




Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Master's in Computer Science	Spring semester, 2019 Open / Confidential
Author: Hafiz Rayaan Shahid	 (signature of author)
Programme coordinator: Antorweep Chakravorty Supervisor(s): Antorweep Chakravorty, Prakhar Srivastav	
Title of master's thesis: Refactoring Monolithic application into Cloud-Native Architecture	
Credits: 30	
Keywords: <ul style="list-style-type: none">• Monolithic application• Cloud-Native application• Microservices architecture• Google Cloud Platform	Number of pages: 66 + supplemental material/other: None Stavanger, June 15, 2019

UNIVERSITY OF STAVANGER

MASTER THESIS

Refactoring Monolithic application into Cloud-Native Architecture

Author:
Hafiz Rayaan SHAHID

Supervisor:
Antorweep CHAKRAVORTY
Prakhar SRIVASTAV

Department of Electrical Engineering and Computer Science
Faculty of Science and Technology
University of Stavanger

June 15, 2019

Abstract

Cloud Native is an approach, using which, applications are developed and run in such a way as to exploit and use the maximum features of cloud computing. Going cloud native does not just mean deploying your code to the cloud but following some rules and patterns to develop the application from the start. We want to evaluate the performance of an on-premise application with a parallel cloud native application through experimentation. We developed a cloud native application from scratch, using Platform as a service (PaaS) by Google cloud platform (GCP), following all the patterns that will be described in this report. GCP provides PaaS to develop, run and manage your applications without having to manage the infrastructure. This cloud native application is a counterpart of a monolithic and on-premises application called DUP (Delivery Platform). The performance analysis is based on metrics e.g. zero-downtime deployment, continuous deployment, automation of DevOps, extensibility of microservices, effective provisioning and efficient roll out strategies. We discover that the cloud native architecture performs better under these metrics.

Acknowledgements

First of all, I would like to thank my project advisor Mr. Antorweep Chakravorty who helped me throughout the project with the report. I acknowledge his support towards my thesis and that it is, how it is now, because of him.

I would also like to thank my advisor from Sysco AS, Mr. Prakhar Srivastav, to help me do the implementation part of the thesis. Also, Mr. Hung Huynh who came up with the idea together with Mr. Prakhar.

I acknowledge that without their support, this thesis would not have been as good as it is now. And for that, I am thankful . . .

Contents

Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Chapters	2
2 Background	3
2.1 Cloud Computing	3
2.1.1 Service Models	4
2.1.2 Deployment Models	5
2.1.3 Cloud Computing Characteristics	7
2.1.4 Application Workloads	8
2.1.5 Cloud-Native Properties	10
2.2 Microservices Architecture	13
2.2.1 Characteristics	14
2.2.2 Benefits	15
2.2.3 Complexities	16
3 Related Work	19
3.1 Event Driven Cloud Native Application	19

3.1.1	Objective	19
3.2	Evaluation of Two Deployment Patterns	20
3.2.1	Bring Your Own Code	20
3.2.2	Bring Your Own Container	20
3.3	Cloud Native Applications	21
3.4	Cloud Application Architectures	21
4	Design and Implementation	23
4.1	On-Premise Solution	23
4.1.1	Architecture	23
4.1.2	Application workflow	26
4.1.3	File Based Ingestion	26
4.1.4	Multi-Cloud Support	26
4.2	Google Cloud Platform	27
4.2.1	Cloud Storage	27
4.2.2	Cloud Function	28
4.2.3	Kubernetes Engine	28
4.2.4	Pub/Sub	28
4.2.5	Bigtable	28
4.3	Cloud-Native solution	28
4.3.1	Architecture	29
4.3.2	Implementation	30
	Module 1	30
	Module 2	33
	Module 3	37

Github	37
5 Experimentation and Evaluation	39
5.1 Setup	39
5.2 Decreased time to market	39
5.2.1 Provisioning and Scaling	40
5.2.2 Continuous Integration	41
Zero Downtime	42
5.3 Extensibility and Security	42
5.3.1 Extensibility	42
5.3.2 Security	42
5.3.3 Agility	43
5.3.4 Observability	44
6 Conclusion and Future Work	47

List of Figures

2.1	Properties of cloud computing	4
2.2	Component Management	6
2.3	Relation between cloud characteristics, service and deployment models	9
4.1	DUP architecture	24
4.2	DUP workflow	25
4.3	File based ingestion	27
4.4	DUP cloud architecture	29
4.5	Project description	31
4.6	Module 1 workflow	31
4.7	Cloud function deployment	32
4.8	Subscription Details	33
4.9	Module 2 workflow	34
4.10	cloudbuild.yml code snippet	35
4.11	Cloud build trigger	35
4.12	Meta data logs	36
4.13	Data as Json object	36
4.14	Module 3 workflow	37
4.15	Bigtable instance	38

4.16	Bigtable Monitoring	38
5.1	Comparing Provision and Scaling	40
5.2	Deploying Trigger Function	45
5.3	Deploying Parser Service	45
5.4	Deploying Consumer Service	46
5.5	Average for Adding Feature	46

List of Tables

5.1 Extensibility	43
5.2 Security	43
5.3 Agility	44
5.4 Observability	44

Chapter 1

Introduction

In the inception of internet, physical servers were used to manage the infrastructure that runs the applications. Physical servers are quite useful because we can configure them according to our needs and are very powerful. The failure rate is considerably low because of maximum power supplies, fans etc. But, all of this comes with a lot of unnecessary overhead. Physical servers are noisy, hard to maintain since they require man power to keep them alive and are quite expensive. The servers can never be utilized to their maximum potential. When you want to run multiple applications on one server, conflicts related to software and network routing arise because of trying to utilize maximum out of a server's processing power [14]. In recent years, there has been a considerable leap in technology due to which large amount of data are being generated and consequently requiring large storage and excessive processing power. All of this data needs to be stored and processed somewhere that is much stronger than physical on-premise servers and easy to manage without wasting man power and money.

Here, cloud computing provides the required services. Cloud is infrastructure, hardware and software, provided over the internet as a service. So that companies can get rid of their noisy, expensive on-premise servers [4]. Computing power, storage and other services are provided on demand. So, unlike physical servers, companies do not need to worry about the overhead and not being able to utilize their resources to the maximum. You only pay for what you use and can easily scale up or down depending on the requirement. Now a days, organizations are migrating their applications to cloud. Some are also migrating the whole infrastructure, software and hardware. But moving your systems to cloud does not necessarily mean that you are utilizing the cloud to its maximum potential. According to a research [7], There are some patterns and strategies, using which, you should make an application that exploits the cloud usage to its true potential. These applications are formally known as Cloud Native Applications(CNA). Monolithic applications, defined as being a big block of code without any loosely coupled components, are different from CNA. Recently, with the rise of cloud computing, companies started

getting rid of their monolithic applications and going to cloud or developing the cloud native version from the scratch. This need for creating cloud native applications with the same functionality of the already existing monolithic on-premise applications has been the motivation for the work provided in this thesis.

1.1 Chapters

This section we briefly talk about all the chapters written in this thesis. The workflow of this report is based on 6 chapters. Each chapter has its own content and challenges that we will discuss here briefly and completely inside the chapters. Following are the chapters that construct this report in a sequence:

- Chapter 1: Introduction
It contains the introductory material to the thesis topic and how it affects the industry today.
- Chapter 2: Background
It consists of of the history, background information and theoretical concepts that are important to understand the work done.
- Chapter 3: Related Work
It consists of two research papers related to our topic and how we build on or use their work in our thesis.
- Chapter 4: Design and Implementation
It contains the detailed information of the design and how we implement the application.
- Chapter 5: Experimentation and Evaluation
It contains the experiments performed on the implementation to evaluate the application.
- Chapter 6: Conclusion and future directions
It concludes the thesis with discussing the findings of the thesis and gives future recommendations for research.

Chapter 2

Background

This chapter sets the background to the thesis. It gives the necessary information to understand the topics related to the implementation part.

2.1 Cloud Computing

The past few years showed that the data are being generated increasingly [2]. To handle this huge amount of data, we need excessive computing power and storage. Having physical server's on-board comes with a heavy price. It is expensive and needs special manpower to maintain it. Moreover, it can never be fully utilized by running multiple applications on it because of network routing and software conflicts. To overcome these problems, cloud computing has been introduced. Cloud computing can be defined as a virtualized, managed, scalable, distributed shared pool of resources that provide computing power as a service [4, 37]. The resources can be created and used on-demand as per convenience without having to worry about wasted computing power and other resources. Cloud computing has different meaning for different companies. The companies that provide the cloud services are known as cloud providers. They are responsible for managing the cloud resources and keep them running. Cloud consumers are the companies or individuals who access the cloud services. These are mostly the companies that don't want to manage and buy the hardware that supports applications. Alternatively, they use cloud services to build applications so they can focus more on the business side.

The benefits of cloud computing gave the world a different view of application development. Many applications are developed on cloud and many are being migrated. Applications in the field of Data analytics, IoT, Distributed systems, Streaming engines are widely developed using cloud services. The popularity of cloud

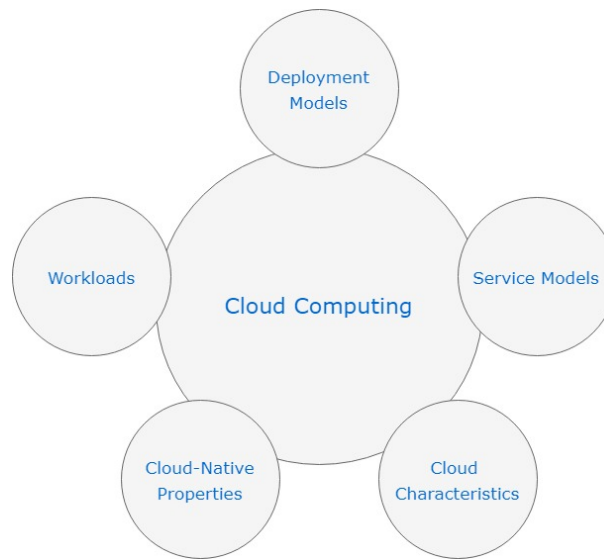


FIGURE 2.1: Properties of cloud computing

computing is peaking and has resulted in different cloud service deployment models and strategies, Service models and other characteristics [2]. Figure 2.1 shows the properties of cloud computing, further we will study them in detail.

2.1.1 Service Models

Different service models are used to provide cloud resources to the consumers. The service models vary in the levels of abstraction that range from providing software as a service down to a basic processing power or entire infrastructure to create applications. According to a definition [27, 36, 2], there are three cloud service models which can be defined as:

- **Infrastructure as a Service (IaaS)**

Infrastructure as a service (IaaS) is a cloud service in which a cloud provider provides the consumers with computing resources such as data storage and networking facilities, servers. Companies use this infrastructure together with their own platform to run and manage applications. Key features include paying for IaaS on demand rather than buying the hardware and having to maintain it. It saves a lot of resource and money. Moreover, it can be scaled up or down depending on the storage and computing requirements.

- **Platform as a service (PaaS)**

It is a cloud service that provides its consumers with a platform using which, they can build and manage their applications. Similar to IaaS, consumers get processing and storage resources. Additionally, in this service model, consumers get prebuilt tools by the provider to customize their applications and test them. Companies can focus entirely on development without worrying about managing the infrastructure. Cloud providers are in charge of managing underlying operating system, server and security concerns. Teams can work in a collaborative effort remotely on the same application.

- **Software as a service (SaaS)**

It is cloud service model in which a user is provided with the cloud provider's cloud based software. Unlike IaaS and PaaS, consumers do not have to develop a certain application to use it neither install it on their side. They can access the software, that resides on provider's server, as a service through the internet or an API. Users can use this software to store and manage their data. It is a subscription model. The stored data on SaaS is secure and will not result in a loss in case of failure.

When an organization wants to use these service models, they need to know what components they have to manage and which are providers responsibility. Figure 2.2 shows the relation of managing components in regards to different service models.

