Cloud deployment

research document

for

HeardIT

Author: Mihail Vasilev

Date: 10-04-2024

Contents

[1. Introduction 3](#_Toc167033743)

[2. Main question 3](#_Toc167033744)

[3. Sub-questions 3](#_Toc167033745)

[4. Conclusion 10](#_Toc167033746)

1. Introduction

The purpose of this document is to outline the research process taken during the cloud deployment process of the HeardIT application. The research is split into the following sections *– Main research question*, where the main question is defined, *Sub-questions*, where the sub-question derived from the main questions are defined and answered and *Conclusion*, where the answers to the sub-questions are combined in order to answer the main question. The final section is the *References* section where the sources of the information used during the research are presented.

This research takes into account a lot of different aspects of the cloud development sphere. This document reflects my current level of application deployment. During the duration of the semester new parts will be added, updated and expanded upon.

1. Main question

In this section the main question will be established. In order to complete the research, the main question needs to have a concrete answer. For this reason, it is important to define the main question well. The main question will also allow us to create the sub-questions that will help us answer the main question.

The main research question is:

*What technologies and methods are most suitable for deploying HeardIT to the cloud?*

Answering this question will allow us to determine the best architecture design for the HeardIT application. In the next section I will establish the sub-questions that were derived from the main question and I will formulate an answer to each of them.

1. Sub-questions
2. What cloud providers are suitable for the HeardIT application?

There are many cloud providers that can be suitable for a web-application like the one I am working on. For this reason, the choice will be influenced by the specific requirements for technologies and scalability that have been established beforehand (*check User Requirements – HeardIT.docx)* and also the budget.

One of the aspects that I wanted to touch upon during the development of HeardIT was the usage of a professional cloud service provider since this would allow me to gain real experience working with a popular and widely used platform. Additional functionalities, which I might have not known that I would need could also come in handy once I actually started using the cloud services and a provider with a well-established community support and documentation would make my work process easier and would create less risks and dependencies.

After considering both my technical and non-technical requirements, I came to the conclusion that most suitable cloud provider would be one of the big three: *Amazon Web Services (AWS)*, *Microsoft Azure* or *Google Cloud Platform (GCP)*. In the table I have summarized the most important factors that influenced my choice between the three main cloud providers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cloud provider | Compute and Containerization | Database and Storage | Networking and Security | Monitoring and cost management |
| Amazon Web Services (AWS) | Comprehensive range of computing options suitable for various workload types: EC2 for virtual servers, ECS for container management and Lambda for serverless computing | Amazon RDS, DynamoDB, Aurora, Amazon S3, EBS and Glacier | Robust networking features and security options for building secure and scalable applications: Amazon VPC for network isolation, AWS Direct Connect | Various pricing models and tools for cost estimation and optimization.  Amazon CloudWatch for monitoring and logging. |
| Microsoft Azure | Virtual Machines, Azure Kubernetes Services (AKS), Azure Functions | Azure SQL Database, Cosmos DB, Azure Blob Storage | Azure virtual network, ExpressRoute and advanced networking features | Azure Monitor and Azure Log Analytics |
| Google Cloud Platform | Compute Engine, Google Kubernetes Engine (GKE) and Cloud Functions | Cloud SQL, Cloud Spanner and Cloud Storage | Cloud Virtual Network and Cloud Interconnect | Stackdivre Monitoring and Stackdriver Logging.  Different pricing models depending on usage. |

Each of these cloud service providers brings a lot to the table and allows their users to have a reliable, easy to use and relatively affordable cloud platform where they can deploy their applications. AWS is by far the most widely used cloud provider, followed by Azure and GCP as stated in Slingerland (2023). Choosing any of them would be a great option for my project since all provide a lot of features and capabilities and have great communities and documentation.

In the end it came down to personal preference and the budget constraints. My budget for this project is minimal I want to make the most of any free package that cloud providers could offer. Microsoft Azure was a great option since I could use my university account to have a free package of 100$ credits to use. However, Google Cloud Platform offered a 90 days and 300$ credits for their free package. Additionally, they provide a service which allows the user to set-up their application in such a way that when it is not being actively used, credits are not deducted. Also, they provide convenient deployment capabilities directly from GitHub Actions which greatly reduced the effort put into creating the CICD pipelines for HeardIT.

