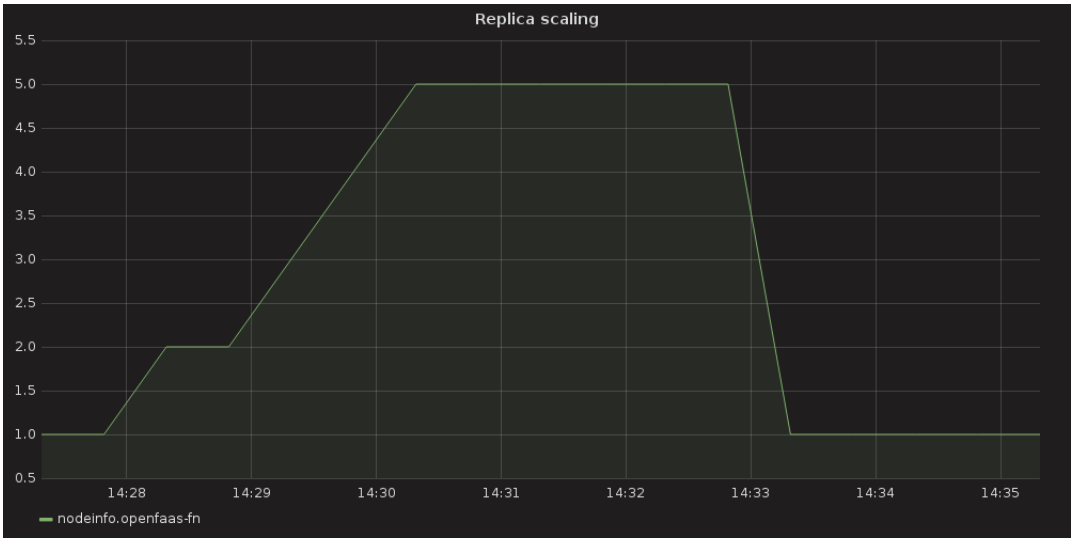


Figure 14: Replica Scaling when testing OpenFaaS Scaling.

On the other hand we have the OpenWhisk installation, this framework scales different from OpenFaaS and we cannot specify a min and max amount of instances for a specific function. OpenWhisk has similar to OpenFaas a controller (OpenFaas calls this the API gateway) which takes requests and puts them in Kafka topics. OpenWhisk than has invokers which subscribe to these topics and listen for new messages. The invokers take the request and deploy it to a worker that actually executes the action. When there is no available worker, the invoker instantiates a new one in order to serve the request. As long as the resources are not maxed out or do not hit any configured limits, new workers can be instantiated by an invoker. In order to make this more clear we can have a look at figure 15.

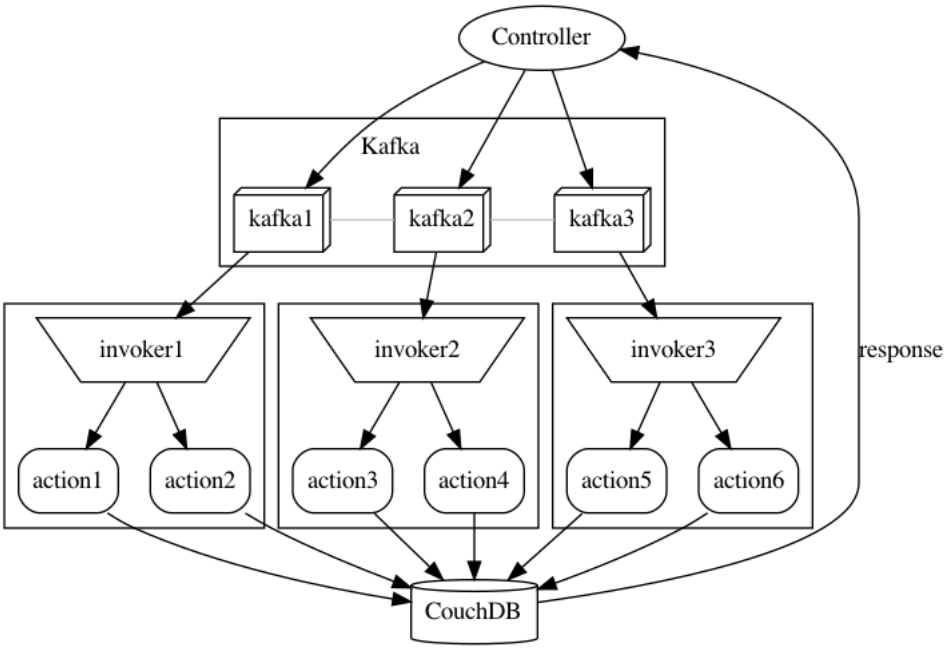


Figure 15: Flow of a request in OpenWhisk

In the image we see a controller which receives every message. Based on which function the message needs to reach, it is put on a certain Kafka topic. This means that one controller can direct messages to multiple different functions and their Kafka Topics.

What happens with OpenWhisk is that as long as the resources of the cluster are not maxed out, invokers will be taking the messages from the Kafka topics and activate actions for them on workers. As a result we cannot speak of a replica count as with OpenFaaS as it is just a number of activations of actions. Therefore, there is a difference in the used metrics to prove scaling.

In order to tune this scaling, there are some parameters which can be changed from the default in order to create a correct way of scaling for a specific installation. In this example we will be deploying a basic API application similar to the one deployed to the OpenFaas environment. In order to scale this application the parameters are tuned as follows:

```
limits:
  actionsInvokesPerminute: 60
  actionsInvokesConcurrent: 30
  triggersFiresPerminute: 60
  actionsSequenceMaxlength: 50
  actions:
```

```
    time:
      min: "100ms"
      max: "5m"
      std: "1m"
    memory:
      min: "128m"
      max: "512m"
      std: "256m"
    concurrency:
      min: 1
      max: 1
      std: 1
    log:
      min: "0m"
      max: "10m"
      std: "10m"
```

These parameters are mainly there to prevent an overloading of the cluster. Lets briefly look over the important ones. *actionsInvokesPerminute* limits the maximum amount of invocations per minute, *actionsInvokesConcurrent* limits the maximum concurrent invocations, *actionsSequence-MaxLength* indicates the maximum sequence length of an Action., *triggersFiresPerminute* limits the maximum triggers invoked per minute.