2.1.2 Deployment Models

Most cloud providers have hubs containing hundreds of servers that help load the data faster to a user. Often, data is brought closer to the consumers by geographic location to support fast loading. Cloud Deployment Models are based on the location of data. There are four deployment models that can be defined as follows [27, 36, 2]:

- **Private Cloud**

It is an infrastructure used by self-contained companies. This cloud infrastructure can be managed by the company itself or a third party organization. It gives a higher level of security. Data can not be tampered because it is backed up internally, using a firewall and can be hosted both externally and internally. Private clouds is mostly used by companies that require high security, data availability, and high management.

- **Public Cloud**

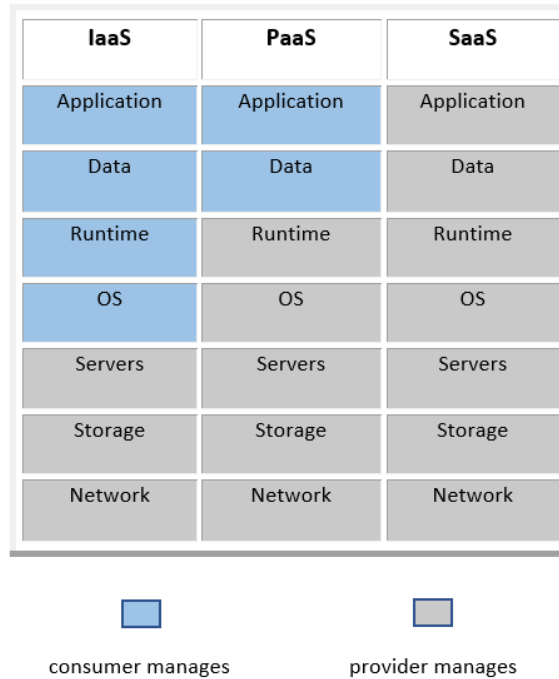


FIGURE 2.2: Component Management

This cloud infrastructure can be used by public on a provided network. Public does not mean that consumer's information is publicly available but that consumers can access it using different access controls. The infrastructure is not controlled by the consumers. It is based on different paying policy such as pay per user or a shared cost model for all the consumers. Public cloud is best suited for companies whose requirements are changing constantly. It is quite popular for web application development and data storage among different sized companies.

- **Community Cloud**

This cloud service is shared by organizations working in a particular community, for example government associations, commercial organizations, hospitals, banks etc. The members of one community share similar concerns such as security, performance and privacy. This cloud deployment model can be managed by a company itself or a third party organization [2].

- **Hybrid Cloud**

This deployment model includes the properties of public and private clouds, but each may work as a stand alone entity. In this deployment model, external and internal providers can provide the resources [1]. It is ideal for scalable

and secure applications. A company might need a Hybrid cloud to create applications based on the application's requirement. An example would be an application that stores data in a secure way using private cloud but communicates with the users using public cloud.

2.1.3 Cloud Computing Characteristics

Today, it is essential that organizations maximize the potential benefit of cloud computing. For that, knowledge of cloud characteristics is very important. Cloud computing is defined by (NIST) National Institute of Standards and Technology through following characteristics [27, 28].

- **On-demand self-service**

Provisioning cloud resources is possible from the cloud provider without human interaction. Basically, an organization can provide needed resources without contacting to the cloud provider. The resources might include VM instances, Databases etc. These organizations need to access a web portal that allows them to access and use the cloud services and provision and de-provision services On-demand [27].

- **Broad network access**

Many consumer platforms are provided over the network to access the cloud resources. Primarily, services are available on a network that is a high broadband link, for example the internet. On the other hand, Private cloud might use a local connection such as (LAN) local area network. The two important aspects of broad network access and cloud computing are latency and Network bandwidth. Since these aspects are related to quality of service, a very important part of cloud computing. It mainly affects the applications that are time sensitive[27].

- **Multi-tenancy and resource pooling**

Cloud services are build in such a way to support multi-tenancy. Multi-tenancy lets multiple consumers use the same infrastructure or the application while still maintaining their data security and privacy. A real life example would be people living in a shared apartment and still have their rooms to have their privacy and security of personal belongings.

On the other hand, resource pooling means that same resource is used to service multiple customers. The resource pool of the cloud provider should be large enough to provide service to every consumer and economy of scale.

- **Rapid elasticity and scalability**

A very important part of cloud computing is that organizations can be provisioned with resources when they need them and can remove them when they are finished. Also, resources can be scaled up or down in accordance with the requirements. When organizations scale down the resources to their needs, they cut down cost.

Basically, elasticity is related to a rapid provision and de-provision of resources that might include VM instances, Databases or whole applications[27]. On the other hand, scalability is a gradual and planned process. For example, organizations plan to scale up or down over a period of time and cloud can handle that.

Testing and developing applications is directly related to rapid elasticity and scalability. If an organization requires a few VMs to test an application before deploying into production, they can just create the virtual machine instances right then without having to wait for the physical ones.

- **Measured service**

The consumer organizations only have to pay for what they have used because the resources are being metered. Charge-per-use capability can be leveraged to optimize resource utilization. The cloud providers are monitoring the use of their resources and charge the consumers accordingly [11].

At this stage, it is important to show the relation of three main properties of cloud computing to each other. Figure 2.3 is a visual representation of the interaction of cloud service models, cloud deployment models and the essential characteristics.

2.1.4 Application Workloads

In cloud computing, workloads is the usage of cloud resources within a given time. It will help to provision or de-provision the resources by understanding workloads [11, 2]. Following are the workloads that the cloud applications may encounter:

- **Static Workload**

It has optimized the the utilization of the cloud resources within specified period of time. Provisioning or de-provisioning is not required in this case. Since, At the time of provisioning, the required resources are provisioned and at the same time a small amount is over-provisioned to handle the overhead [2].

- **Periodic Workload**

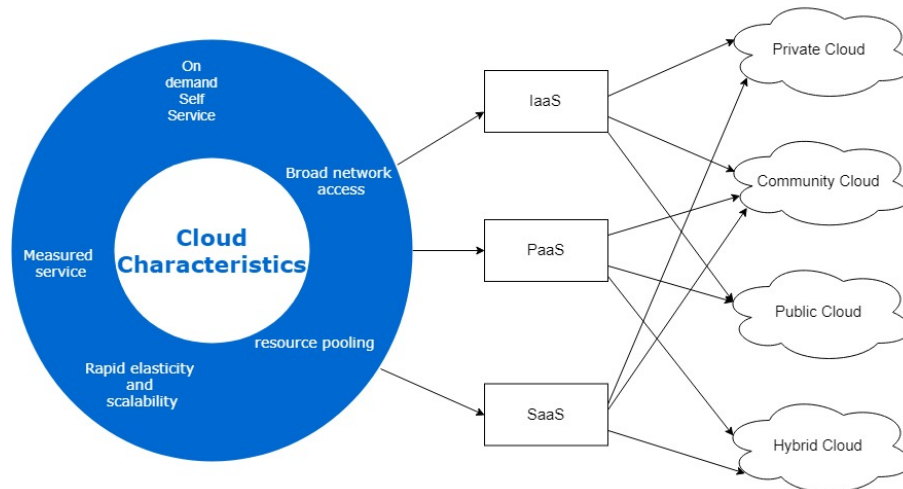


FIGURE 2.3: Relation between cloud characteristics, service and deployment models

This workload is important for periodic tasks for example paying the salaries to the employees every month. This results in peak periodic utilization. So, when there is peak time, consumers need more cloud resources and want to get rid of extra resources once peak time is finished. One way to scale would be to statically scale up or down as in static workload. Another way would be to scale elastically. In this case, over-provisioning is not required since the resources are exactly how much needed at all times [2].

- **Once-in-a-Life Workload**

In this case, the resources are needed after a long time. This might happen in the case of E-commerce applications where the user orders a specific product. So, the peak time for the utilization of resources for this application would only when an order is placed. We need to fill the under utilization gaps for this workload. One way would be statically scaling, which might be inefficient due to possible under provisioning. Another way is to scale automatically where just the required resources are used.

- **Unpredictable Workload**

This workload occurs when the resources are utilized randomly and in unpredictable times [2]. Static scaling can not be used in this case, due to the randomness and unpredictability of the resource usage. So, the alternative option, elastic scaling would be ideal for this case and predicting the peak time is not necessary.

- **Continuously Changing Workload**

This everlasting changing workload is due to the continuous fluctuation in the utilization of resources. The rate of fluctuation remains constant. Both static and elastic scaling can be used to cope with this workload [2]. In static scaling, we can divide the provisioning and de-provisioning into steps such as we can provision 50 GB storage space when needed and de-provision 50 GB when not. Alternatively, in elastic scaling, resources are provisioned consistently with changing requirements.

2.1.5 Cloud-Native Properties

There are different scaling techniques for each workload and thus requirements of the application design are also different. The most popular design is to divide the application into smaller components and manage them separately. But, splitting the application into smaller components is not enough to take full advantage of the cloud environment. The application has to be designed in such a way as to ensure the maximum utilization is cloud services [2]. Following are the properties that a cloud native application should have [11, 12, 2]:

- **Distribution**

There is a big pool of resources on which the cloud resources are distributed in a scattered way. The cloud-native application should be developed in such a way that it should be consistently distributed with the resources [2].

- **Elasticity**

Elasticity and scalability can be used interchangeably [35]. It can be defined as the ability to manage the increasing requirements with a maintained cost [6]. Elasticity must automatically cope with different workloads. Therefore, cloud-native applications should be developed to scale as required. There are two ways of achieving scalability, horizontally by adding more cloud resources or vertically by adding more functionalities to the existing resource.

- **Isolated State**

The cloud-native applications should be stateless. This requirement gives rise to isolated state property. Elasticity will result in scaling the applications up or down, which means applications will have more instances and thus applications need to be stateless. Each instance of the application has to be managed as a stand-alone component [2]. However, the states can be stored in a database to be retrieved as required.

- **Automated Management**

Automated management has a real importance when elasticity results in provisioning and de-provisioning of resources. There should be a way to monitor the changing requirements and provide the required resources and remove the unneeded resources.

- **Loosely Coupled**

The different components of a distributed system sometimes depend on each other to work consistently. These components need to be independent from one another by eliminating the dependency. So that scaling of applications is easily done and failing components do not affect the other parts of the application [11].

The twelve factor app Building cloud-native applications can be very tricky due to the continuous augmentation, constant collaboration of the teams working on it. There must be guidelines that can be followed to design and develop cloud-native applications. A methodology called The Twelve-Factor App exists, that gives guidelines to build applications with following properties [24]:

- It is a declarative format that results in automation.
- It is portable in different environments and platforms.
- It is compatible with multiple cloud platforms.
- The development and production parts are totally divided which gives rise to continuous deployment.
- It is highly scalable and does not have any effect on the architecture itself [24].

Twelve Factors

1. Codebase

A version control system tracks the different versions of the code, for example git [2]. Companies do not need different codebase for different deployments. A single codebase can be shared among all the deployments like staging, production.

2. Dependencies

A dependency manager should be used by a cloud-native application like NPM (NodeJS Package Manager) and dependencies should be declared explicitly through some dependency declaration tool. This simplifies the application and its settings.

3. **Config**

The config factor is a configuration setting of an application's different deployments like staging, production etc [2]. The config of an application includes all the resources used such as a backing service, pre-deploy settings for different deployments or any credentials required. This config is not the same as the application's internal configuration and should not be included within the application. But, at the same time this configuration should be easily accessible to change between deployments like environment variables.

4. **Backing services**

These are the services used by the applications to attain its standard operations such as data storage services (BigTable, BigQuery etc) [2]. A cloud-native application should treat the internal and external services as the same.

5. **Build, release, run**

The codebase can be deployed into production using different steps [2]:

- **Build:** The code is transformed into builds, an executable format.
- **Release:** This stage combines the build produced in the previous stage with a config specific to the deployment. Consequently, a release that includes the build and the config is created.
- **Run:** This step is responsible for running the release generated in the release step in an execution environment.

6. **Processes**

Multiple processes might be used to execute a cloud-native application. These processes are stand-alone and do not share anything among each other [2]. Thus, the application is stateless.

7. **Port binding**

The twelve factor application binds a port to a Hypertext Transfer Protocol service to provide a web facing interface without having to rely on webserver injection. The application is totally independent of web servers for providing web interface.