For this reason, my final choice as the cloud provider that I am going to use for the HeardIT application is Google Cloud Platform (GCP).

Methods used:

* Literature study – this method was used to determine the general functionalities, requirements and costs that each cloud provider offers
* Available product analysis – this method was used when researching what options for a cloud provider I have
* Document analysis – this method was used to go more in depth into each of the cloud providers and determine which one would be the best for my needs

1. How to automate the deployment process to the cloud?

As a part of my project, I wanted to automate the deployment process of the application to the cloud provider. The professional way of deploying applications to the cloud is through the use of a CICD pipeline. A CICD pipeline which stands for Continuous Integration/Continuous Delivery (or Continuous Deployment), is a set of automated processes that allow software development teams to deliver code changes more frequently and reliably. Generally, a CICD pipeline is split into two main sections: the Continuous Integration (CI) is responsible for verifying that the application is built, passes all tests successfully and the code quality is checked for inconsistencies; the Continuous Delivery/Deployment (CD) is responsible for the deployment process of the application to the cloud.

With the definition of what a CICD pipeline is established, the next step was to find out what tools are available that provide CICD automation. The requirements for this are relatively straight forward: it has to provide me with all the required automation for the building, testing and code quality; it has to have connection to the cloud services that I am going to deploy to and in terms of budget, it has to be free. The tool with the least amount of effort to setup would be the preferred choice for me.

These requirements narrowed my options significantly. I use GitHub as my code repository for the HeardIT application. GitHub has an integrated pipeline service in the form of GitHub Actions which provides free runners that can execute the commands and processes for the CICD pipeline. It also has a free plan which is unlimited in duration and provides a significant number of free minutes per month for process automation. (Features • GitHub Actions, 2024) For a bigger company, where commits are done on a much greater scale, the free plan for GitHub Actions might not be enough for their CICD pipelines but in my case, it serves perfectly since this is an individual project.

GitHub Actions also provides its users with a lot of support and documentation and is widely used by the GitHub community. This means that there are a lot of materials and sources that I can use when implementing my CICD pipeline.

For these reasons, I decided to use GitHub Actions as my CICD tool. This would allow me to seamlessly automate the build, test, check code quality and deploy to the cloud processes of my application.

Methods used:

* Available product analysis – this method was used when researching what options for a CICD pipeline I have
* Document analysis – this method was used to go more in depth into the functionalities I can introduce in my CICD pipeline
* Community research – this method was used to see what other users have been using for creating their CICD pipelines

1. How to set-up the CICD pipeline for the front and back-end repositories?

At this point I had already established that I was going to use GitHub Actions as my CICD pipeline and I would be using Google Cloud Provider as my cloud deployment service. The next step is to actually create my pipeline. To do this I firstly had to determine the main steps that had to be executed in my pipeline.

These steps are the following:

1. Build step – the application is built
2. Test step – the application is regression tested
3. Check the code quality – determining if there are any issues or improvements in code quality and security aspects
4. Deploy – the deployment of the application to the cloud

With these main automation steps established, the next part is to find out how implement them. To do so, a YAML configuration file is created inside the repository which specifies each step and what to be done in each of them. This configuration fie includes many technical specifications such as the type of runner that is going to be used to execute the specific step. In some cases, some tasks should be executed on specific operating systems but in my case, this is not an issue. Because of that I decided to stick to the main Linux based runners. They provided me with the required capabilities to execute my tasks.

The process for doing the building and testing is relatively straight forward, the runners is executing a build of each part of the application. Then the application is tested by running the defined tests. The code quality is checked with the use of Sonar Cloud which is code quality and security tool that can be connected to GitHub repositories.

The code quality and security aspects of the application are expansive topics that deserve their own researches so I will not go into too much detail here. For more information, check \_\_\_\_*the code quality documents*\_\_\_\_\_\_\_.