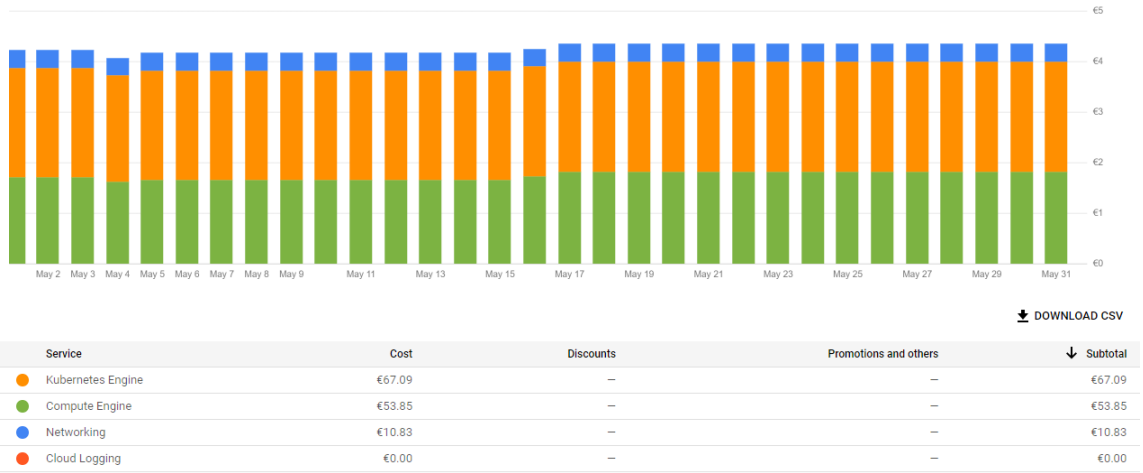
Since there is no specific parameter for when to scale up (like in OpenFaaS, the scalefactor), we will not change these for testing this requirement.

13.1.2 Costs

To wrap up the quantitative research for the validation we will have a look at the cost estimation of running the serverless to answer requirement **FR4. For the new architecture, a cost estimation must be made as the new solution should lead to a similar or lower cost than the current architecture.** It is very hard to make an exact guess on the costs of running a serverless iPaaS. Therefore we will be making an educated guess based on some assumptions. We will be using only on-demand pricing since it is difficult to estimate the dynamic workload and therefore fit a contract over on-demand pricing. For pricing a serverless framework running in a Kubernetes Cluster we have two cost components. One static component and one dynamic component. The static costs are costs that are independent of the load and data processed. The dynamic cost component is dependent of the load and data processed by the system and therefore grows with the usage of the framework.

We will first start by looking at the static cost component: Google Kubernetes Engine applies a $ 0.10 per cluster per hour management fee resulting in around € 70.00 per month. Then there are some networking fees that include services such as static IPs, loadbalancers and DNS.

The largest part of the costs is the Computing and Storage costs. These can be considered dynamic since they change based on the load and traffic processed by the cluster. After running our prototypes for 1 month in the Google Cloud we found the monthly costs of both frameworks to be as in figures 16 and 17.



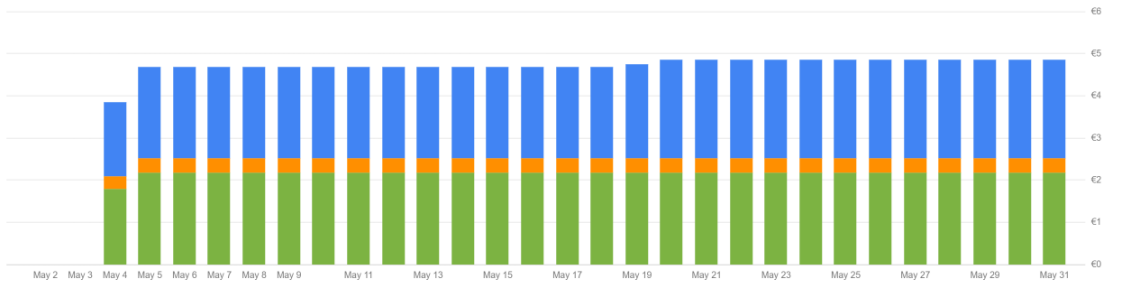
Service	Cost	Discounts	Promotions and others	Subtotal
Kubernetes Engine	€67.09	–	–	€67.09
Compute Engine	€53.85	–	–	€53.85
Networking	€10.83	–	–	€10.83
Cloud Logging	€0.00	–	–	€0.00

Figure 16: Costs of OpenWhisk on GKE

For OpenWhisk this totals to € 131.77 for one month. For OpenFaas the cluster had a problem at the beginning of May where due to a configuration mistake the OpenFaaS installation did not run anymore and needed a new installation. Therefore, we will be looking at the average for the rest of the month and use this to calculate the monthly costs under normal use. We find a total of € 152,02 for the OpenFaaS cluster.

In order to get an idea of the costs when using the framework in production for an iPaaS we estimate the amount of currently used machines fitting on the serverless framework. Due to confidentiality, we cannot publish any exact current numbers on costs of the eMagiz iPaaS infrastructure. However, in previous research within eMagiz [12] some numbers were made publicly available. In this research it became clear that current VM's have very varying utilisation based on the integrations and it's use.

Since the environments investigated in the research on Kubernetes [12] are still valid, we will be using them here to make an educated guess on the costs. The first environments has a CPU



DOWNLOAD CSV

Service	Cost	Discounts	Promotions and others	Subtotal
Compute Engine	€62.42	–	–	€62.42
Kubernetes Engine	€60.26	–	–	€60.26
Networking	€9.73	–	–	€9.73
Cloud Logging	€0.00	–	–	€0.00

Figure 17: Costs of OpenFaaS on GKE

utilisation of around 6% on average during the day with peak times hitting just over 70% CPU usage from 18:00 until 0:30. This environment consisted out of 32 CPU cores and 128GiB RAM.