8. **Concurrency**

Application's workload is divided into several processes. Each process has a type which corresponds to a certain type of work [2]. All work of the same type is assigned to that process which. All the work is being done by different processes at the same time.

9. **Disposability**

The processes in these applications can be thrown away at any time. The usage depends on the application's requirements and when a process is not needed anymore, it is disposed.

10. **Dev/prod parity**

The deployment stages like development, staging and production should not be very different from one another [2].

11. **Logs**

The events on the application can be monitored easily if the logs are well designed and treated as stream of events.

12. **Admin processes**

The administrative work is done by a one-off process. The dependencies that support the admin process are isolated.

2.2 **Microservices Architecture**

The applications developed and managed as a single big chunk of code are known as monolithic applications. Monolithic, in the start of development process, look very simple and easy to understand. It's comparatively easier to change some functionality without having to worry about other services. But, These benefits vanish off with the increasing complexity of the application. They suddenly are hard to grasp and agility becomes a problem when developers have to make a lot of effort just to make small changes. The most effective way to deal with increasing complexities of the applications is to divide and separate the common functionality into a separate service that iterates independently. Consequently, agility is increased by letting the developers make changes to just the service they want without effecting other services [2]. Every microservice is a separate entity which can be handled by separate teams, can be written with the choice of their language and can be scaled up or down as required.

We can say that this architecture not only scales up or down as required but resists failures and is known as microservices architecture [29]. This architecture divides the application into smaller services with unique operations. The microservices are developed and designed by keeping the cloud-native application properties in mind, which are already discussed. In order to have deeper insight on microservices architecture, we need to look at the characteristics and other properties that govern them.

2.2.1 Characteristics

A microservices application does not have to obey all of these characteristics, but it should be a minimum of these [9]. Following all of them will make a consistent, extensible and failure resistant application.

- **Business oriented**

The applications are broken down to smaller components called microservices. The boundaries along which the application is split, should be properly created so that the system integrity remains intact. So, these microservices are supposed to be developed based on the business needs. This can take a lot of effort from different teams working in a company.

- **Design for failures**

When the application is split into smaller microservices, the failures are inevitable. The developer who creates the service should make sure that a failed microservice does not kill the whole system. There are monitoring systems that find the failures throughout the application lifetime. We can quickly fix the failures on detection [26].

- **Decentralized Data Management**

Since microservice is a loosely coupled application that has everything inside it that it needs. It should also have its own persistent unit such as database. This brings in a lot of options to use as databases such as Postgres, Google Bigtable. These databases can be SQL or NoSQL depending upon the requirements [2]. Although, in some cases a centralized data management system is required to inform all the microservices about a transaction. But, this can be done using a messaging system such as Kafka. So, keep the databases completely decentralized so they can remain loosely coupled.

- **Discoverability**

The infrastructure used to create an application can fail. So, its better to create microservices so that the configuration remains intact. If any service needs to connect to other services, it shall discover them easily. But, services can be found easily if all the services are registered in container registry system.

- **Inter-service communication design**

The microservices are deployed using different deployment strategies such as containers, virtual machines etc [2]. But how are they supposed to communicate with one another? This question is a very important to answer for a reliable microservices application. For one way communication, it is possible to use a messaging service like Kafka. But for two way communication we

would have to expose REST endpoints so that other services can create the http requests to reach out [2]. This way both microservices can talk to each other without having to risk security and they will still be loosely coupled. There are some cases where services want a chat like messaging or a queue of REST calls to get something done. In that case we can create another layer for messaging that stays between services and lets them chat with each other.

- **Evolutionary Design**

The requirements for the systems are always changing which results in the addition of new functionalities. To add a new feature, just the concerned microservice is updated and deployed. The entire system does not have to be updated [2].

2.2.2 Benefits

Many companies complain about the challenges they have to face with monolithic architecture related to scalability, agility etc. Now, we take a look at some of the important factors that makes us choose a microservices architecture [30].

- **Agility**

Using microservices architecture, small teams can be independent and own their services they develop. Each team has a role to play and understands it's small and independent job, which makes the development cycle speed up, decreasing the cycle times. The organization benefits greatly from this aggregated throughput [30].

- **Innovation**

Innovation comes with each small team can choosing it's own frameworks, technologies appropriate for it, and other important tools. The ownership of services is due to the responsibility that comes with it. The merge of the Development and operations, called DevOps, within a team decreases the possibility of clash of goals [30]. The deployment of agile processes is never stopped. Instead, the whole application deployment process shall be automated as continuous delivery, from committing the code to deploy the running service.

- **Quality**

Using the microservices architecture also improves code quality. Dividing the application and design into smaller modules is the same as in object oriented programming. It's easier to build, reuse, compose and maintain the whole structure of code.

- **Scalability**

Microservices should be strictly decoupled for a better development of very large applications [30]. This should be considered as a prerequisite for a better performance because it allows each service to be developed with its optimal tools, appropriate programming language, frameworks.

The services that are decoupled can be scaled independent from others and horizontally. Adding additional servers, Horizontal scaling, is dynamic since it does not care about the limitations of other servers. Running the application on a larger machine, Vertical scaling depends on the limitation of other servers and causes downtime during scaling. Moreover, Failing modules can be replaced automatically improving application resilience.

- **Availability**

The failure is isolated in one of the services and is easy to locate and fix using microservices architecture. We can use different techniques to decrease the chances of failure such as caching, health checking, monitoring etc. This increases the application's availability [30].

2.2.3 Complexities

Every architecture comes with its challenges and complexities, despite the benefits we discussed earlier. A microservices architecture also has its complexities. This section states some of those challenges and trade-offs [31].

- **Distributed Systems**

There are a set of problems called Fallacies of distributed systems that come along with microservices architecture since it falls under Distributed computing [31]. The developers that are beginners in distributed systems can assume that the network on which the communication is happening is reliable, the bandwidth is infinite and the latency is minimum, which is not the case.

- **Migration**

when migrating an application to a microservices architecture from a monolithic architecture, the right border line for microservices should be realized. The process is hard and requires developers to extract the dependencies that are connected down with the persistent layer [31].

- **Versions**

The process of versioning can be complex for microservices. There are some patterns that can be followed such as route based versioning and can be developed at API layer.

- **Organization**

An organization architecture goes side by side with the microservices architecture. An organization might face problems like creating efficient team structure, following the DevOps approach and design effective communication between development and operational tasks [31].

- **Architectural Complexity**

The challenges are inside the code repository in on-premise architecture. But, in microservices, the challenge is to make the services communicate with each other rather than having one big bunch of code [31].

Chapter 3

Related Work

In this chapter, we look into the work that has been done in this field and how we are going to build on it or use some of its features in this thesis. Cloud-native infrastructure is a comparatively new field but there is a significant amount of research done on this architecture. We are going to look into some of the studies that are related to our work and we will build on their research.

3.1 Event Driven Cloud Native Application

This study [2] is about serverless computing and how it affects software development. Serverless computing performs the best with event-driven systems. The demand for serverless computing is very high due to consumers wanting more efficient and easy ways to receive events with abstraction along with the challenges like handling communication, various implementations. Serverless applications are better in performance from the applications that use servers. There are a lot of cloud providers who are providing services that are deployed on cloud and do not use your own servers. Those services include Amazon Web Services Lambda, Google Cloud Functions and Microsoft Azure Functions [6]. Each service has some advantages and challenges that a developer might face.

3.1.1 Objective

The evaluation and comparison is performed on the above-mentioned serverless computing services. The experimentation is based on what challenges and limitations each service has and how substantial they are. It is important to define how much a service is dependent on other provided services to decrease the efforts of

developing a function. It means that service provider will be very strong if they have increasing number of services.

In our thesis, we have used an event driven and serverless function, Google Cloud Function. This is used to trigger every time a file is uploaded on Google Cloud Storage. The results in this thesis has helped us with choosing and using this service. Since Google Cloud Function is comparatively new and is still in Alpha build, there is more room for research.

3.2 Evaluation of Two Deployment Patterns

In this study [3], there is research and evaluation of the two deployment strategies linked to Cloud Native applications. There are no research work done in terms of patterns being available in the industry. So, there is still room for literature reviews and studies which gave rise to this study. The advantages and disadvantages of the deployment strategies are studied in this report and an application is developed using these deployment strategies. This application helps to evaluate the deployment patterns practically. There are discussions about the pros and cons of the findings and a decision tree is built for consumers to choose their required pattern. In our thesis, we have used this research between the deployment patterns and used it to develop our cloud native application. To know what each pattern offers, we should go through them briefly.

3.2.1 Bring Your Own Code

In this pattern, developers need to only care about the code they write and upload it. The cloud providers take care of creating the containers and deploying them on cloud [3].

3.2.2 Bring Your Own Container

In this pattern, containers are constructed by the developer and deployed on the cloud to run. The cloud is only responsible for running the containers which gives the developer more insight and control over the application [3].

3.3 Cloud Native Applications

In this research [13], there is a study about different key concepts of cloud native applications (CNA). There are some example CNAs which are built on these concepts. The study discusses the most important key factors in cloud-native architecture, one is microservices architecture and the other one is serverless computing and how cloud services are fully managed. These ideas are discussed in detail and their influence on the future of cloud applications is stated through analysis.

It concludes that the cloud providers are constantly developing services that are fully managed and the consumers do not have to manage the resources. So, the overhead of scaling and managing resources has shifted from the user to the vendors. These services work the same as any microservice in terms of being scalable and resilient. Also, some Serverless computing examples are Google cloud functions and AWS lambda. This study helped us with understanding the microservices architecture and serverless computing through examples and detailed analysis. We have used the microservices architecture to build our application and created a serverless function using google cloud functions.

3.4 Cloud Application Architectures

This research paper consists of cloud native architectures and how it got where it is and what is the future [25]. It shows the results of a development project that was about migrating an application from one cloud infrastructure to another one. As an additional study, the similarities and dissimilarities are analyzed of several cloud applications. Using this commonality study, cloud developers can take advantage designing applications wisely for each cloud provider. Throughout this paper, required mappings of cloud native applications are analysed, some more research papers have been studied and performed action research on cloud. This research has introduced two main factors of cloud computing, one is the evolution of the cloud applications is visualized as the evolution of making resource usage optimized. The other one is that these resource usage optimizations are because of cloud native applications being built in a completely different way after various standards have been set through research.

Chapter 4

Design and Implementation

In this chapter, we are going to analyse the on-premises application and see how we can recreate it in a cloud-native way and deploy it on cloud using Google cloud platform services, which we will later discuss in this chapter. The application that we are going to redesign according to cloud-native architecture already exists and is called Digital Utility Playground (DUP), a delivery platform [5]. We will go through it's architecture in the following following section.

4.1 On-Premise Solution

DUP is a Data Delivery Platform [5], whose main functionality is to receive data in different formats, take it through an ingestion system, perform data processing before finally storing it in a persistent unit to apply machine learning algorithms and to retrieve useful information. No cloud resources are used in this application. In other words, this solution is deployed completely on-premises.

4.1.1 Architecture

This application can be divided into three sub modules which are applications, infrastructure and security. Figure 4.1 shows how these modules are separated from each other. Following is the description of each module:

1. Applications

There are several sub applications working together to create the delivery platform such as Ingestion applications, Internal applications, Analytics and Integration services.

