Cloud deployment research

Table of contents

[Table of contents 2](#_Toc138015133)

[Introduction 3](#_Toc138015134)

[Main question 3](#_Toc138015135)

[Research methods 3](#_Toc138015136)

[Sub questions 3](#_Toc138015137)

[What are the advantages of cloud computing? 3](#_Toc138015138)

[What are our requirements? 4](#_Toc138015139)

[Which cloud provider is most suitable taking the application's requirements into account? 5](#_Toc138015140)

[Which services of each cloud provider are of interest to us? 6](#_Toc138015141)

[Comparing the service providers and services that are of interest, which provide the most value to us? 8](#_Toc138015142)

[Service provider 8](#_Toc138015143)

[Service(s) 9](#_Toc138015144)

[How do we upload the application to the cloud service? 10](#_Toc138015145)

[Conclusion 13](#_Toc138015146)

[Recommendations 14](#_Toc138015147)

[Sources 15](#_Toc138015148)

# Introduction

Our application is currently still in a 'development phase', where the backend and frontend are in different repositories. To demonstrate the full application we create a docker image of our own parts so that another person can demonstrate their part working fully. If we want to deliver our application as an actual piece of enterprise software, we need to combine everything and deploy it.

As for scope, the client has provided us with an Azure environment to use for hosting purposes. They want to be able to run the application on-premise, but for our project they requested we deploy it in their cloud environment. This brings a lot of questions, like what services within Azure can help us with this? How can we handle scalability while it's deployed? How do we implement this without making the application dependent on cloud services? These are all things to take into consideration when deploying the application to the cloud environment.

# Main question

How do we upload our application to a cloud service-provider?

# Research methods

**(Library) Literature study**Because the domain of cloud services is very big, it’s good that we first find general information about it before we start researching. This gives us a good base for the rest of the research. Any question that revolves around a general cloud-based concept, we can first use literature study to get some ground to base the rest of the research on.

**(Field) Document analysis**  
Some of our research questions revolve around figuring out specifics about certain technologies. For example, the question about which cloud services apply to our case requires us to look into the specifics of how certain services work. This means we have to look at the documentation provided by the service provider about their services. By doing this we get a good understanding of how the services or technologies work which allows us to make a good judgement on what to use.

**(Workshop) Prototyping**Once we have all the specifics figured out we can try applying our knowledge by making a small prototype and uploading it to the cloud provider. This has to be a very small prototype that doesn’t take a lot of compute power, as to ensure we don’t make too much unintended costs.

# Sub questions

## What are the advantages of cloud computing?

Cloud computing is a broad term used to describe various different IT resources that run over the internet (Amazon, sd). This allows you to run multiple parts of your application, infrastructure or other service on the cloud. All the infrastructure related to hosting these services is handled by the cloud provider, while you only get charged for the computing power you use.

**Scalability**

With cloud services you can let the cloud provider handle scalability for you without sacrificing performance or availability. Most of the time this means it increases the number of servers, storage or other resources on-demand (Simplilearn, 2023). This allows your application to virtually scale infinitely without any effort from the user’s side. This also goes the other way around. Whenever the amount of resources is not necessary anymore, it scales them down again saving you costs.

Having this be handled automatically relieves time and stress from the user’s team which allows them to spend more time on other tasks.

**Cost**

Most cloud providers only charge you for the services or resources you use (Google, sd). In comparison to hosting everything yourself, this can be very cost effective. You don’t have to pay for electricity or maintenance costs. You also don’t pay for any computing power not used, as you only pay for the resources that are being used.

A disadvantage of this is that you have to be careful with what services you use, or what code is being run as it can be easy to over-spend. Infinite loops in your code for example can unnecessarily stir up costs (Greene, 2023).

**Flexibility**

Users can scale, customize and access cloud services from anywhere in the world with only an internet connection (IBM, sd). This makes it incredibly easy to adjust your service based on any needs that arise. If the need arises you can instantly scale your database up, for example, without any infrastructure adjustments.

**Reliability & Availability**

Cloud services in general are very reliable. Big providers usually have thousands of servers everywhere in the world, so if one crashes your traffic can get redirected to another one quickly (Quali, 2023). They also perform frequent backups to restore lost data, or you can set up your own backup routine.