Once the application passes the build and test steps, it can be deployed to the cloud. To do this a docker container is firstly created and published to Docker Hub. Afterwards, the GitHub Actions runner authenticates with the Google Cloud Provider service and the container is pulled and deployed to the cloud. With this the automation process of the CICD pipeline is finished. The result is that my application is automatically tested and deployed to the cloud.

In **Figure 1**, you can see an illustration of the CICD pipeline process.



Figure 1 - CICD Pipeline

Methods used:

* Literature study – this method was used to determine the general functionalities, requirements and costs that GitHub Actions offers
* Document analysis – this method was used to go more in depth into the functionalities I can introduce in my CICD pipeline as well as documentation on how to implement them
* Community research – this method was used to see what other users have been using for creating their CICD pipelines
* Unit test – this method was used for creating the unit tests that are executed in the pipeline
* Prototyping – this method was used for the implementation of the steps that need to be executed in the pipeline

1. How to deploy back-end cluster to Google Kubernetes Engine (GKE)?

The next part of cloud deployment was to find out how to deploy my backend services to a cluster on Google Cloud. To do this I had to follow several steps and perform several actions. These steps, in order of execution are:

1. Create a cluster on GKE.
2. Install Google Cloud SDK Shell and authenticate with Google Cloud
3. Create the Kubernetes yaml file with the proper deployments, services and other configurations.
4. Deploy the yaml file to the GKE cluster

Now that the main steps to deploy a Kubernetes deployment to GKE are established, I will explain in more detail the actions that are required in each step.

4.1 - In order to deploy my application, I first needed to initialize a cluster on Google Cloud. To do this I followed the guides provided by Google in order to create my project and create a cluster that is connected to it. The process is relatively straight forward and for this step, everything is in the GKE dashboard. From there I had to create a cluster, set a chosen cluster name and regions and then wait for it to initialize. The process of the actual initialization of the cluster is done by google automatically. It is normal for the cluster to take several minutes to finish initializing.

4.2 - Once this is done, there are several ways that one can have access to the cluster. I chose to install on my local machine the Google Cloud SDK Shell which allows me to directly connect to my cluster and be able to execute commands like the kubectl commands needed to deploy my services. The process is also fairly straightforward with the installation, I just followed the instructions and prompts provided by the installer. An important thing that is needed here is to authenticate with Google Cloud. To do so, use the same account that was used to create and initialize the cluster. Another important detail was to select the same region as the one my cluster was created in. Google has many different regions where clusters are hosted and maintained. Depending on the chosen regional settings, the deployment may behave differently and have different costs. Different regions have different price for their usage. This is a thing that I also had to take into account. For my project, I chose to host it in europe-west4 due to the reasonable price and regional options. Once I was authenticated and had selected my cluster, the next step was to deploy my application.

4.3 - To deploy a microservices application using Kubernetes, one must first create yaml configuration files where you specify the types of pods, deployments, services, gateways and other configurations are that needed. In my case, I create a deployment type for each of my backend services. Each of these deployments gets the latest version of my backend services docker images and creates pods using the docker images. The docker images are extracted from DockerHub, where they are stored. Services, that are used to establish the inner-cluster communication between the services, are created for each deployment. In order for the cluster to be accessible from the outside world, or in my case my front end, I created a gateway configuration. This gateway uses a gatewayclass controller to facilitated the exposing of the cluster and a loadbalancer that controls the incoming load to the cluster. An httprouter is also created so that requests are routed to the proper service inside the cluster. With this my yaml configuration file was created. The next step was to deploy it to GKE.

4.4 - To deploy the yaml file, I had to follow several steps that were needed in order for my configurations to work. GKE has the main kind of configurations preinstalled already, however, I wanted to use specific gatewayclass controller for my gateway.

Gateways in Kubernetes use different kinds of controllers depending on the type of cloud provider you are using. For example, Google, AWS and Azure have their own specific gateway controllers that may differ in the specific ways to set them up. There are also many different open-source gateway controllers that again have some differences depending on the use case. They all follow the Kubernetes way of defining the gateway and httprouter but may differ under the hood with the actual process. In my case, I decided that a gateway controller that is easy to setup and use is Envoyproxy. It is easy to set-up and use while also providing me with all of the needed functionality. At first, I wanted to use the GKE gateway controller, but due to it being proprietary and also requiring me to buy a domain, I opted to use Envoyproxy.