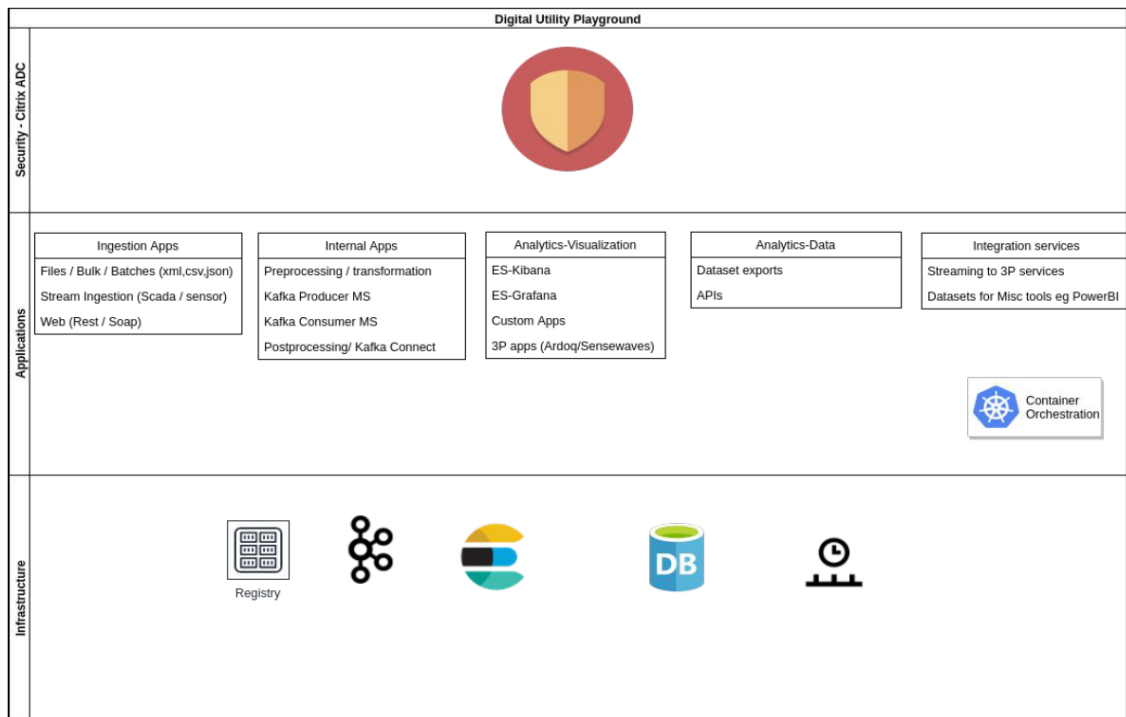


FIGURE 4.1: DUP architecture

Ingestion applications are used to ingest structured or unstructured data into the system from various sources. The sources include files/objects with different formats such as csv, xml and json, Stream ingestion from sensors or Rest/Soap API calls over the web.

Internal applications are used to perform different operations such as pre and post-processing on the data, producing records/items from the data to Kafka topics, consuming data from Kafka topics and store in a persistent unit such as SQL database or elastic search.

Analytics is used to visualize the data and query over it to retrieve useful information. Analytics is a fundamental part of data delivery, to study your data visually and use it in machine learning algorithms and tools.

Integration services can be provided with the processed data to use as required. There can be multiple integrations connected with this system that need the data coming in with different formats but processed into the same structure.

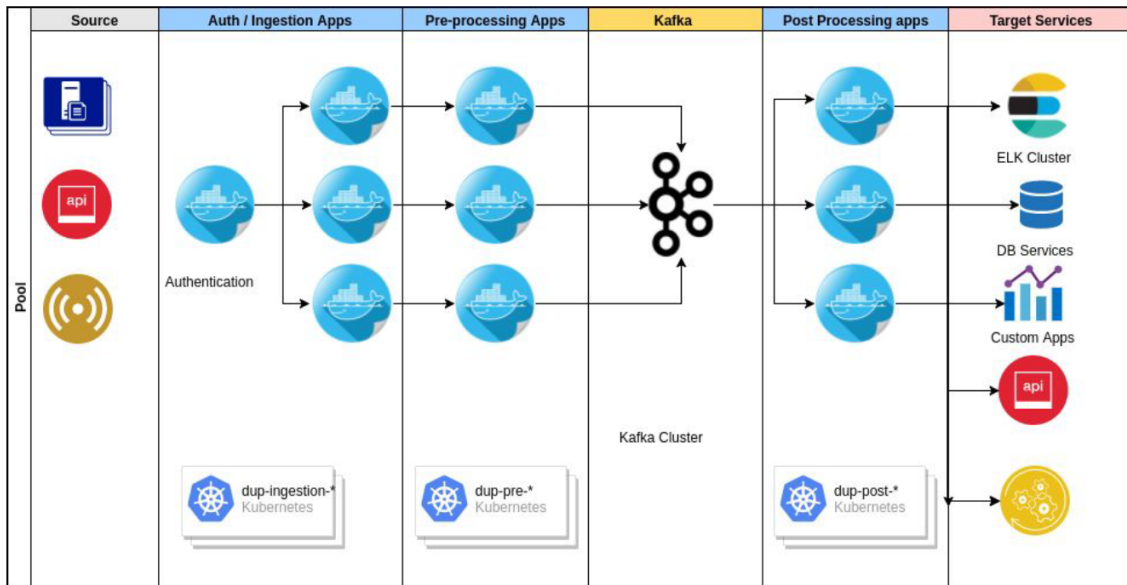


FIGURE 4.2: DUP workflow

2. Infrastructure

The infrastructure includes all the services used together with the applications. These services include:

- **Kafka Clusters:** It is used as a distributed messaging system to exchange data between the sub-applications.
- **Database:** It is used to store the processed data in a tabular form.
- **ELK Clusters:** It is used to store big data that is easily searchable.
- **Container orchestration:** The above mentioned applications are converted into containers and need to be managed by an orchestration system such as Kubernetes, Openshift, Hashicorp etc.
- **Active Registry:** It stores the container image with complete management facilities such as image signing and security scanning.

3. Security

Citrix Application delivery controllers (ADC) help make the applications adaptable to the protocols and networks widely used. They ensure the applications to be optimal, highly available and without security risks. They help in fundamental tasks such as authentication, request propagation and load balancing.

4.1.2 Application workflow

In the previous section, we discussed the existing modules in detail. Now, we look at how these modules are designed to interact with each other, from ingesting the data to storing it in the database. According to Figure 4.2, there are three main steps in the workflow:

- **Step 1:** The Authentication/Ingestion application takes the data from the source and passes it onto the pre-processing application.
- **Step 2:** The pre-processing application takes the data, applies required pre-processing algorithms on it and puts it on a Kafka cluster.
- **Step 3:** The post-processing application reads the data from the cluster, applies post-processing techniques on it and sends it to the target services.

4.1.3 File Based Ingestion

Earlier, we discussed about different data ingestion techniques that are used in DUP architecture. To name a few, they are Stream Ingestion (sensor), Web based ingestion (Rest/API) and File based ingestion (xml,json,csv). All of these ingestion are used in DUP but in this thesis, we are just going to discuss File based ingestion. Since, this is the use-case we will implement in a cloud native way to compare with this on-premises solution. According to Figure 4.3, File based ingestion has 5 steps.

- **1 and 2:** The customer logs into Citrix ADC for authentication.
- **3:** The customer uploads the files through DUP dashboard.
- **4:** The customer fills the required form and an upload request is sent. Moreover, the backend APIs identify the customer's information from headers.
- **5:** The files are uploaded on the customer's sftp directory.

4.1.4 Multi-Cloud Support

There are two ways to use cloud services in DUP architecture. First is that DUP can use cloud services like SQS/Kinesis or Dataflow/PubSub instead of setting up kafka. BigTable or Cloud SQL can be used as a targeted service. In this way, we don't need computing power from the cloud provider, just the storage resources. The applications are still running on Premise servers. The second way is that the whole

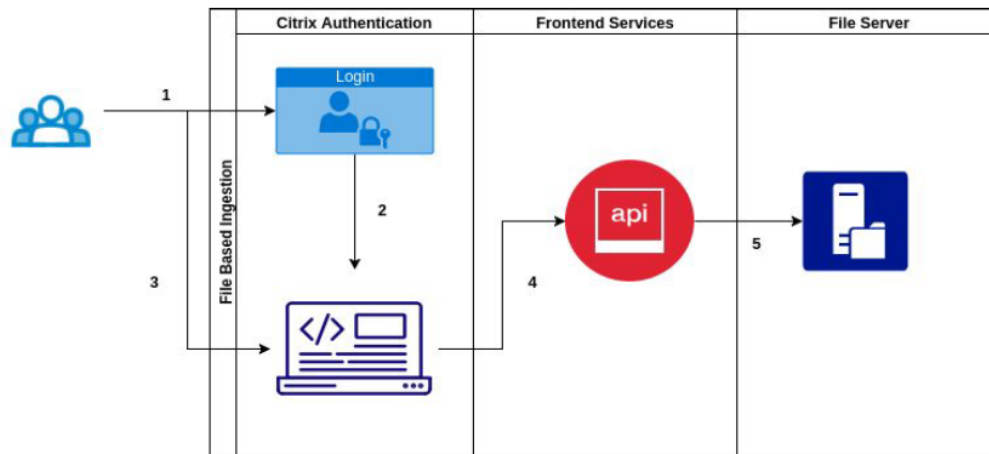


FIGURE 4.3: File based ingestion

application together with sub-applications and the entire infrastructure is migrated to cloud. This will require the computing resources like VM clusters to run the applications and also storage resources. This is the one that we are going to adapt for this thesis.

4.2 Google Cloud Platform

GCP provides the platform as a service to create cloud native applications. To do that, the on-premise application has to be redesigned in order to be consistent with GCP services. All GCP services related to storage, computing power, Data analytics run on the same infrastructure google uses for its own products like Gmail [10]. We are going to go through the services we are using in our application briefly to understand their importance.

4.2.1 Cloud Storage

Using this service, consumers can store and retrieve large amounts of data anywhere in the whole world [19]. Google Cloud storage can be used with various purposes such as serving a web application data, storing data for analysis or uploading big data objects like files to get downloaded later.

4.2.2 Cloud Function

Google Cloud functions are services that provides a platform to deploy functions in a serverless environment [18]. The developers do not need to worry about managing the infrastructure, cloud functions manage everything. These functions are built to be uni-functional and can be triggered on an occurrence of an event. The event that cloud functions are linked to, can be a file upload or delete on cloud storage or data published on Google Pub/Sub.

4.2.3 Kubernetes Engine

This is a container orchestration environment which lets the developers write the code and not worry about the deployment strategies [20]. It handles the deployments with automation instead of building pipelines. It is an open source service which makes it ideal for any kind of application might it be on-premise or on cloud. It basically provides us with a cluster of fully managed virtual machines that run our docker containerized applications. Also, lets us store the containers into container registry.

4.2.4 Pub/Sub

It is a messaging service which lets applications/microservices exchange data with each other [21]. It is asynchronous and fully decouples the applications. It consists of a component called topic which is used to publish and consume the data. The data can be of any format such as a string object or Json object.

4.2.5 Bigtable

It is a NoSQL data base service provided to the customers for storing and retrieving huge amount of data [8]. It is mainly used for analytics on Big Data and google uses it for its own products like Maps, search etc.

4.3 Cloud-Native solution

In this section, we will redesign and implement DUP architecture according to google cloud platform's services that we discussed in the previous section. We will follow

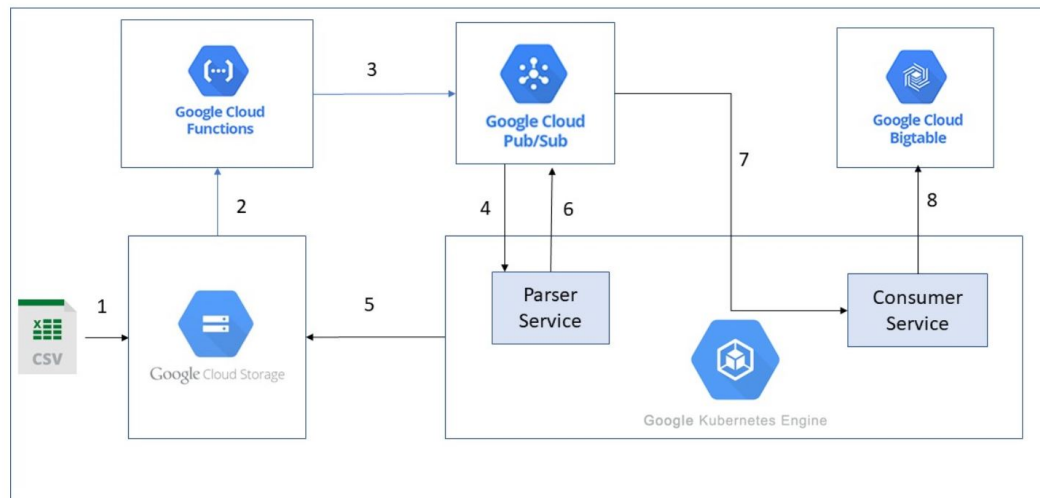


FIGURE 4.4: DUP cloud architecture

the rules and theoretical bounds for creating cloud native applications discussed in Chapter 2.