The second environment had during weekdays an CPU usage between 8 and 18%. In weekends this averaged at 4%. Some peaks around 55% were observed but these were very short. RAM usage averaged around 50 %. This environment consisted out of 16 CPU cores and 64GiB RAM.

The last environment looked at in the Kubernetes research [12] had an average CPU usage of around 8 and 9%. The RAM usage was around 80 and 71% with no spikes in RAM or CPU. It consisted out of a 16 CPU cores and 64GiB RAM.

Overall we can state that the 3 environments are having more CPU and RAM available than they on average need. When we multiply the usage to the amount of CPU and RAM available in the set we know the amount of actually used CPU and RAM of the integration environments.

- For environment 1: 21 hours a day of 6 % CPU usage (21 * 0.06 * 32 = 40.32), 3 hours a day of 70 % CPU usage (3 * 0.7 * 32 = 67.2). This means that an average of 4.48 core/hour is needed to run this environment.
- For environment 2: 5 days a week 24 hours a day of 13 % CPU usage (5 * 24 * 0.13 * 16 = 249.6), 2 days a week 24 hours a day of 4 % CPU usage (2 * 24 * 0.04 * 16 = 30.72). This means that an average of 1.67 core/hour is needed to run this environment every day.
- For environment 3: 24 hours a day of 8.5 % CPU usage (0.085 * 16 = 1.36) This means that an average of 1.36 core/hour is needed to run this environment.

This means that when we combine the 3 environments we would need 9 cores to run them. Of course, when all environments peak during the same time, this would not be enough but when there are many environments combined in the serverless framework the changes of peaking at the same time are pretty low while having enough room for scaling up a certain environment during peak times. When looking at the GKE machine types, an e2-highcpu-16 would be suitable for this with 16 cores CPU when purely looking at CPU. When we include RAM, e2-highmem-16 might be a better choice as it still has 16 cores of CPU power but it has 128 GiB of RAM instead of 16. Monthly costs of this instance would be $527.8776 instead of the $24.45719 of the e2-medium instance in the prototype. Total dynamic costs of the old situation would be around $1550 [12]. Dynamic costs of the new situation can be guessed around $560 when no peaks occur at the same

time. This educated guess provides a benefit of the new architecture with FaaS over the baseline Architecture when taken into consideration that is is based on comparable usage of CPU and RAM.

It must be taken into consideration that these estimations are very rough as they are based on numbers and statistics of currently developed integrations. In order to have these integrations work on a FaaS basis, an overhaul of the code and architecture needs to be done and different CPU and RAM usages may occur.

As our prototype was made in the Google Cloud and we can take the billing reports from there, previous numbers are based on Google Cloud Costs. Since eMagiz is running their infrastructure in the AWS or Amazon Web Services Cloud we will be comparing the pricing of these services in order to validate whether they can be assumed comparable. In order to do so, a table 7 has been created where we will list the costs of the used Google Kubernetes Engine (GKE) components and their AWS counterparts and their costs. Overall, AWS is slightly more expensive than the Google Cloud when looking at the costs of the compute instances. Costs taken for this table are from June 2023 and zones are chosen in America (us-central1-a for GKE and use-east-1 for AWS).

	GKE:	**AWS:**
Kubernetes Cluster Management Fee	$0.10/hour	$0.10/hour
Compute Engine (e2-medium, t2.medium)	$0.033503/hour	$0.0464/hour

Table 7: Comparison of costs between Google Cloud and Amazon Web Services

Of course, cloud infrastructure costs are not the only costs that need to be taken into consideration when looking at moving towards a serverless based infrastructure. The costs for actually adapting the code and functions to work on a serverless framework as well as the continuing maintenance costs for keeping the cluster and framework updated need to be looked at. In order to get a better understanding of these costs, a section about costs is included in the qualitative section at 13.2.3.

13.2 Workload and Security

As it hard to quantitatively measure every requirement, we conducted a qualitative research as well. This research focused on the workload for the DevOps teams as well as the security and isolation possibilities within the new solution. In order to gain an insight into whether or not these requirements are fulfilled, a semi-structured interview is conducted with the product manager and a developer of eMagiz iPaas solution. Both persons are invited to the same interview in order to encourage discussions in case of a disagreement amongst them.

During this interview, the prototype of both serverless frameworks is demonstrated including the process of deploying a new function to the frameworks. Since both persons are already familiar with setting up and managing a Kubernetes cluster this is not explicitly demonstrated.

13.2.1 Deployments and Maintenance

After the demonstration, the interviewees are asked on their opinion about the process of deploying a function to the serverless frameworks. Compared to the current architecture based on virtual machines, this process is a big improvement in terms of time and complexity. Where previously, a new machine needed to be configured and spinned up they now deploy just the function to the already existing serverless framework directly from their IDE. OpenFaas is based around Docker images so even the current Constant Integration / Constant Delivery (CI/CD) can easily be implemented and not a lot of changes need to be made from the current workflow of deploying images. For OpenWhisk, this is a bit different which is less in favour compared to OpenFaaS. As OpenWhisk uses Java compiled actions over Docker Images it has no out-of-the-box support for an image storage and therefore a CI/CD based on that. It has possibilities to work straight from GitHub but this would mean that the current workflow needs a turnaround and therefore is the less favourable option of the two and would not lead to a big decrease in workload compared to the current situation according to the interviewees.

Secondly, is it asked how maintenance intensive they predict the proposed solution to be in terms to managing the current VPC based machines. Currently, machines are updated and sized manually per environment. In case of a multi-tenant solution only the Kubernetes Cluster including the serverless framework needs to be updates. Scaling is done automatically for both frameworks which saves the hassle of redeploying customers on new machines when their integrations grows over time. Overall, the interviewees think it will lead to less maintenance time but it is hard to add a figure in terms of time or hours to it. A side-note to add to this is that by moving to such an architecture, more is happening in a kind of blackbox way and therefore it might be harder to find and solve problems in case any arise. Therefore, the metrics and dashboarding functionality is important and both frameworks do support these tools.