## What are our requirements?

During sprint meetings, we have concluded that the following requirements are set regarding cloud providers & hosting:

* No vendor lock in. The company wants the system to be also deployable on-premise, which means that tight coupling with a cloud platform might give issues down the line.
* Costs should be kept to a minimum.
* Support to scale services independently.

However, after talks with our stakeholders they also added additional requirements regarding the software architecture. Even though they want to keep costs as low as possible, a microservice type software architecture does cost more compared to a standard architecture. Within a microservice architecture, multiple services are scaled independently. Each service often uses their own database and configuration, health monitoring.

During development the choice is made to introduce Docker to streamline the development process and enable faster deployment. It provides a lightweight and portable environment for software development by packaging the applications and dependencies into a container. This is important for us to know, as most cloud providers also offer separate services for containerized applications.

We are also using a document-based database, as the data that is generated in the project is JSON-based. The JSON will directly be stored in this database, and retrieved to build forms or surveys that will be displayed to the end-user.

## Which cloud provider is most suitable taking the application's requirements into account?

Applications rely heavily on cloud computing, which means that the choice of a suitable cloud provider becomes more important. Selecting the right cloud provider that suits the application (and company) needs involves various factors, such as:

* Taking the requirements of the company into account
* Taking the requirements of the application into account
* Cost-efficiency and management
* Learning curve for developers
* Scalability of services
* Security of services

By gaining an understanding of the available options, and comparing them to the application and company needs, an informed decision can be made. By answering this research question, we want to explore which cloud providers and services are available while taking the requirements of the project into account. (Microsoft, sd)

**Which are the top cloud providers as of 2022/2023?**

There are several top cloud providers available that offer various services for software development. To get an idea of which providers are chosen the most, some literature study and community research is done to be able to create a list of options. This list will be used to further research into which services are useful for the project and allow us to make an informed decision.

Based on various resources such as blogs and community websites, the following cloud providers are used by developers and hold the biggest market share:

1. Amazon Web Services (AWS)
2. Microsoft Azure
3. Google Cloud Platform (GCP)
4. Alibaba Cloud
5. Oracle Cloud Infrastructure (OCI)
6. IBM Cloud
7. DigitalOcean
8. Linode

To not overcomplicate this research, we have chosen to further research the top 3 of these cloud providers based on their popularity (market share and users), which are AWS, Azure and GCP. (c-charpcorner, 2023)

**Top 3 Cloud providers**

* Amazon Web Services (AWS)

AWS is the leading cloud provider in the world and known for its comprehensive suite of services and infrastructure. They offer scalable, reliable, and secure services with strong integration capabilities. Because of the broad range of cloud solutions such as computing, storage, databases, AI, and serverless computing it is popular among businesses of all sizes.

* Microsoft Azure

Azure is owned by Microsoft and is a robust platform that combines infrastructure, platform services and SaaS. It integrates easily with other products offered by Microsoft, and has a wide range of tooling, hybrid cloud capabilities, AI, and strong customer support.

* Google Cloud Platform (GCP)

GCP is owned by Google and is known for its scalability and performance. They offer a wide range of services, advanced analytics and AI / machine learning capabilities. They also have strong support for containerization and Kubernetes. GCP is appealing to organizations that focus on data analytics, AI and machine learning.

By these descriptions, you can conclude that most of these cloud providers seem to generally provide the same kind of features. For this reason, we looked at the project's requirements so we can look for services or features that are of interest to us.

## Which services of each cloud provider are of interest to us?

Amazon Web Services (AWS)

|  |  |
| --- | --- |
| Data Storage | * Amazon DynamoDB * Amazon DocumentDB |
| Containerization & Orchestration | * Amazon Elastic Container Registy (ECR) * Amazon Elastic Container Service (ECS) * Amazon Elastic Kubernetes Service (EKS) * AWS Fargate * AWS App2Container |
| Messaging | * Amazon Simple Queue Service * Amazon MQ |
| Configuration & Management | * AWS Secrets Manager |
| Monitoring & Service Health | * Amazon CloudWatch * AWS X-Ray * AWS CloudTrial * Amazon CloudWatch Container Insights |
| Cost Management | * AWS Budgets * AWS Cost Explorer |
| Standard Hosting | * Amazon EC2 / Auto Scaling * AWS App Runner * AWS Elastic Beanstalk |