4.3.1 Architecture

The architecture for the cloud native application has to be developed again in order to cope with google cloud platform's services. GCP, like any other cloud provider, provides its consumers with their client API. This API can be used to connect to GCP's services and use them as required. Figure 4.4 shows the overall architecture of the cloud-native version of DUP. In the Figure 4.4, it is visible that only the File based ingestion scheme is used. The numbering in the figure represents which task is going to be performed first sequentially.

This architecture is based on microservice architecture, cloud-native properties and 12 factor app as discussed in chapter 2. We have divided the whole architecture into several modules. This helps us to develop each module as a separate and stand-alone unit. Lets discuss a high level description of each module. The numbering in each module step represents the numbering in the Figure 4.4.

- **Module 1**

1. Upload a .csv file to google cloud storage bucket.
2. Create and trigger cloud function on cloud storage (bucket) file upload.
3. Publish meta data of the file to google cloud Pub/Sub's topic "FileUpload-Topic" in that function.

- **Module 2**

Create and deploy a service (Parser) using KBE that:

4. Consumes meta data from "FileUploadTopic".
5. Reads file from the bucket.
6. Parses file content to JSON and Publishes each JSON object to "FileContentTopic".

- **Module 3**

Create and deploy a service (Consumer) using KBE that:

7. Consumes each JSON from "FileContentTopic"
8. Stores that JSON to "FileContentTable" in BigTable.

4.3.2 Implementation

Before we start implementing module 1, there are some prerequisites to complete. These pre-conditions include:

- Creating a project on GCP with command:
cmd [16]: *"gcloud projects create projectID"*
- Enabling billing for that project.
- Enabling cloud function, storage and other APIs to be used.
- Installing and setting up the google cloud SDK.
- Installing the updated gcloud components with the command:
cmd [16]: *"gcloud components update"*
- Setting up the development environment such as Python, Java, Go.

After finishing this setup. we check in the GCP console if the project has been created successfully. Figure 4.5 shows the description of the projects created.

Module 1

Since, we are just focusing on File-based ingestion in our implementation. So, the first step is to see how we can ingest/upload a file. For that, we create a bucket in the google cloud storage. This bucket gives the functionality to upload a file.

Manage resources								
+ CREATE PROJECT DELETE								
Filter tree								
<input type="checkbox"/>	Name	ID	Status	Requests	Errors	Charges	Labels	Actions
<input type="checkbox"/>	▼ No organization	0						⋮
<input type="checkbox"/>	DUP-Cloud	ambient-polymer-228615						⋮
<input type="checkbox"/>	searchengine	searchengine-141010						⋮

FIGURE 4.5: Project description

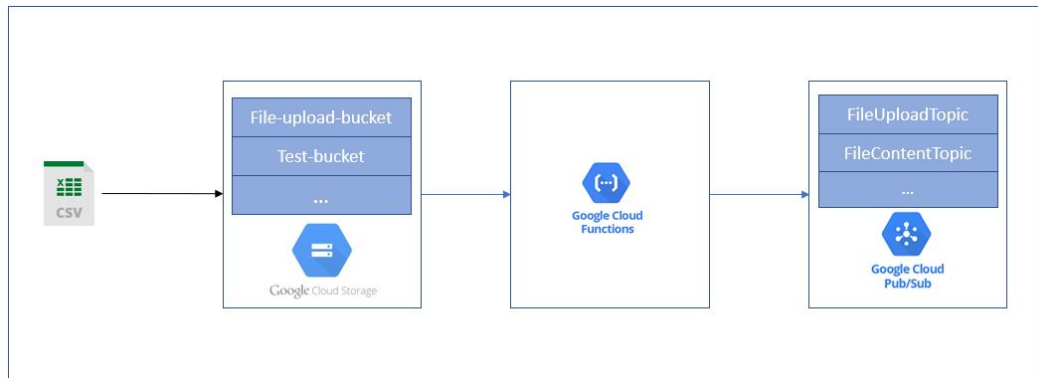


FIGURE 4.6: Module 1 workflow

Following is the command we can use to create a bucket:

cmd [16]: `gsutil mb gs://BUCKET-NAME`

For example: **cmd [16]:** `gsutil mb gs://file-upload-bucket2032`

Now that we have created a bucket, we need deploy a cloud function that is triggered every time a file is uploaded. The reason for this event trigger is to read the meta data of the file and publish it on Pub/Sub. So that the services looking for uploaded files can read the meta data off Pub/Sub and consequently read the file from the bucket. Figure 4.6 shows the architecture of module 1.

We also need to create a Pub/Sub Topic with a subscription that any service can use to read the data off that Topic. Following is the command used to create a topic:

cmd [23]: `gcloud pubsub topics create TopicName`

For example:

cmd [23]: `gcloud pubsub topics create FileUploadTopic`

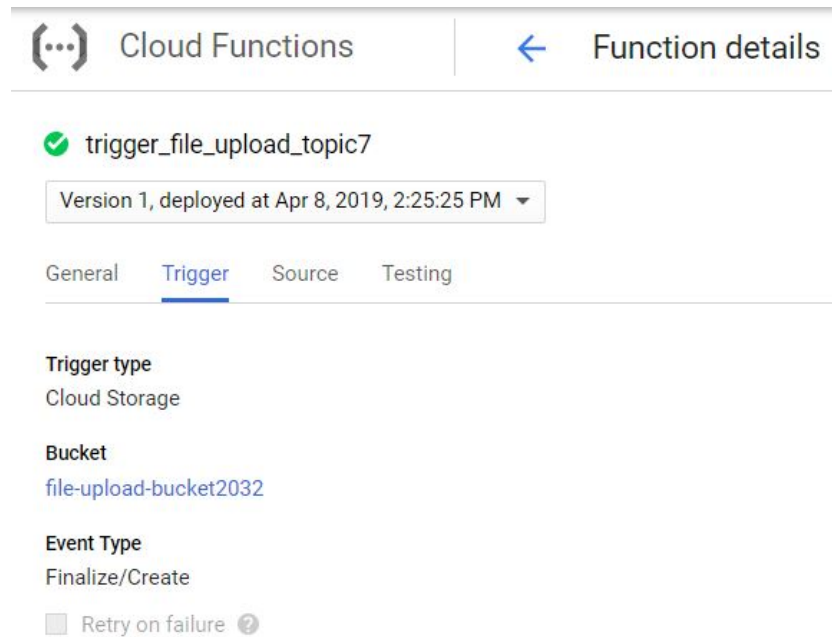


FIGURE 4.7: Cloud function deployment

In the current situation we have the following events that can trigger from cloud storage [16]:

- **Finalize:** This event occurs when an object is uploaded on cloud storage.
- **Delete:** This event occurs when an old version of a file is deleted or it is overwritten .
- **Archive:** It is only used in buckets with versions. Basically, it triggers when a file is deleted or archived.
- **Metadata Update:** This event occurs when the metadata of the file is changed.

Algorithm 1 Cloud function

```

1: procedure TRIGGER-FILE-UPLOAD-TOPIC(data, context)
2:   projectId ← config.get("PROJECTID")
3:   topicName ← config.get("TOPICNAME")
4:   topicPath ← publisher.topicPath(projectId, topicName)
5:   encodedData ← json.dumps(data).encode('utf - 8')
6:   future ← publisher.publish(topicPath, data = encodedData)
7:   return future.result()

```

projects/ambient-polymer-228615/topics/FileUploadTopic
Subscriptions: 1 Snapshots: 0

User labels
None

[Subscriptions](#) [Snapshots](#)

<input type="checkbox"/> Subscription name	Delivery type	Subscription expiration	Labels	Acknowledgement deadline	Project
<input type="checkbox"/> projects/ambient-polymer-228615/subscriptions/FileUploadTopicSubscription	Pull	On	None	10 seconds	ambient-polymer-228615

FIGURE 4.8: Subscription Details

In our case, we need to trigger a function when a file is uploaded. So, we need the Finalize event trigger function. We have used python as the programming language to develop the function. Algorithm 1 shows the procedure which takes the data meta data of the file and publishes on "FileUploadTopic". Figure 4.7 shows the deployed cloud function. We have created a subscription to the topic. Figure 4.8 shows the details of the subscription.

Following is the command to deploy the cloud function:

```
cmd [16]: "gcloud functions deploy function-name --runtime nodejs8 --trigger-resource bucket-name --trigger-event google.storage.object.finalize"
```

For example:

```
cmd [16]: "gcloud functions deploy Trigger-File-Upload-Topic --runtime nodejs8 --trigger-resource file-upload-bucket2032 --trigger-event google.storage.object.finalize"
```

Now, we upload the file to trigger the function. Following is the command to upload a file:

```
cmd [16]: "gsutil cp file-name.csv gs://bucket-name"
```

For example:

```
cmd [16]: "gsutil cp test-file.csv gs://file-upload-bucket2032"
```

Module 2

We have the meta data of the uploaded file into the Pub/Sub topic. Now, we want to read this data and based on it we read the file from cloud storage bucket. Figure 4.9 shows the architecture of Module 2.

Firstly, we need to create a parser service that reads the metadata of the file, reads the file content and publishes the content back to Pub/Sub. We created the parser

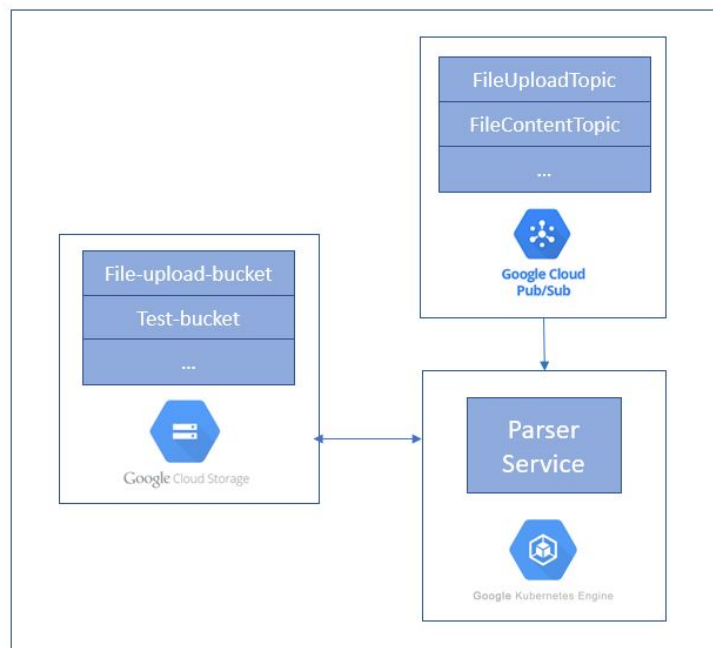


FIGURE 4.9: Module 2 workflow

microservice in java and deployed it on google kubernetes engine (KBE). To deploy the parser service we first need to create a cluster of virtual machines on KBE. A cluster of VM is used to run docker containerized applications. So, following is the command used to create a KBE cluster:

cmd [22]: `gcloud beta container --project "ambient-polymer-228615" clusters create "dup-cluster" --zone "us-central1-a" --username "admin" --cluster-version "1.11.8-gke.6" --machine-type "n1-standard-1" --image-type "COS" --disk-type "pd-standard" --disk-size "100" --scopes "bigquery","cloud-platform","cloud-source-repos","cloud-source-repos-ro","compute-ro","compute-rw","datastore","default","gke-default","logging-write","monitoring","monitoring-write","pubsub","service-control","service-management","sql","sql-admin","storage-full","storage-ro","storage-rw","taskqueue","trace","userinfo-email" --num-nodes "3" --enable-cloud-logging --enable-cloud-monitoring --no-enable-ip-alias"`

In this command [22], we need to set the scope and permissions for the gcp services that our application can use along with the project name, cluster name, zone, machine types.