13.2.2 Security & Isolation

In order to validate whether or not the serverless frameworks are adequately equipped to provide security and isolation measures we looked at the available technologies that are often used to provide this. First, communication to and from the serverless framework, both frameworks are compatible with a ingress controller. This controller provides the capabilities to access the services from outside the cluster. This controller provides services such as Transport Layer Security (TLS) termination and loadbalancing. By providing this Secure Sockets layer (SSL) functionality we can make sure that only authorized senders are actually sending messages to the framework by using certificates. Due to the limited time, we only looked at manually created certificates but both frameworks should be compatible with certificate managers as cert-manager for Kubernetes [49]. Additionally, it is mentioned that the cluster itself needs to be hardened but this can be done through Kubernetes Security best practices. OpenFaas has explicit compatibility with role-based authentication control (RBAC) which is a big plus since most developers are already familiar with this according to the

interviewees. Overall, OpenFaas has better documentation on the possibilities of security and hardening. OpenWhisk might have some of these functions as well but it is less documented for sure making it harder to implement.

In the case of a multi-tenant environment, isolation of data from different tenant is important. First, we look at OpenFaas which runs it's functions in individual pods to which rules can be assigned that there is no communication across multiple simultaneously running functions. A similar process happens for OpenWhisk which spins up a docker container for every function where isolation can be enforced too. In case of isolation requirements, both frameworks do pass according to the interviewees.

13.2.3 Costs

In order to say something useful about the costs of adapting the current applications and code to a serverless approved variant we ask the interviewees to make a guess towards the amount of time that is needed to move from the baseline to this solution. This can be considered a one-time cost as it is not an ongoing process and in case of new functions being added to the platform, there is a benefit as identified in the section on workload above.

A change in costs also occurs when looking at the maintenance of this solution. As already mentioned in the part about workload, both interviewees foresee less maintenance compared to the current architecture but it is hard to give an exact number in terms of hours or costs. In future research it would be good to create a full-scale environment in order to gain more insight in the amount of maintenance required by the new architecture implementing serverless technology.

13.3 Overview of tested requirements

In order to conclude our validation of the solution we will have a final look at the requirements as set in chapter 9. In table 8 we explain in which chapter which requirement was tested during this research and whether or not this requirement has been fulfilled.

Requirement:	**Where has it been tested?**	**Outcome:**
FR1. The new architecture with serverless ***should*** reduce the workload for DevOps teams where infrastructure needs to be configured and maintained.	In chapter 13 Validation	Partly fulfilled
FR2. Based on the workload of the system, the infrastructure ***must*** scale up to accommodate the increased workload.	In chapter 10 Serverless Solutions and chapter 13 Validation	Fulfilled
FR3. When there is no demand for a certain function to be ran, the machine(s) ***should*** scale to zero.	In chapter 10 Serverless Solutions and chapter 13 Validation	Not fulfilled (Not possible in free editions used in prototype.)
FR4. For the new architecture, a cost estimation ***must*** be made as the new solution ***should*** lead to a similar or lower cost than the current architecture.	In chapter 13 Validation	Partly fulfilled
NFR1. The serverless solution ***should*** have a license that allows an iPaaS provider to use it in a commercial product.	In chapter 10 Serverless Solutions	Fulfilled
NFR2. The serverless solution ***must*** be well-maintained / actively developed.	In chapter 10 Serverless Solutions	Fulfilled
NFR3. The serverless solution ***should*** be used by some big companies in order to prove a certain magnitude within the field.	In chapter 10 Serverless Solutions	Fulfilled
NFR4. The serverless solution ***should*** natively support the programming language (Java) as often used by standard iPaaS solutions.	In chapter 10 Serverless Solutions	Fulfilled
NFR5. The new architecture ***must*** take security and isolation (of client environments) into consideration.	In chapter 13 Validation	Fulfilled
NFR6. The chosen solution ***should*** be open-source to provide CSP-agnostic capabilities.	In chapter 10 Serverless Solutions	Fulfilled
NFR7. The chosen solution ***must*** be compatible with Kubernetes container orchestration framework.	In chapter 10 Serverless Solutions and chapter 13 Validation	Fulfilled
NFR8. The chosen solution ***should*** be compatible with iPaaS typical triggers.	In chapter 10 Serverless Solutions and chapter 13 Validation	Fulfilled
NFR9. The chosen solution ***could*** be having a business support in order to work through high-impact problems.	In chapter 10 Serverless Solutions	Partly fulfilled (OpenFaaS does, OpenWhisk does not.)

Table 8: Overview of which chapter tests which requirement anf if it was fulfilled or not.

14 Conclusion & Discussion

In this chapter we will be briefly looking back at the answering of the different sub questions. Than we will draw a conclusion from these answers that will contribute in answering the main research question: How can serverless be implemented for an iPaaS solution in order to improve scalability and reduce infrastructure costs? Additionally, we will elaborate on the limitations and contributions of this research. The chapter is concluded with recommendations for the company and some ideas for future research on the topic of serverless within an iPaaS.

14.1 Revisiting the Research Questions

In this section we will be answering the research questions defined in chapter 7.4. In this research we had a closer look at the suitability of serverless for use in iPaaS environments and how an iPaaS provider could implement serverless. In order to answer this question, it was important to look at certain requirements such as portability/vendor lock-in, scaling, security and costs. We will shortly look at the sub questions of this research before continuing to answer the main research questions.

The first subquestion was **SQ1. What cloud models are currently in use and what defines them? (SaaS, PaaS (iPaaS) and IaaS)?** which was answered through a literature research in chapter 8. Overall, there are three main cloud models: SaaS, PaaS and IaaS. They mainly differ in the amount of management that is done by the provider instead of by the enduser. With SaaS, only the data in the application is managed by the user. The application, runtime, OS, Servers, Networking etc. are all managed by the CSP. With PaaS, the application and data is configured and managed by the user, all other components are managed by the Cloud Service Provider. Lastly, the IaaS is a form where only the infrastructure including virtualization, servers, storage and networking is provided by the Cloud Service Provider. Everything else is managed by the user. Popular CSP's such as AWS, Google Cloud and Microsoft Azure are examples of an IaaS. The company where this research is conducted, eMagiz, is an example of a special kind of PaaS as it delivers a platform where customers can build applications specifically targeted toward integrations. This type of PaaS is called an integration-Platform-as-a-Service (iPaaS).