Microsoft Azure

|  |  |
| --- | --- |
| Data Storage | * Azure Cosmos DB * Azure Cosmos DB (MongoDB API) * Azure Database for MongoDB |
| Containerization & Orchestration | * Azure Container Registry (ACR) * Azure Container Instances (ACI) * Azure Kubernetes Service (AKS) * Azure Migrate * Azure Container Apps |
| Messaging | * Azure Queue Storage * Azure Service Bus * Azure Event Grid |
| Configuration & Management | * Azure Key Vault |
| Monitoring & Service Health | * Azure Monitor * Azure Application Insights * Azure Monitor for Containers * Azure Diagnostics |
| Cost Management | * Azure Cost Management + Billing |
| Standard Hosting | * Azure Virtual Machines * Azure App Services |

Google Cloud Platform (GCP)

|  |  |
| --- | --- |
| Data Storage | * Google Cloud Bigtable * Google Cloud Firestore |
| Containerization & Orchestration | * Google Container Registry * Google Kubernetes Engine (GKE) * Google Cloud Run * Google Cloud Migrate for Anthos |
| Messaging | * Google Cloud Pub/Sub |
| Configuration & Management | * Google Cloud Secret Manager |
| Monitoring & Service Health | * Google Cloud Monitoring * Google Cloud Trace * Google Cloud Audit Logging * Google Cloud Operations |
| Cost Management | * Google Cloud Cost Management |
| Standard Hosting | * Google Compute Engine * Google App Engine |

(Google, 2023) & (Amazon Microsoft and Google cloud infrastructure market, 2022)

To reduce vendor lock-in, we want to minimize relying on cloud service to handle things for us.

**Messaging** and **configuration** can be done within the application itself without much extra trouble. Having these be hosted by the cloud provider would increase the reliability on the cloud and make it more difficult to transfer everything to an on-premise application, or to move it to a different provider.

**Cost management** and **monitoring** is a nice addition, but not necessary to have. This is also outside our project’s scope, so we will not be looking at it unless the client specifically asks for it.

While **containerization & orchestration** can also be handled without a cloud service using Kubernetes for example (Red Hat, 2022), it gives much more benefit in the cloud. Due to time constraints it is also not possible for us to set up a Kubernetes cluster, so it would make more sense to deploy the application in a container server that can handle scaling for us. Because our application consists of multiple different services, and in the future it’s expected there will be multiple new services added, we can deploy our application in these container services.

**Data storage** is easy to replace within an application as most of the time it’s only an address and credentials you have to fill in. If the data storage would move to a different provider you would only have to change this configuration in the application. Hosting the database in the cloud won’t worsen vendor-lock by much. It also makes it easier to scale when necessary as most cloud services do this automatically.

If we have everything managed in our application like container orchestration, then there is no need to deploy the app to a specialized service (Mohamed, 2019). Instead, we can deploy the app as a whole to a **standard hosting** service. In our case we don’t have this managed yet, however. Due to time constraints, we don’t have everything set up to just deploy the application in full.

This leaves data storage and containerization & orchestration as valid types of cloud services to deploy our application to.

## Comparing the service providers and services that are of interest, which provide the most value to us?

### Service provider

The service providers itself don’t differ a lot from each other. It is especially not very relevant as the cloud deployment we are planning will not be used in production, since the applications will be run on-premise. Realistically, any cloud service provider works for us. Still, we decided on using Azure for a number of reasons.

Azure’s data storage service, CosmosDB, has the support for using the MongoDB API. This makes it easy for us, or the next group, to switch out the database solution later for a locally hosted one. The client wants the application to be runnable on-premise and not be cloud dependent, so if we build everything on cloud based storage systems then it will be more difficult to deploy the application on a local environment later. Since CosmosDB has the option to use the MongoDB API, we would only have to change some of the configuration in the application later down the line. The code itself should be relatively untouched.

Azure is designed to be a very safe cloud provider. It is designed on the Security Development Lifecycle (Saviant Consulting, sd), along with applying a lot of industry-wide security standards, which makes it an incredibly safe piece of software. Security is one of, if not the most important aspect of cloud deployments. A safe place to host your application is then the top priority.

The client provided an Azure resource group for us to use and deploy parts of the application to. The client is used to Azure for cloud deployment and wants us to work with it as they have to continue development after us (or after the entire application is developed). This is the most important reason to pick Azure, as most of the advantages of Azure also apply to the other cloud providers.