Once we created the cluster, we can deploy our application on this cluster. Cloud build trigger is used for continuous deployment and integration. Cloud build lets us connect with a github repository and trigger the build every time we commit

```

2  steps:
3    - name: 'gcr.io/cloud-builders/docker'
4      args: ['build', '-t', 'gcr.io/$PROJECT_ID/dupcloudparser-image2032', '.']
5      timeout: 500s
6    - name: 'gcr.io/cloud-builders/docker'
7      args: ['push', 'gcr.io/$PROJECT_ID/dupcloudparser-image2032']
8    - name: 'gcr.io/cloud-builders/kubectl'
9      args: ['apply', '-f', 'deployment.yml']
10     env:
11       - 'CLOUDSDK_COMPUTE_ZONE=us-central1-a'
12       - 'CLOUDSDK_CONTAINER_CLUSTER=dup-cluster'
13     timeout: 500s
14     images: ['gcr.io/$PROJECT_ID/dupcloudparser-image2032']
15
16

```

FIGURE 4.10: cloudbuild.yml code snippet

Build information	
Status	✔ Build successful
Build id	f1916ded-1c3e-42cb-a3a8-0be2bfa61c68
Image	gcr.io/ambient-polymer-228615/dupcloudparser-image2032
Trigger	Push to master branch (dup-parser-trigger)
Source	GitHub sysco-middleware/dup-cloud-parser ↗
Git commit	d2e52ceb fde78479a7e4b56f93a5f3bd9b6d0d2b ↗
Started	June 6, 2019 at 2:54:32 PM UTC-7
Duration	47 sec

Build steps		expand all
✔ gcr.io/cloud-builders/docker	build -t gcr.io/ambient-polymer-228615/dupcloudparser-image2032 .	12 sec ▾
✔ gcr.io/cloud-builders/docker	push gcr.io/ambient-polymer-228615/dupcloudparser-image2032	9 sec ▾
✔ gcr.io/cloud-builders/kubectl	apply -f deployment.yml	7 sec ▾

FIGURE 4.11: Cloud build trigger

changes to that repository. We need a *cloudbuild.yml* file in the application that performs the following steps [17]:

- Creates a docker container image of the application.
- Pushes that image to the container registry.
- Deploys the image to KBE cluster.

Figure 4.10 is the code snippet of the *cloudbuild.yml* file. The steps in the code are performed sequentially.

```

▶ [i] 2019-06-06 14:56:14.381 PDT file name: ": "test-file.csv"
▶ [i] 2019-06-06 14:56:14.381 PDT bucket: ": "file-upload-bucket2032"
▼ [i] 2019-06-06 14:56:14.375 PDT Data: {"bucket": "file-upload-bucket2032", "contentLanguage": "en",
"contentType": "text/csv", "crc32c": "ewxLdA==", "etag":
"CPP+kbvslICEAE=", "generation": "1559858157748083", "id": "file-upload-
bucket2032/test-file.csv/1559858157748083", "kind": "storage#object",
"md5Hash": "YwNkoUSIRjEplgNUPiieg==", "mediaLink":
"https://www.googleapis.com/download/storage/v1/b/file-upload-bucket2032
/o/test-file.csv?generation=1559858157748083&alt=media",
"metageneration": "1", "name": "test-file.csv", "selfLink":
"https://www.googleapis.com/storage/v1/b/file-upload-bucket2032/o/test-
file.csv", "size": "1264", "storageClass": "MULTI_REGIONAL",
"timeCreated": "2019-06-06T12:55:57.747Z", "timeStorageClassUpdated":
"2019-06-06T12:55:57.747Z", "updated": "2019-06-06T12:55:57.747Z"}

```

FIGURE 4.12: Meta data logs

```

{
  "key": "1",
  "data": {
    "id": "3cc740e1-bf07-4064-bb0e-76d3c1550a47",
    "latitude": "5.0815622",
    "longitude": "61.596215",
    "meter_reading": "0.578",
    "meterpoint_id": "707057500025064507",
    "net_station": "53240T1",
    "post_code": "6908",
    "reading_date": "2018-08-28T12:00:00-0700",
    "time_verdi": "1",
    "value": "7"
  },
  "metadata": {
    "file_name": "test-file.csv",
    "bucket_name": "file-upload-bucket2032"
  }
}

```

FIGURE 4.13: Data as Json object

Once we have the application ready with the cloud build file. We create the trigger that is connected with the github repository of this application. Figure 4.11 shows the trigger details.

Now, that our parser service is deployed on kubernetes, it can read the metadata from the Pub/Sub topic. figure 4.12 shows the logs containing the meta data and how the necessary information to read the file is extracted from it such as bucket and file name. The file is then read from cloud storage using this information. The test file we are using is about meter readings of smart meters.

Once the file is read, we take each row and create a json object and push it to the Pub/Sub topic "FileContentTopic". The data consists of attributes such as is, meter_reading, meterpoint_id etc. Figure 4.13 shows the json object we created from the first row of the file content.

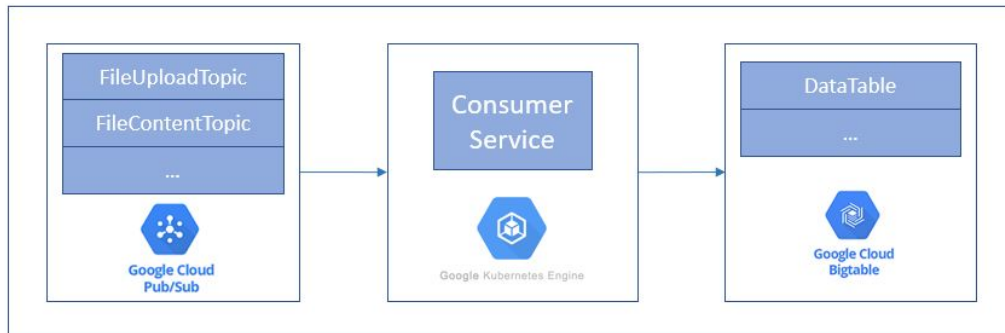


FIGURE 4.14: Module 3 workflow

Module 3

In this module, we read each json object from "FileContentTopic" and save it in google cloud Bigtable. For that, we first need to create an instance of Bigtable. Inside that instance, we create a table where we can store our data. Figure 4.14 shows the workflow of module 3.

The first thing is to create a Bigtable Instance. Following is the command to create a Bigtable instance:

```
cmd [15]: "cbt createinstance filecontenttable FileContentTable filecontenttable-c1 us-central1-a 3 SSD"
```

We can check the instance details in Figure 4.15. Now, we can create a table inside this instance. Following is the command that creates a Bigtable table:

```
cmd [15]: "cbt createtable DataTable"
```

Now, the Bigtable database is ready to receive the data from the consumer service. This microservice is also going to be deployed on Kubernetes engine like Parser. It posts the Json data that it gets from Pub/Sub's "FileContentTopic" to Bigtable's "DataTable". Figure 4.16 shows the monitoring platform that Bigtable provides.

Github

The cloud native application is developed using microservices architecture. These microservices are supposed to be deployed on google cloud platform. The code for these microservices is hosted as github repositories. The repositories are public and added in the citation of this thesis [34, 32, 33].

FileContentTable

Instance ID: filecontenttable Type: Production Storage: SSD

filecontenttable-c1

CPU utilization	Rows	Throughput	System error rate
Average: 0.2%	Read: 0/s	Read: 0 B/s	0%
Hottest node: 0.2%	Write: 0/s	Write: 0 B/s	

Cluster ID	Zone	Nodes	Storage utilization	Tables available
✔ filecontenttable-c1	us-central1-a	3	<div style="width: 100%; height: 10px; background-color: #ccc; border: 1px solid #ccc;"></div> 0 B / 7.5 TB	—/—

FIGURE 4.15: Bigtable instance

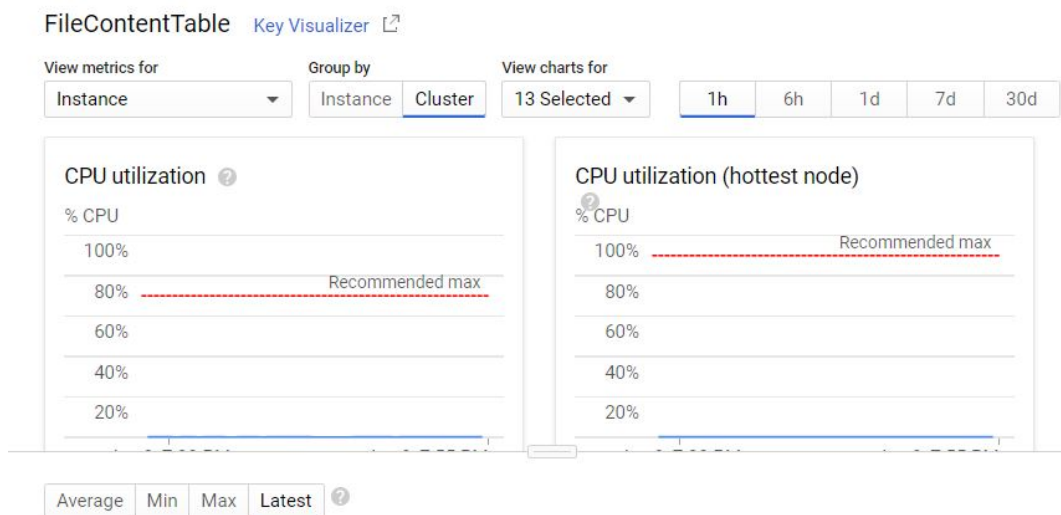


FIGURE 4.16: Bigtable Monitoring

Chapter 5

Experimentation and Evaluation

This chapter consists of the experiments performed and evaluation of the on-premise and cloud native applications. In this thesis, all the experiments are done to achieve two main objectives, Decreased time to market and Extensibility and security. Since our objective is that it takes minimum time to develop and deploy the application with maximum security so that user experience becomes better.

5.1 Setup

The on-premise application is developed and deployed on local servers. The specifications of the on-premise server was initially 4 GB of RAM and 120 GB Hard disk. Later, Due to adding more features, it was changed to 8 GB RAM and 250 GB Hard disk. On the other hand, it's counterpart cloud native application is deployed on google cloud platform using PaaS model. We use Kubernetes engine's standard cluster of 3 nodes to deploy services. Each node in a standard cluster has 4 GB of RAM. This can be scaled up to a high performance cluster when required. We ran the experiments first on the on-premise application using metrics such as provision and scaling, continuous integration and deployment, testing, zero downtime, extensibility, security, resiliency, agility, observability. We noted down the results from the first experiments and performed the same experiments on the cloud native application. We will put all these results ahead in the report and talk about the advantages and disadvantages of each architecture based on our evaluation.

5.2 Decreased time to market

In this section, we will list the experiments and their results, performed to make the delivery time minimum to the user. There are some specific cases which affect the

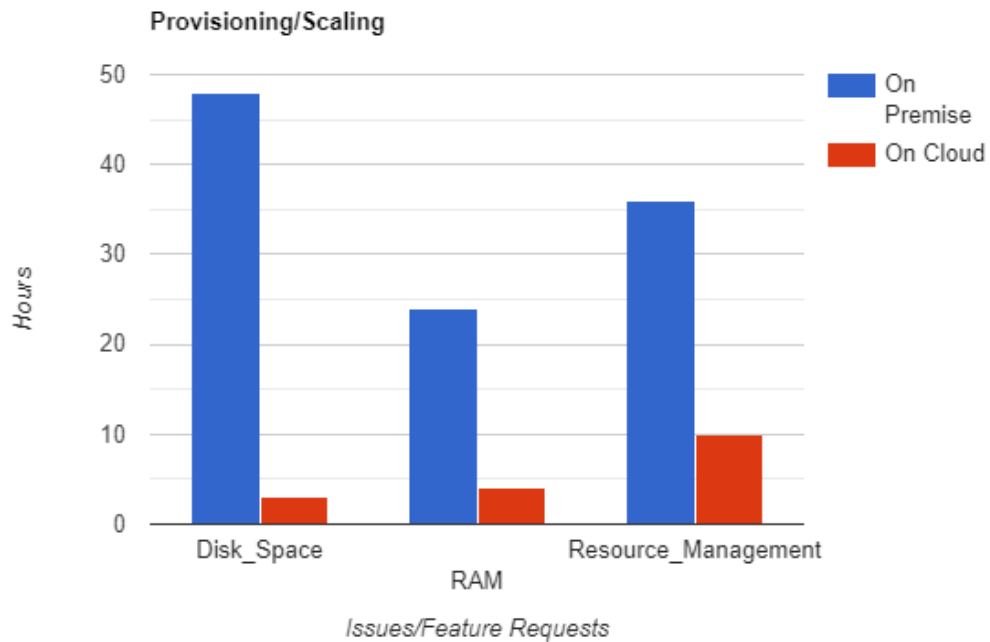


FIGURE 5.1: Comparing Provision and Scaling

time to market. We will evaluate all of these cases one by one.