The second subquestion was **SQ2. What are the state-of-the-art serverless technologies/frameworks that currently exist and in what way do they differ from each other?**. In order to answer this question, serverless technology is divided into two main types. Namely Backend-as-a-Service (BaaS) which is mainly concerned with additional services of CSP's that do not need or need less management from the user. Secondly, the term Function-as-a-Service (FaaS) is introduced which is the type we are mostly interested in in this research. FaaS makes it possible to run on-demand functions without the hassle of managing the underlying infrastructure and extra services such as scaling. A research was carried out to find all state-of-the-art serverless FaaS technologies. As a result we constructed a list with commercial and open-source serverless technologies. The commercial technologies include AWS Lambda, Google Cloud Functions and Microsoft Azure Functions. Besides these commercial variants, we looked at the open-source versions as well to prevent vendor lock-in. This resulted in the following frameworks: Apache OpenWhisk, Fission, Knative, Kubeless, Nuclio and OpenFaaS. Besides the difference of open-source vs commercial the technologies also differ on available documentation, user base, license and much more. This is described in detail in chapter 10. This list is compared to the requirements as set in the following subquestion to help decide on the correct technology to use in the proposed solution.

The third subquestion was not specifically literature related but was answered by an expert interview session. The question was **SQ3. Which functional and non-functional requirements are in place for a serverless architecture to ensure similar business functionality, a decrease in costs and DevOps but increase scalability for an iPaaS solution?**. This question

serves as an input for the requirements to test the available technologies against. Since there are many possible solutions it is important to find out requirements that are of importance for when such a technology is used within an iPaaS solution. This is done by interviewing multiple persons individually through a semi-structured interview. As a result, a list of requirements was constructed and prioritized via the MoSCoW method. This can be found in detail in chapter 9. The most important "must requirements" resulting from the interviews are concerning the requirement that the environment needs to auto-scale in case of an increased workload, the framework or technology must be well-maintained and/or actively developed, the technology must be compatible with Kubernetes container orchestration framework and the technology must provide measures to provide security and isolation. In the following subquestion we will be answering how the different technologies and frameworks actually fulfil these selected requirements.

The fourth subquestion was **SQ4. How do the state-of-the-art serverless technologies/frameworks fulfill the elected requirements?** which was answered through literature and prototyping. Following sub question 3, a list of functional and non-functional requirements was constructed. Some of these requirements could already be answered through literature research such as the ability to work with Kubernetes or the requirement of having a license that allowed for use in a commercial product. Other requirements needed a small-scale prototype of the serverless solution to actually test the requirement. This prototype and the validation done with it can be found in chapter 12 and 13. Overall it can be stated that the OpenFaas framework does fulfill most of the requirements and OpenWhisk follows closely. The main difference between these two frameworks is in terms of documentation and ease of use when looking at security and isolation.

The fifth sub question concerns the architecture an iPaaS solution. Specifically the infrastructure that enables the functioning of an iPaaS platform. That resulted in **SQ5. What is the prevailing architecture of an iPaaS infrastructure?** The architecture model was created on the basis of a research at eMagiz. By interviewing and investigating their platform through internal documentation, the underlying technologies for providing the services of an iPaaS became clear which served as input for the baseline architecture. The architecture was created with the ArchiMate 3.1 reference in mind. It was found that the baseline architecture revolves around 3 main business processes, namely: setting up an (new) integration, managing an integration and processing customer's data through an integration. In the baseline architecture these processes are handled in the application layer by a VPC and Infra Cloud as well as additional services provided by the portal service of the iPaaS.

The sixth sub question is **SQ6. How can the state-of-the-art serverless technologies/frameworks be implemented into the current typical iPaaS infrastructure?**. In this question the prevailing architecture is adapted in order to implement serverless technologies. A closer look is taken at what components can be re-used from the prevailing architecture and which changes need to be made in order to facilitate the serverless framework. This can be found in chapter 11. In this chapter it was found that the main changes in architecture are actually occurring in the business process where data of customers is actually processed. The portal for building and designing an implementations is unchanged as well as the tools used to manage the integration. In order to create the new architecture, the integration which is currently ran in a customer specific VPC is replaced by a multi-tenant Kubernetes Cluster running an open-source FaaS framework to process functions on request and with auto scaling capabilities.

The last sub question is **SQ7. What measurable improvements on scalability and costs of infrastructure does the new architecture for implementing serverless technologies into iPaaS bring?**. In order to answer this question, the prototype was validated by doing various tests on it. These test varied from stress-testing the environment in order to prove and test the auto-scaling functionality as well as making a cost estimation after running the environment for a while. The results that were obtained showed that the autoscaling functionality worked as predicted and that by auto-scaling the environment the workload could be handled without wasting resources as in the baseline architecture. Additionally an improvement in costs was visible as there were less

costs for wasted resources.

Overall, the answers of these sub-questions contributed to the answering of the main research question: **MQ1. How can serverless be implemented for an iPaaS solution in order to improve scalability and reduce infrastructure costs?**. To describe the answers shortly, the opensource frameworks OpenWhisk and OpenFaas can be used in a Kubernetes Cluster to run an iPaaS environment while still maintaining portability. This adaption of a serverless framework leads to less wasted resources as the functions of multiple integrations can all be ran on the same cluster instead of spinning up a VPC for every customer. This change therefore leads to a improved scalability and reduction of infrastructure costs while maintaining other requirements such as preventing vendor lock-in and have compatibility with iPaaS typical triggers. The suggested architecture works with tooling for logging and metrics that eMagiz is already familiar with, this helps in a transition as the developers do not need to learn new tools. The biggest impact of this change is that every integration needs to consist of a series of functions instead of consisting out of a full applications. This would require a certain amount of reprogramming the current Java applications.