### Service(s)

For **data storage**, we will use two different data storage services. Azure offers multiple types of data storage like MySQL, PostgreSQL or their own CosmosDB. Additionally, companies like MongoDB offer their services on the Azure marketplace to add to your cloud environment. This gives you the benefits of their database services, while keeping the application in the cloud. The advantages of cloud services then also apply to these non-azure services like horizontal scaling and availability (MongoDB). Some database types are also integrated within CosmosDB directly.

Because our application uses JSON to store everything we need a NoSQL-based database. To minimize vendor lock-in while keeping scalability we can use one of the NoSQL options within CosmosDB that utilizes an external service. For example, we can’t use ‘CosmosDB for NoSQL’ as it would rely too much on CosmosDB directly. Instead, we will be using ‘CosmosDB for MongoDB’. With this choice the application will be using the MongoDB API, which allows us or a future group to easily switch away from cloud services when necessary. On top of that, the application still benefits from the advantages of CosmosDB.

As previously mentioned, we will be using a **containerization & orchestration** service to host the application and orchestrate the containers. There are a number of them to choose with different functionalities, but these are the most important ones:

* **Azure Kubernetes Service (AKS)**: A Kubernetes service that simplifies the deployment and management of Kubernetes Clusters.
* **Container apps:** Service that allows you to run your containerized applications without having to worry about orchestration or infrastructure.
* **Container instance:** This just hosts your application without you having to worry about management or orchestration. It’s just hosting your container in the cloud. The difference between this and container apps, is that the container apps allows you to upload multiple containers as one app.
* **Container registry:** A private registry service to store and manage docker images or other artifacts from. You can upload your docker images to this private registry to then upload them to a container service, like a container instance.

From this we can identify 3 ones that are actually able to host the application: AKS, container apps and container instance.

To choose the right one, let’s review the app’s requirements:

* App consists of multiple services running on different docker images
* App is expected to run on-premise at different locations
* App should be scalable, currently is not

With these criteria in mind, the container apps are the best choice. The container instance is meant for a single container, while our application consists of multiple containers. While we could still create one big image of all the services combined, it completely undermines the whole application’s architecture. It’s built on multiple services so they can scale independently, turning it into one application would not allow this to happen. The Kubernetes service is also a good option, but due to time constraints we cannot implement this. Container apps on the other hand automatically orchestrates the different containers so they can automatically scale up or down depending on demand.

## How do we upload the application to the cloud service?

Now that we know which service to use and which provider, we can start actually uploading our app to the cloud service provider. Due to time constraints, we limited this deployment to only upload the backend. Even though it is not the full application, we are aware of the steps to take to upload a part of it which should prove very helpful for either us during the transfer, or the next group to work on this project.

Azure Container Services allows you to add multiple containers to make your application. To make this work for our implementation, we first uploaded our different services as Docker images to our private Azure Container Registry. Because of the versatility of Docker, we didn’t have much trouble with this step. To upload the images to the azure registry, we only had to install the Azure CLI to log in to Azure, prefix the image with the name of the registry and push the image. Docker does all the work relating to pushing the image, and Azure only handles the authentication (Microsoft, 2023). We now have a registry with the different parts of our application in the cloud environment:

Afbeelding met tekst, schermopname, lijn, Lettertype

Automatisch gegenereerde beschrijving

Figure 1: List of images in our container registry

Next, we created a new Azure Container App instance. This would take our images from the registry and host them in a controlled, orchestrated system. This is done very easily by creating a new revision and selecting all the images to put in the container app. On top of just adding or deleting the images to use for this revision, we can adjust a lot of the settings that are used as well. For instance, the amount of CPU cores and memory that are assigned to each image, specific environment variables for an image and fine-grain control over scalability.

Afbeelding met tekst, schermopname, nummer, Lettertype

Automatisch gegenereerde beschrijving

Figure 2: Azure portal screen to deploy a new revision to the container app

Because this implementation is not meant to immediately be put in production, we did not touch any of the scalability or performance settings as everything works normally without it. For any future implementations where the userbase is expected to grow, these options should be adjusted to accommodate this.