5.2.1 Provisioning and Scaling

Most applications, after a certain time, need to be scaled up or down depending upon the usage and memory/resource management. For the applications that are on-premise, it is a very long and cumbersome task to scale up. That is mainly because RAM and Hard disk have to be physically put in the servers and get allocated to the required application. The developers need to solve the software issues that come with it.

The cloud native application, on the other hand, is easily scalable. RAM, Hard disk, Virtual machines can be easily increased on demand. We only have to use and pay for the services we are using at a given time, no more no less.

We ran some experiments to scale up the on-premise application and compared it with the cloud-native application. First experiment is to increase the Disk space, which on-premise application took 48 hours. We changed it from 120 GB to 250 GB. On the other hand, Disk space on a cluster is increased just on demand. We need

to take the cluster down, go through the instructions for increased resources, and increase it. It took 3 hours in total for the whole process. For RAM it took 24 hours on-premise server and took 4 hours on cloud. Since, we only need to change the cluster from standard to CPU intensive cluster. Managing resources for on-premise application was a big task due to the teams working on it. It took 36 hours and on cloud, it took 10 hours because of it being one use case of the entire application. The difference of timings are huge in comparison of on-premise and cloud-native, that is because cloud is managed by providers and you just need to demand. For on-premise, you need to physically add the modules and manage them in software. Figure 5.1 shows these experiments in a plot.

5.2.2 Continuous Integration

Continuous deployment and integration is a fundamental part of application development today due to the DevOps culture. Developers need to continuously merge all branches of code a couple of times daily in order to avoid bugs and conflicts later. Also, we need an automated setup for deploying changes into production. So, the developers can be totally focused on the job in hand rather than worrying about the deployment strategy.

With on-premise applications the process of integration and deployment is easier said than done. Developers need to create CI/CD pipelines to deploy changes. Automation is not possible here, developers manually need to setup a build tool such as Jenkins. In other words, developers have to worry about the Operational work such as deployment strategies on top of developing the application.

On the other hand, the cloud native application consists of automated operational tasks. Developers do not need to setup any build tools or create pipelines. GCP provides us with cloud build triggers that we discussed in chapter 4. Cloud build provides the resource to deploy the changes into production by just committing your code into the master branch of your Github repository. It lets you connect your microservice running on kubernetes engine to Github. An automated pipeline is generated and the developers only focus on the development part rather than the operational tasks.

To compare the performance of the two applications in terms of CI/CD, we ran some experiments. These experiments consist of keeping the deployment times of each microservice and compare which architecture gives better results. Figure 5.2 shows the deployment times for multiple attempts to deploy the function that triggers when a file is uploaded. The on-premise times were 4,5,4.5,5.6,4,5.2 (hours) and on cloud they were 0.1,0.2,0.1,0.3,0.2,0.3 (hours). Figure 5.3 shows the deployment times for multiple attempts to deploy the parser service. The on-premise times were

5.4.2,4.1,5.2,6.1,5.1 (hours) and on cloud they were 0.3,0.2,0.1,0.4,0.3,0.4 (hours). Figure 5.4 shows the deployment times for multiple attempts to deploy the consumer service. The on-premise times were 3,4,4.5,5.3,4.5,4.7 (hours) and on cloud they were 0.2,0.2,0.1,0.1,0.3,0.2 (hours). Figure 5.5 shows the average time of each service and compares the two approaches. The average time for deploying cloud function on-premise was 4.7 hours and on cloud was 0.3 hours. For Parser service, it was 5 hours on-premise and 0.4 hours on cloud. For consumer service, it was 4.3 hours on-premise and 0.3 hours on cloud.

You can see the huge differences in deployment times of on-premise and cloud native applications. That is because cloud native approach provides us with automation around DevOps.

Zero Downtime

Another thing to consider when talking about CI/CD is the ability to deploy changes in the application without any down time. The on-premise application does not support zero downtime deployment because the application has to be taken down of the production server in order to change it and deploy it back. On the other hand, the cloud native application provides with the functionality to deploy new versions of the application while the old version is still running. Doing this, the customers or other integration services using your application do not experience down time.

5.3 Extensibility and Security

5.3.1 Extensibility

This is an important factor to analyse when evaluating a cloud native application. Extensibility is one of the main differences between cloud native and monolithic applications. Table 5.1 shows the percentages of different features related to extensibility for on-premise and cloud. We can clearly see that for loose coupling, modularization and extensibility on premise numbers are not above required. But using GCP, microservices are 100% loosely coupled, modularized and are extensible.

5.3.2 Security

The developers do not have to implement security. Google is making GCP the most secure cloud platform. Security at Infrastructure, resource, API and identity level

Features	Sub level	On-premise status(%)	Required status(%)	GCP level(%)
Loose coupling	code (service)	70	min 90	100
	Infrastructure	20	min 90	100
Modularization	code (service)	80	min 90	100
	Infrastructure	10	min 90	100
Extensibility	code (service)	70	min 90	100
	Infrastructure	70	min 90	100

TABLE 5.1: Extensibility

should be high but also the effort for implementation should be low. The cloud provider GCP, provides us infrastructure, hardware and software, so we do not need it handle security there which makes it low in Figure 5.2. Also, each resource such as kubernetes engine, Pub/Sub are provided by GCP, so we do not need to implement security. At API level on the other hand, GCP provides us with client API which we use to access the resources. But, still we need to download a security key and include it in the program, which makes it medium level effort in Figure 5.2.

Features	Security level On-premise	Effort On-premise	GCP level	GCP effort
Infrastructure level security	High	High	High	Low
Resource level security	Medium	High	High	Low
API level security	High	Medium	High	Medium
Identity based	High	High	High	Low

TABLE 5.2: Security

5.3.3 Agility

GCP is very impressive agility wise. Its easy to understand services with proper help from its client libraries makes it easy to migrate to. Table 5.3 shows the difference of agility on-premise and on GCP. Development takes some effort because you need to redesign the application according to GCP services. So, it is medium in Table 5.3. Testing for GCP is high because it provides us with proper logging and monitoring mechanism, using which we can test if it works as intended or not. Also, deployment is high in terms of agility because of automation around DevOps

through cloud build trigger discussed in the previous section. On the other hand, on-premise application has low development and testing agility while still having better deployment.

Features	On-premise level	GCP level
Development	Low	Medium
Testing	Low	High
Deployment (Production)	Medium	High

TABLE 5.3: Agility

5.3.4 Observability

GCP provides you with services to monitor the application. It gives you a platform to show logs, tracing and report errors. Table 5.4 shows difference between observability features. GCP provides with Stackdriver Trace, which makes tracing bottlenecks into a service easy and thus it is high in Figure 5.4. GCP also provides with Logging and monitoring service such as Stackdriver Logging. It gives all the logs from the whole application in one place making it easier to read the logs and which is why it is high in Figure 5.4. There is no error reporting on-premise but GCP provides us with Stackdriver Error Reporting, using which errors are isolated and effected services are shown.

Features	On-premise level	Expected level	GCP level
Tracing	Medium	High	High
Logging	Low	High	High
Error Reporting	NA	High	High