14.2 Limitations

During this research, certain choices were made which can act as a limitation. These limitations should be taken into consideration when interpreting the findings and implications of this study.

- Temporal Constraints: Frameworks and technologies used and evaluated in this research are active and available in the period from the October 2022 until March 2023. Choices in this research are made based on the maturity and status of this time period. However, due to the rapidly evolving nature of serverless computing it might be possible that new technologies and frameworks might be more applicable or advantageous in the near future for a similar use-case. Therefore, it is important to note that the findings and conclusions of this study are constrained by the technological landscape as it existed during the research period and that subsequent advancements may provide alternative solutions for achieving similar objectives.
- In this research, only open-source FaaS frameworks were considered for the prototype and the following validation phase because of the "must" requirements where the solution must be cloud-agnostic. As most iPaaS platforms do actually focus on one specific CSP, this limitation could be mitigated by having this requirement as a could instead of a must. In that case, the commercial variants of FaaS by parties such as AWS, Google and MS Azure could be validated by a prototype as well and this may yield different results. Especially when looking at cost reduction and maintenance as this takes out the maintenance factor which open-source solutions bring.
- This research focused on the implementation of FaaS within an iPaaS where the starting point is the as-is or baseline architecture. The results of the research are therefore influenced by earlier design choices of the iPaaS used in this case study instead of starting with a complete blank paper.

14.3 Contributions

In this section, the contributions made by this research to both research and practice are discussed. The findings and outcomes of this study have provided valuable insights and advancements in the following areas:

14.3.1 Contribution to research

This research focused on giving guidelines for implementing serverless technology, specifically FaaS, into an iPaaS platform. Currently, the scientific field of serverless is limited with only 1384 results in Scopus at the end of 2022. As in this research multiple forms of serverless technologies are compared

with each other it results in a state-of-the art overview of where open-source serverless frameworks are at nowadays and which are still looking to be promising participant(s) in the open-source serverless market.

Overall, this research contributes to research by providing:

- A state-of-the-art comparison of available serverless FaaS technologies including commercial and open-source solutions. Certain factors such as if the technology is actively developed, has a license to use it commercially and if it has a large user-base with big companies amongst their users to prove a certain magnitude are compared and contrasted in order to form an overview. This information can be useful for other research to help deciding on the use of a certain technology for testing or investigating.
- A baseline architecture of a prevailing architecture infrastructure of an iPaaS platform is constructed. This architecture can be used in other researches about iPaaS technologies to serve as in input or example of an possible architecture of an iPaaS platform. Additionally, it can be used to compare the architecture to other PaaS Providers.
- The research provides guidelines for adopting serverless technology into an iPaaS platform including tool selection and installation description. Aditionally, these guidelines hold for other PaaS products with a similar baseline architecture.
- In this research we explored the potential use of FaaS within the context of an iPaaS infrastructure. As a result, during this research multiple follow-up questions arose which are further explained in section 14.5 on future research.

14.3.2 Contribution to practice

The contributions to practice have similarities with the contributions to research. For example when looking at possible candidates for a serverless project, this research can be used as a contribution towards the final selection when similar requirements of this project are in place.

Overall, this research contributes to practice by providing:

- Guidelines for implementing serverless FaaS technology into an iPaaS platform including a reasoned choice for the used framework based on the information retrieved in the time of this research (2022-2023). Additionally, tools for additional services such as metrics and logging are touched upon including its configuration.
- A architecture of the baseline and an architecture with serverless technology implemented which can be used by the company to better understand their infrastructure and the connection between the different technologies and tools used.

14.4 Recommendations

During this research it was found that specifically open-source frameworks are relatively new and still on the rise and therefore lack extensive support through a community of users. As a result, we would not yet recommend to go full open-source serverless even tough the benefits of serverless technology in the field of iPaaS are described and present in this research. It should be noted that as commercial variants are left out of the validation as they void the requirement on vendor lock-in, these are more widely adopted and do possess the support of a larger community base. Overall, it would be wise to keep following future advancements in the market of serverless FaaS technologies and see how the support and wide adaptation grow the maturity of such technologies. By doing this, it is likely to start changing the iPaaS platform's architecture in a way to support these new technologies as soon as possible and start to benefit from the improvements these technologies bring. Additionally, we would recommend to lower the vendor lock-in requirement prioritization wise. As

currently most parts of the platform are already in running on AWS services it would be wise to look into the possibilities that this CSP brings when looking at FaaS services. Eventually, eMagiz could look into running the FaaS services simultaneously on another commercial variant such as Google of Microsoft to cater to the client not wanting to connect to AWS.

lastly, we would like to encourage eMagiz to start looking into different pricing models. Currently, the product is licensed on a per integration basis without taking into consideration the actual amount of data that is processed by the integration. By moving towards an automatically scaling environment, doors are opened toward a pay-per-use pricing model where the customers actually pay for the amount of data that is processed by the integration.

14.5 Future Research

During this research, we gained valuable insights into the applicability of serverless FaaS technologies in the field of iPaaS. However, due to resource and time constraints it is not possible to investigate every aspect of the suggested solution and thus possible directions for future research were identified. This section outlines potential future research directions to enhance the understanding, adoption, and evolution of serverless FaaS within an iPaaS platform.

Because of time constraints, a prototype and validation where made with test functions and test data specifically designed for this research by the iPaaS provider in order to test whether the technology would work with functions used within the platform. To get a better insight and possibly discover more information on costs and compatibility it would be wise to perform a large scale test with real data processed by the platform. Possibly, running in a mirror configuration to see whether the new architecture would be a suiting fit for the iPaas Provider. In order to do this, current data 'pipelines' within the platform need to be adopted for use with a system based around functions instead of around complete applications. After successful adaption to functions, one could start processing production data and compare the performance to the old situation.

During this investigation, the input and testing were done with one iPaaS provider due to time constraints. In order to validate whether or not the same results and conclusion are applicable for other iPaaS providers as well, testing needs to be done with these providers. Even tough, most providers are offering the same basics, the extra tooling can differ from provider to provider and can influence the choice of architecture and technology.