Once a new revision is deployed, it takes some time to provision, and then it’s fully available on the web! This multi-container solution did present some problems, however. One of these was the built in ingress service. Ingress exposes HTTP and HTTPS routes from outside the app to services within the app (Kubernetes, 2023). Within the container app, we can specify one port that the ingress will target to send traffic to.

With the first deployment there was no API gateway yet. This meant that we could only expose one port to the internet at a time. The containers that did not have their port targeted could not be accessed by anyone, leaving parts of the application unusable. This was then fixed by creating the aforementioned API gateway. The problem was fixed, because the ingress could then target the port of the gateway so that the gateway could send the traffic to the correct destination.

After the gateway was built and everything was configured correctly, we could finally send requests to the application. To send these requests, we used Postman.

Afbeelding met tekst, schermopname, software, nummer

Automatisch gegenereerde beschrijving

Figure 3: Successfully created new survey on the deployed app

Afbeelding met tekst, software, nummer, Webpagina

Automatisch gegenereerde beschrijving

Figure 4: Successfully fetched the created survey from the deployed app

Afbeelding met tekst, schermopname, software, nummer

Automatisch gegenereerde beschrijving

Figure 5: Successfully submitted a new response to the deployed app

Afbeelding met tekst, software, nummer, Webpagina

Automatisch gegenereerde beschrijving

Figure 6: Successfully fetched all the responses from the survey, showing internal messaging works within the cloud app

The backend application works completely within the cloud deployment, while being in separate containers and communicating via a message queue.

As part of the transfer for the next project, I have included the steps of how to do this in the README.md of the git repository.

# Conclusion

To upload our application to a cloud provider, we first looked at the advantages that cloud providers give. We found that namely scalability, cost, flexibility and reliability & availability prove to be big advantages of uploading an application to the cloud. Next, we noted down our application’s requirements for cloud hosting as a base for determining the rest of the research questions. Based on previous stakeholder meetings we set up the following list of requirements:

* No vendor lock in. The company wants the system to be also deployable on-premise, which means that tight coupling with a cloud platform might give issues down the line.
* Costs should be kept to a minimum.
* Support to scale services independently.

Based on this list, we set out to research which cloud providers were the most suitable. We first chose the top 3 cloud providers to thin out the amount of choices to pick from. Then, we slimmed down the services from these providers to a list of services that apply more to our case, like data storage, container orchestration and standard app hosting. Now that we had filtered the massive list of cloud providers and their services to a small list of providers and service types that are actually relevant to us, we could finally make a decision on which ones we would pick for the actual implementation.

We chose Azure as our cloud service provider because of a number of reasons. Mainly because the platform is very secure and offers data storage integrations that will be easy to replace later when deploying the app on-premise. The most important reason however, is because the client requested it. They have to work with the app after we are done, and since they are already familiar with Azure it will not be ideal if we suddenly change the cloud provider.

Within Azure, we chose to use ‘CosmosDB for MongoDB’. This is because it uses the MongoDB API, so later when we swap the storage host to a new one we don’t have to change the code, only some configuration. Avoiding vendor lock-in is one of the most important requirements of our application and this is a good step in that direction.

For the app deployment we chose to use ‘Azure Container Apps’. Container apps allow you to upload your application in multiple containers. It then hosts these containers and handled orchestration, load balancing etc. for you. Since we do not have anything like this built into the application, like Kubernetes, this proves to be a very good solution. When developing this app further to run locally, the container app can be replaced with a Kubernetes configuration.

Finally with the provider and services set in stone, we started to upload the application to the Container App. Due to time constraints we were only able to upload the backend, though with these steps known and documented it should not prove difficult to upload the frontend as well in the future. First, we created a container registry to upload the services’ images to. Once the images were uploaded, we created a new revision for the container apps. When this revision was deployed, we noticed that we could only expose one container’s port in the ingress to the public, meaning only one service could really be used. To fix this, we created an API gateway that takes the incoming traffic and sends it to the correct service. This way, we could tell the ingress to expose the port of the gateway to the public and the rest of the traffic could be managed by the gateway. After this was solved, the application worked like normal.

To finalize the cloud deployment, we documented the steps of how to upload the application to the Container App like we did so the next group could easily continue working on it.