TABLE 5.4: Observability

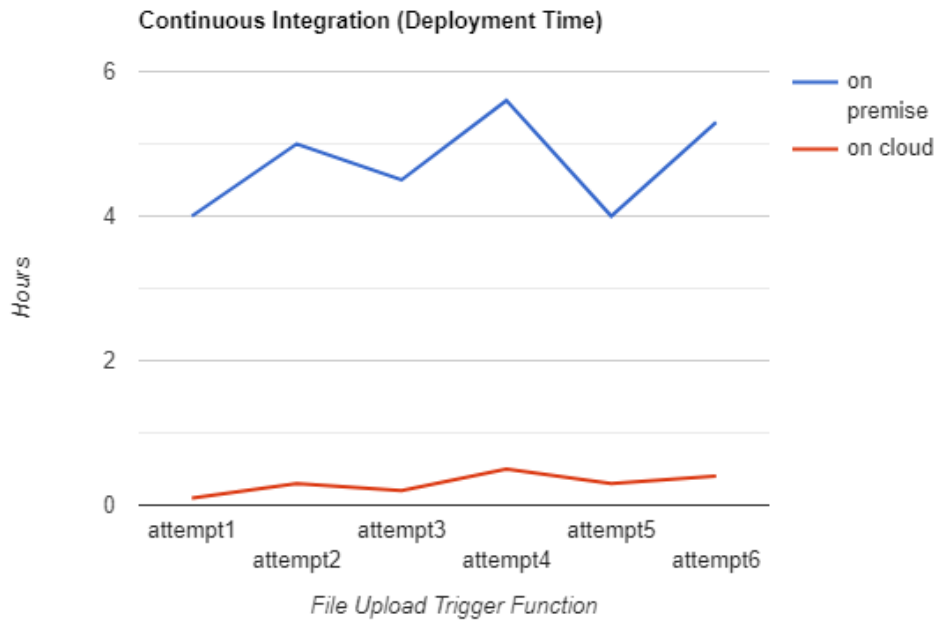


FIGURE 5.2: Deploying Trigger Function

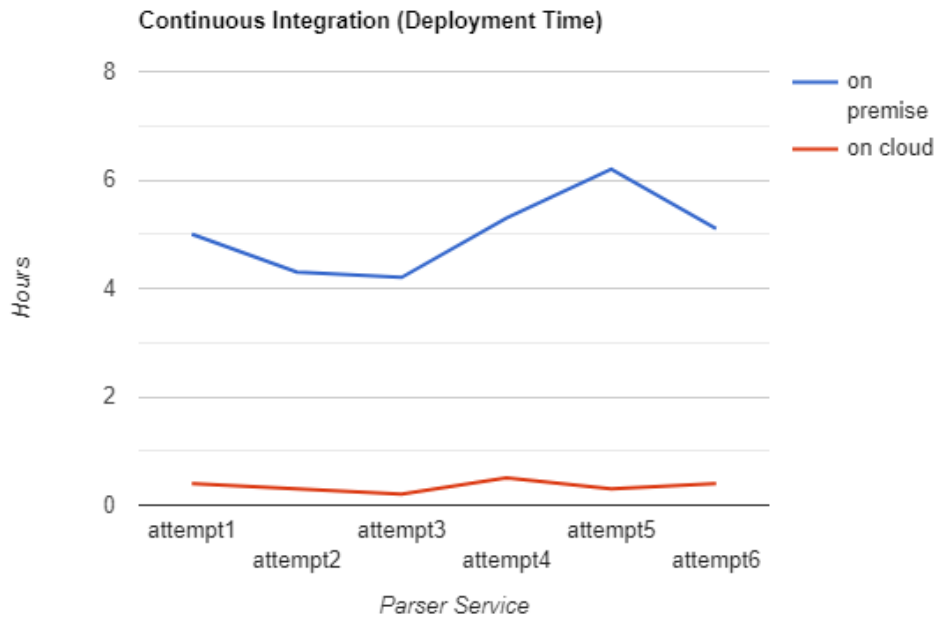


FIGURE 5.3: Deploying Parser Service

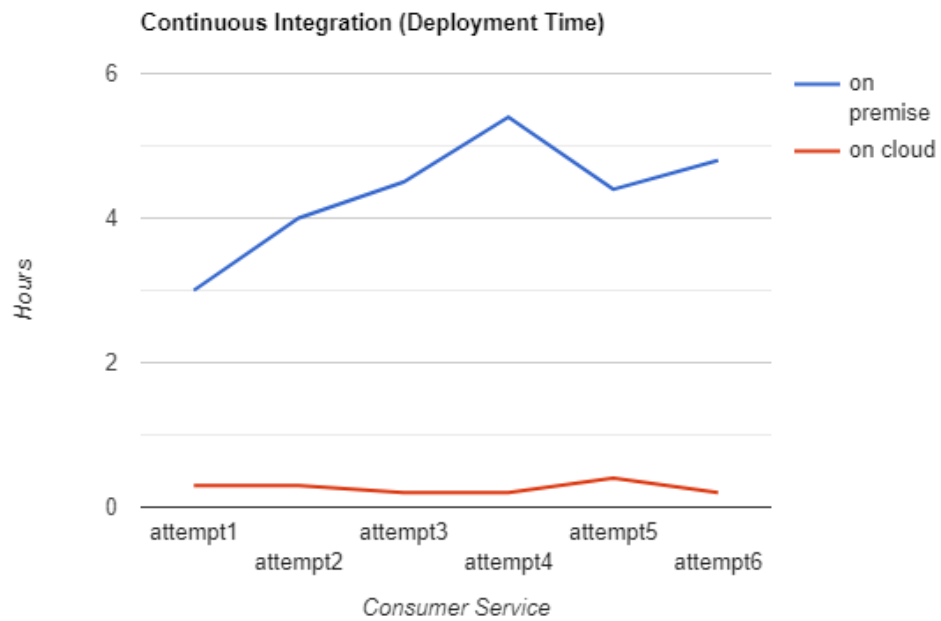


FIGURE 5.4: Deploying Consumer Service

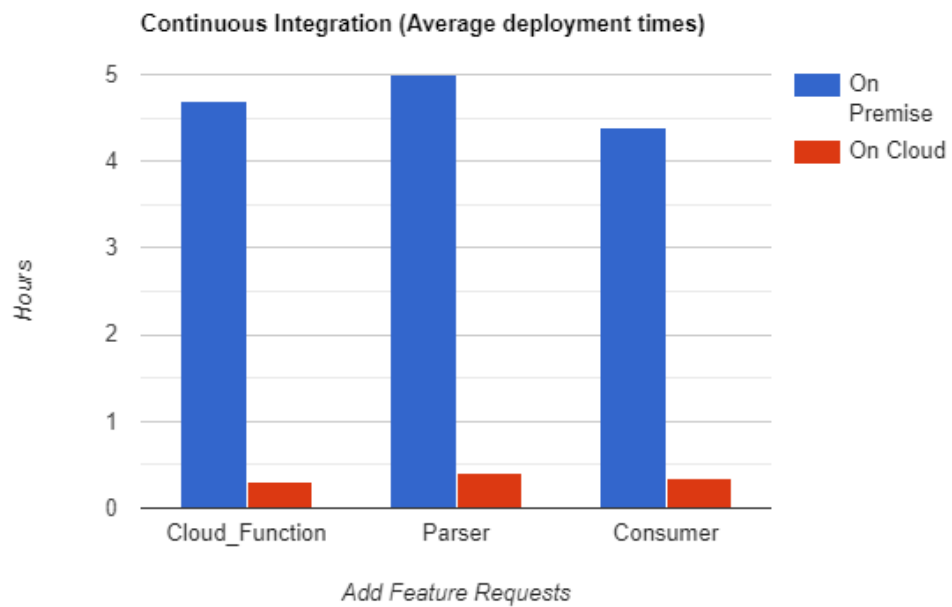


FIGURE 5.5: Average for Adding Feature

Chapter 6

Conclusion and Future Work

Cloud computing has made a huge impact on application development today. Applications that needed big noisy, expensive and hard to manage servers do not need them anymore. Cloud provides them the whole Infrastructure as a Service, Platform as a Service, or Software as a Service, making it easy to develop and deploy the applications unlike on-premise.

Migrating to cloud comes with its challenges. The companies either move their applications on cloud or develop cloud native applications from scratch to exploit cloud potential to its maximum. To do that, they need to follow some characteristics and cloud native properties to create an application that is best suited for cloud.

We saw that to create a cloud native application from an on-premise monolith, the design for the application can be same throughout different cloud providers, since they all provide with alternate services. But, the developers need to develop the application according to each cloud. Each cloud platform e.g. Google cloud platform gives it services with a specified set of client libraries to access those services. To program the application for some other cloud provider, we will have to use different client libraries to access their services.

In this thesis, we also created a cloud native application from an on-premise monolith. First, we designed the architecture based on google cloud platform services. Then, we developed the application in a cloud native way following all the properties stated in the background chapter. Finally, we run some experiments to analyse and evaluate the performance of the two architectures. The main differences we looked into are related to DevOps culture, ease of implementation, continuous development and integration, extensibility and scalability, security and decreased time to market. The experiments have shown some drastic results. The findings are all hugely in favor of cloud native architecture. Since, it is easily extensible, microservices are scalable and can be added for additional functionality, automated at deployment ends, continuous deployment takes minutes in response to hours

on-premise. Provision and scaling for computing resources is easier since it can be added on demand rather than having to physically manage them on on-premise servers.

In the future, there is scope for improvement and additional functionality on this report. Firstly, we can add machine learning algorithms in our application. since we worked on just File based ingestion usecase in this thesis, we can develop the whole DUP on-premise application in a cloud native architecture. Support for multi-cloud can be added in the future to make it easier to migrate to other cloud providers, the possibilities are limitless with cloud.

Bibliography

- [1] M. Ahronovitz, D. Amrhein, and P. Anderson. “Cloud Computing Use Cases - A white paper”. In: (Sept. 2010), pp. 20–21. URL: <http://cloudusecases.org>.
- [2] Tareq Ahmed Ali Al-Maamari. “Aspects of Event-Driven Cloud-Native Application Development”. MA thesis. Universitätsstraße 38 D–70569 Stuttgart: University of Stuttgart, 2016.
- [3] Mirna Alaisami. “Cloud-native Applications: Authoring and Evaluation of Two Deployment Patterns”. MA thesis. Universitätsstraße 38 D–70569 Stuttgart: University of Stuttgart, 2018.
- [4] Michael Armbrust et al. “Above the Clouds: A Berkeley View of Cloud Computing”. In: 1 (Feb. 2009), pp. 1–9. URL: <http://citeseerx.ist.psu.edu/viewdoc/citations?doi=10.1.1.149.7163>.
- [5] Sysco AS. *Digital Playground (a part of SYSCO Energy Cloud)*. URL: <https://sysco.no/smidig-overgang-till-2gen-ams-hur-kan-vi-bli-battre-an-norge/>.
- [6] A. B. Bondi. “Characteristics of Scalability and Their Impact on Performance”. In: *Proceedings of the 2Nd International Workshop on Software and Performance* (2014), 195–203. URL: <http://doi.acm.org/10.1145/350391.350432>.
- [7] K. CHANDRASEKARAN. *Essentials of CLOUD COMPUTING*. 6000 Broken Sound Parkway NW, Suite 300: Taylor & Francis Group, LLC, 2015.
- [8] Fay Chang et al. “Bigtable: A distributed storage system for structured data”. In: *In proceeding of the 7th conference on USENIX symposium on operating systems design and implementation - VOLUME 7*. 2006, pp. 205–218.
- [9] S. Daya, N. Van Duy, and C. Ferreira K. Eati. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. Reading, Massachusetts: IBM Redbooks, 2016.
- [10] Edureka. “Google Cloud Platform Fundamentals”. In: (2018). URL: <https://www.edureka.co/blog/what-is-google-cloud-platform/>.
- [11] C. Fehling et al. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Publishing Company, Incorporated, 2014, pp. 21–23.

- [12] A. FELDMAN and C. HENRY. “Best Practices for Developing Cloud-Native Applications and Microservice Architectures”. In: *The Net Stack* (Feb. 2015), p. 23. URL: <http://thenewstack.io/best-practices-for-developing-cloud-native-applications-and-microservice-architectures>.
- [13] Dennis Gannon, Roger Barga, and Neel Sundaresan. “Cloud-Native Applications”. In: *IEEE Cloud Computing* 4 (Sept. 2017), pp. 16–21. DOI: [10.1109/MCC.2017.4250939](https://doi.org/10.1109/MCC.2017.4250939).
- [14] Justin Garison and Kris Nova. *Cloud Native Infrastructure*. Ed. by Virginia Wilson and Nikki McDonald. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O’Reilly Media, Inc., Nov. 2017.
- [15] Google. “Bigtable Instance creation”. In: (2016). URL: <https://cloud.google.com/bigtable/docs/creating-instance>.
- [16] Google. “Cloud storage Tutorial”. In: (2016). URL: <https://cloud.google.com/functions/docs/tutorials/storage>.
- [17] Google. “Creating Build configuration”. In: (2016). URL: <https://cloud.google.com/cloud-build/docs/configuring-builds/create-basic-configuration>.
- [18] Google. “Google Cloud Functions. Google Inc. 2016”. In: (2016), p. 48. URL: <https://cloud.google.com/functions>.
- [19] Google. “Google Cloud Storage. Google Inc. 2016”. In: (2016). URL: <https://cloud.google.com/storage/>.
- [20] Google. “Google Kubernetes Engine. Google Inc. 2016”. In: (2016). URL: <https://cloud.google.com/kubernetes-engine/>.
- [21] Google. “Google Pub/Sub. Google Inc. 2016”. In: (2016). URL: <https://cloud.google.com/pubsub/>.
- [22] Google. “Kubereneets Cluster Creation”. In: (2016). URL: <https://cloud.google.com/kubernetes-engine/docs/how-to/creating-a-cluster>.
- [23] Google. “Pub/Sub Topic Management”. In: (2016). URL: <https://cloud.google.com/pubsub/docs/admin#pubsub-create-topic-cli>.
- [24] Heroku. *The twelve-factor app methodology for building robust SaaS*. 2017. URL: <http://12factor.net/>.
- [25] Nane Kratzke. “A Brief History of Cloud Application Architectures”. In: *Lübeck University of Applied Sciences* 1 (Aug. 2018), pp. 10–21.
- [26] J. Lewis and M. Fowler. “Microservices , a definition of this new architectural term”. In: (2014), p. 26. URL: <http://martinfowler.com/articles/microservices.html>.

- [27] Peter Mell and Timothy Grance. "The NIST Definition of Cloud Computing". In: *NIST Special Publication 800-145* (Sept. 2011), pp. 20–21. URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>.
- [28] Goran Novkovic. *Manufacturing in the Cloud. Part II: 5 Characteristics of Cloud Computing*. 2019. URL: <http://blog.mesa.org/2017/08/manufacturing-in-cloud-part-ii-5.htmls>.
- [29] C. Richardson. *Microservice architecture patterns and best practices*. 2018. URL: <http://microservices.io>.
- [30] Amazon Web Services. *Benefits of Microservices*. URL: <https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/benefits-of-microservices.html>.
- [31] Amazon Web Services. *Challenges of Microservices*. URL: <https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/challenges-of-microservices.html>.
- [32] Hafiz Rayaan Shahid. *DUP Cloud consumer Microservice*. <https://github.com/rayaanshahid/dup-cloud-consumer>. 2019.
- [33] Hafiz Rayaan Shahid. *DUP Cloud Function Microservice*. <https://github.com/rayaanshahid/dup-cloud-function>. 2019.
- [34] Hafiz Rayaan Shahid. *DUP Cloud Parser Microservice*. <https://github.com/rayaanshahid/dup-cloud-parser>. 2019.
- [35] Edwin Shouten. "Rapid elasticity and the cloud". In: (Feb. 2012), p. 23. URL: <http://www.thoughtsoncloud.com/2012/09/rapid-elasticity-and-the-cloud/>.
- [36] Rishabh Software. *Cloud Computing Deployment And Service Models*. 2019. URL: <https://www.rishabhsoft.com/blog/basics-of-cloud-computing-deployment-and-service-models>.
- [37] Luis M. Vaquero et al. "A break in the clouds: Towards a cloud definition". In: *ACM SIGCOMM Computer Communication Review* (2009), pp. 50–55. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.176.6131&rank=3>.