This research only focused on the application of serverless FaaS technology within iPaaS environments. It might be possible that this technology is applicable to other cloud models as well such as SaaS. This might be especially interesting for environments that have to deal with big peaks in the workload such as bulk data processing products. In order to prove this generalisation, additional research needs to be done.

As mentioned in the limitations section, only open-source serverless technologies were taken into consideration for the prototype and validation part of the investigation due to a prevention of vendor lock-in through commercial variants. It is possible that a commercial variant of FaaS might actually yield better results in terms of scaling and cost reduction. Therefore, it would be interesting to test commercial variants against the open-source variants tested in this research.

Another possible direction for future research might be around looking at the possibilities of FaaS technologies within an iPaaS when starting with an empty canvas instead of starting with a baseline architecture as currently implemented by the company. As all design choices can be optimized around the application of FaaS instead of already made design choices from the baseline architecture, this may yield different results compared to the result of this research.

Another aspect which was encountered during this investigation was that a lot of the investigated serverless frameworks had not been actively developed/maintained over the last couple of years. Even tough, the serverless market is still on the rise when looking at the amount of scientific articles recently written about it and the rising application in pratice. It might be interesting to research on success factors of open-source serverless frameworks and why others are not successful anymore.

References

[1] A. Rahman, A. Luis, L. Ferreira, and M. Albano, "Message oriented middleware with qos support for smart grids," 07 2022.

[2] A. Modi, "Iaas vs. paas vs. saas," Jun 2021. [Online]. Available: https://dev.to/cloudtech/iaas-vs-paas-vs-saas-41d2

[3] "Spring and serverless." [Online]. Available: https://spring.io/serverless

[4] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, 2022.

[5] "Openwhisk documentation." [Online]. Available: https://openwhisk.apache.org/documentation.html#documentation

[6] N. Ebert, K. Weber, and S. Koruna, "Integration platform as a service," *Business and Information Systems Engineering*, vol. 59, no. 5, pp. 375–379, 2017.

[7] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *Journal of Industrial Information Integration*, vol. 6, pp. 1–10, 2017.

[8] L. D. Xu, E. L. Xu, and L. Li, "Industry 4.0: state of the art and future trends," *International Journal of Production Research*, vol. 56, no. 8, pp. 2941–2962, 2018. [Online]. Available: https://doi.org/10.1080/00207543.2018.1444806

[9] M. C. Zizic, M. Mladineo, N. Gjeldum, and L. Celent, "From industry 4.0 towards industry 5.0: A review and analysis of paradigm shift for the people, organization and technology," *Energies*, vol. 15, no. 14, 2022. [Online]. Available: https://www.mdpi.com/1996-1073/15/14/5221

[10] S. El Kadiri, B. Grabot, K.-D. Thoben, K. Hribernik, C. Emmanouilidis, G. von Cieminski, and D. Kiritsis, "Current trends on ict technologies for enterprise information systems," *Computers in Industry*, vol. 79, pp. 14–33, 2016, special Issue on Future Perspectives On Next Generation Enterprise Information Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361515300142

[11] T. Keyzer and M. D. Tebbens, "Netbeheerders: Stop met zonneparken daar waar nauwelijks vraag naar stroom is," Feb 2023. [Online]. Available: https://nos.nl/nieuwsuur/artikel/2465631-netbeheerders-stop-met-zonneparken-daar-waar-nauwelijks-vraag-naar-stroom-is

[12] M. Woudstra, "Designing a container management solution to improve flexibility and portability, and reducing cost for ipaas solutions." May 2022. [Online]. Available: http://essay.utwente.nl/90612/

[13] S. Eismann, J. Scheuner, E. V. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, 2022, cited By :2.

[14] A. Ivanov, "Kubernetes vs. serverless: When to use and how to choose?" Jun 2021. [Online]. Available: https://dysnix.com/blog/kubernetes-vs-serverless-part-2/

[15] "Serverless framework," Mar 2022. [Online]. Available: https://en.wikipedia.org/wiki/Serverless_Framework#:~:text=Serverless%20supports%20all%20runtimes%20offered,2015%20under%20the%20name%20JAWS.

[16] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele Univ.*, vol. 33, 08 2004.

[17] F. Kamei, I. Wiese, C. Lima, I. Polato, V. Nepomuceno, W. Ferreira, M. Ribeiro, C. Pena, B. Cartaxo, G. Pinto, and S. Soares, "Grey literature in software engineering: A critical review," *Information and Software Technology*, vol. 138, 2021.

[18] C. Miyachi, "What is "cloud"? it is time to update the nist definition?" *IEEE Cloud Computing*, vol. 5, no. 3, pp. 6–11, 2018.

[19] K. Hashizume, E. B. Fernandez, and M. M. Larrondo-Petrie, "A pattern for software-as-a-service in clouds," in *Proceedings of the 2012 ASE International Conference on BioMedical Computing, BioMedCom 2012*, 2012, pp. 140–144.

[20] "Cloud-apps en platform." [Online]. Available: https://www.salesforce.com/nl/products/

[21] "Zakelijke oplossingen voor stroomlijning en samenwerking — google workspace." [Online]. Available: https://workspace.google.com/intl/nl/business/

[22] "Invoice and accounting software for small businesses," Jun 2022. [Online]. Available: https://www.freshbooks.com/

[23] 2022. [Online]. Available: https://www.emagiz.com/

[24] 2022. [Online]. Available: https://boomi.com/platform/

[25] M. Pezzini and B. Lheureux, "Integration platform as a service: Moving integration to the cloud," Mar 2011. [Online]. Available: https://www.gartner.com/en/documents/1575414

[26] T. Neifer, D. Lawo, P. Bossauer, and A. Gadatsch, "Decoding ipaas: Investigation of user requirements for integration platforms as a service," in *Proceedings of the 18th International Conference on e-Business, ICE-B 2021*, 2021, pp. 47–55.