# Recommendations

With the current implementation of the cloud deployment, there are a few things to improve or change. Namely, a few things to reduce vendor lock-in. These are:

To move away from the Azure Container App service, or any other service that handles container orchestration for you, set up your own Kubernetes configuration/service. This gives you some more fine grained control over how the scaling happens, but most importantly it allows the application to scale without an external service being connected to it. When deploying the application to a local environment the application can work the same as if it was on the cloud (in this use case). Additionally, it allows you to upload the application to any other cloud provider without having to worry if it has a service for container orchestration.

Move away from using Azure’s CosmosDB, and instead have something you can host locally. Set up a MongoDB database to run alongside the rest of the application. That way you can have the database store data on an organization’s own hardware instead of having it hosted in the cloud. With the way we’ve set it up, it should only be a matter of configuration.

Of course, if you still continue working with the application that is deployed on the cloud environment, upload the frontend as well. With the frontend added, it can be considered the full application instead of just a backend.

# Sources

*Amazon Microsoft and Google cloud infrastructure market*. (2022, October 31). Retrieved from Wire19: https://wire19.com/amazon-microsoft-and-google-cloud-infrastructure-market/

Amazon. (n.d.). *Six advantages of cloud computing*. Retrieved from AWS: https://docs.aws.amazon.com/whitepapers/latest/aws-overview/six-advantages-of-cloud-computing.html

Amazon. (n.d.). *What is cloud computing*. Retrieved from AWS: https://aws.amazon.com/what-is-cloud-computing/

c-charpcorner. (2023, May 30). *Top 10 cloud service providers*. Retrieved from c-charpcorner: https://www.c-sharpcorner.com/article/top-10-cloud-service-providers/

Daniel, C. (2023, January 24). *Microsoft Azure Revenue and Growth Statistics (2023)*. Retrieved from SignHouse: https://www.usesignhouse.com/blog/microsoft-azure-stats

Google. (2023, April 25). *AWS Azure GCP Service comparison*. Retrieved from Google: https://cloud.google.com/docs/get-started/aws-azure-gcp-service-comparison

Google. (n.d.). *Advantages of cloud computing*. Retrieved from Google Cloud: https://cloud.google.com/learn/advantages-of-cloud-computing

Greene, T. (2023, January 19). *The Hidden Costs Of Cloud And Where To Find Overspending*. Retrieved from Forbes: https://www.forbes.com/sites/forbestechcouncil/2023/01/19/the-hidden-costs-of-cloud-and-where-to-find-overspending/

IBM. (n.d.). *Cloud computing benefits*. Retrieved from IBM: https://www.ibm.com/topics/cloud-computing-benefits

Kubernetes. (2023, June 1). *Ingress.* Retrieved from Kubernetes: https://kubernetes.io/docs/concepts/services-networking/ingress/

Microsoft. (2022, December 10). *Quickstart: Create an Azure container registry using the Azure portal.* Retrieved from Microsoft Azure: https://learn.microsoft.com/en-us/azure/container-registry/container-registry-get-started-portal?tabs=azure-cli

Microsoft. (2023, March 8). *Push your first image to your Azure container registry using the Docker CLI.* Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/azure/container-registry/container-registry-get-started-docker-cli?tabs=azure-cli

Microsoft. (n.d.). *How to choose a cloud service provider*. Retrieved from Microsoft: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/choosing-a-cloud-service-provider

Mohamed, R. (2019, July 9). *Cloud Application Platform vs Container as a Service vs VM hosted application*. Retrieved from Suse: https://www.suse.com/c/cap-vs-caas-vs-vm-hosted-application/

MongoDB. (n.d.). *Atlas Database. Deploy a multi-cloud database.* Retrieved from MongoDB: https://www.mongodb.com/atlas/database

Quali. (2023, June 7). *What is Reliability in Cloud Computing?* Retrieved from Quali: https://www.quali.com/blog/reliability-cloud-computing/

Red Hat. (2022, May 10). *What is container orchestration?* Retrieved from Red Hat: https://www.redhat.com/en/topics/containers/what-is-container-orchestration

Saviant Consulting. (n.d.). *10 Reasons Why to Choose Azure for your Enterprise*. Retrieved from It360: https://www.it360.biz/10-reasons-why-to-choose-azure-for-your-enterprise/

Simplilearn. (2023, April 6). *What is cloud scalability*. Retrieved from Simplilearn: https://www.simplilearn.com/what-is-cloud-scalability-article#:~:text=2.,without%20investing%20heavily%20in%20hardware.