[27] M. Boisvert, S. J. Bigelow, and W. Chai, "What is iaas? infrastructure as a service definition," Dec 2020. [Online]. Available: https://www.techtarget.com/searchcloudcomputing/definition/Infrastructure-as-a-Service-IaaS

[28] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Analyzing open-source serverless platforms: Characteristics and performance," in *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, vol. 2021-July, 2021, pp. 15–20.

[29] E. Marin, D. Perino, and R. Di Pietro, "Serverless computing: a security perspective," *Journal of Cloud Computing*, vol. 11, no. 1, 2022.

[30] C. Kidd and S. Wickramasinghe, "Serverless vs function-as-a-service (faas): What's the difference?" Aug 2021. [Online]. Available: https://www.bmc.com/blogs/serverless-faas/

[31] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, and F. Leymann, "Faasten your decisions: A classification framework and technology review of function-as-a-service platforms," *Journal of Systems and Software*, vol. 175, 2021.

[32] I. Mohedas, S. R. Daly, R. P. Loweth, L. Huynh, G. L. Cravens, and K. H. Sienko, "The use of recommended interviewing practices by novice engineering designers to elicit information during requirements development," *Design Science*, vol. 8, p. e16, 2022.

[33] W. Adams, *Conducting Semi-Structured Interviews*, 08 2015.

[34] P. Achimugu, A. Selamat, R. Ibrahim, and M. Mahrin, "A systematic literature review of software requirements prioritization research," *Information and Software Technology*, vol. 56, 06 2014.

[35] R. Rabbah, "The state of openwhisk," Mar 2018. [Online]. Available: https://medium.com/@rabbah/the-state-of-openwhisk-ae8c129e8a48

[36] "Knative is an open-source enterprise-level solution to build serverless and event driven applications." [Online]. Available: https://knative.dev/docs/

[37] S. Liao, "Boomi's planned improvements to its hosting environment," Oct 2021. [Online]. Available: https://community.boomi.com/s/article/dellboomisplannedimprovementstoitshostingenvironment

[38] "Supported cloud regions." [Online]. Available: https://docs.workato.com/datacenter/datacenter-overview.html#data-center-locations

[39] J. Arnowitz, M. Arent, and N. Berger, "Effective prototyping for software makers," *Effective Prototyping For Software Makers*, 01 2007.

[40] "Amazon elastic kubernetes service (eks)," https://aws.amazon.com/eks/, 2023.

[41] "Google kubernetes engine (gke)," https://cloud.google.com/kubernetes-engine, 2023.

[42] "Azure kubernetes service (aks)," https://azure.microsoft.com/en-us/products/kubernetes-service/, 2023.

[43] "Free kubernetes," https://github.com/learnk8s/free-kubernetes, 2023.

[44] "Helm - the package manager for kubernetes." [Online]. Available: https://helm.sh/

[45] Prometheus, "Prometheus - monitoring system and time series database." [Online]. Available: https://prometheus.io/

[46] "Grafana: The open observability platform." [Online]. Available: https://grafana.com/

[47] "Openfaas autoscaling for the community edition." [Online]. Available: https://docs.openfaas.com/architecture/autoscaling/#legacy-scaling-for-the-community-edition-ce

[48] "Postman api platform." [Online]. Available: https://www.postman.com/

[49] "Cloud native certificate management - x.509 certificate management for kubernetes and openshift." [Online]. Available: https://cert-manager.io/

Appendices

A Interview Script

Serverless Technology within an iPaaS Interview Script Julian Elsten November 2022

A.1 Introduction

What is the research about: In this research the applicability of serverless technologies for iPaaS solutions in order to improve scalability and reduce operational cloud costs.

What will be done with the answers of this interview: The answers of this interview are used for the scientific research into constructing a method for implementing serverless technology for the field of iPaaS providers. The answers will not be made publicly available to anyone else other than the researcher and the supervisors of the research.

Goal of this interview: To come up with specific functional and non-functional requirements for the final solution that incorporates serverless technology

Ask for permission to record the interview and make sure to state the given permission when recorder is running.

A.2 About the interviewee/stakeholder

1. What is your name and what is your function title/role?
2. What are your responsibilities when looking at the iPaaS solution?
3. To whom are you responsible for performing these tasks?
4. What systems or software do you program on, on a regular or daily basis?

A.3 About the iPaaS solution

1. Can you describe the current architecture (Network and software stack) of the iPaaS solution?
2. What (Open Source) Frameworks are currently in use within the eMagiz Stack?
3. How are these frameworks/solutions chosen?
4. What are important considerations for choosing a certain framework/solution?
5. When looking at performance, are there specific numbers that need to be met for certain processes within the iPaaS solution? (Speed, latency, scalability etc?)
6. Are there any legal requirements or other regulatory requirements that need to be met?
7. How would you describe the current workload on DevOps/Cloud technology?
8. How is security and isolation taken care of within the iPaaS solution? (And how is serverless?)

A.4 Serverless

1. Can you describe the term “serverless” in your own words?
2. Would (part of) the code of the iPaaS solution do you think would benefit of running serverless the most?
3. Which team would need to maintain the serverless components of the infrastructure?

A.5 Finances

1. How are cloud infrastructure costs monitored?
2. Are Cloud infrastructure costs typically very stables from month to month or do they have peaks and lows?
3. What are the typical clouds costs per month and what services are delivered for these costs?
4. How is pricing setup for the clients of the iPaaS solution?
5. What is a client exceeds bandwidth/storage limits?
6. What do you think of a pay per use pricing model?
7. How can serverless costs be best estimated?

A.6 Other systems on the market

1. Are you familiar with other systems available in the iPaaS market?
   (a) What is their size?
2. How does their Architecture differ from yours architecture?
3. Why does it differ from yours?
4. Do you know of serverless plans for the future in any of the competition products?
5. Do other systems program in java as well or mostly in other languages?

A.7 Conclusion

1. Are there any other questions you think I should be asking, or anything else you want to tell me?
2. Can I contact you again if I need to ask some follow-up questions?
3. Would you want to participate in a review of the requirements later on?