Back to deploying my yaml file, I first had to download and establish the Envoyproxy gatewayclass controller. This was done by following the guide on the [Envoyproxy](https://www.envoyproxy.io) website. I encountered some issue but with some debugging I was able to install the gatewayclass controller. The last step was to run *kubectl apply -f hearditgke.yaml* to deploy my application to GKE.

With this I had finished deploying my application to Google Cloud. The process that I described above summarizes the full process that I went through when researching and doing the deployment of the application.

Methods used:

* Literature study – this method was used to determine the actions that I had to take in order to deploy to GKE
* Document analysis – this method was used to get information about how to deploy my application to Google cloud. I used the Google Cloud Docs in order to get specific information about the GKE environment.
* Community research – this method was used to see what other users have done when deploying their applications.
* Peer review – this method was used to see how my colleagues have approached the deployment to a cloud environment
* Root cause analysis – this method was used when I was working on determining why my gateway was not getting properly set up. This was the biggest challenge when deploying to GKE because the information was not very well structured and I had to analyze the root cause of the issue by myself.

1. How to perform load testing of the application?

Creating and deploying HeardIT to the cloud is a big step in the development process. The next step is to establish the automatic load testing of the application. Load testing is a tremendously important part of developing modern enterprise level applications. It handles and monitors the performance of the application under heavy loads. It does so by creating a simulated environment, where the application is tested as if real users are interacting with it. This kind of testing allows developers to see how their application will perform in a real-world setting, where hundreds, thousands or even millions of users are using their application. Data gathered from such testing can be incredibly valuable to the developers and can allow them to create better and more robust and stable products.

In my case, since my application is expected to have to handle a reasonable number of concurrent users, it is vital that such tests are performed. To do so, I firstly had to find a suitable tool that could provide me with accurate and proper load testing capabilities. There are many different tools that I could use to perform my load testing so I decided to compare two of the most popular once: ***k6*** and ***Apache JMeter***.

**k6** - open-source load testing tool designed for testing the performance of APIs, microservices, and websites. Great capabilities and ease of use with its developer-centric approach to creating tests.

Key features:

* Tests creation - uses JavaScript for scripting
* Performance - high performance and handling high loads with low resource consumption
* CI/CD Integration - supporting various integrations like Jenkins, GitHub Actions and others
* Metrics and Monitoring - detailed performance metrics, supports integrations with monitoring tools like Grafana, InfluxDB, and Prometheus

Cons:

* No native GUI, relies on third-party tools and additional setup for graphical interfaces.
* Limited protocol support - primarily HTTP/HTTPS.

**Apache JMeter** - a long-standing open-source load testing tool. Very versatile and can simulate a heavy load on a server, group of servers, network, or object to test its strength or analyze overall performance under different load types.

Key features:

* Tests creation - uses a graphical interface for test creation but also supports for scripting via plugins
* Performance – moderate to high but can be resource-intensive, especially under heavy load
* CI/CD Integration - can be integrated into CI/CD pipelines with some effort and requires some extra configurations
* Metrics and Monitoring – comprehensive reporting capabilities, extensible through plugins

Cons:

* Relatively resource-intensive, especially under high loads
* Relatively steep initial learning curve
* GUI can be cumbersome and slow with large test plans

After considering what these tools provide, I have decided to use **k6** since it is better suited to integrations into CI/CD pipelines and has a better overall performance and the hardware requirements for using it are relatively low.

The next part is to decide where I will be performing the load testing. To ensure the results that I will get are accurate and represent a real performance situation, I have to create an environment where the application experiences a simulated but real load. The way this is done by professionals is by creating a controlled deployment on a test server that is separated from the real deployment but still behaves like the real cloud system.

In my case, the best possible solution is by deploying my application to the Fontys NetLab servers. These servers provide sufficient deployment environment where I can simulate the real cloud deployment while also keeping the cost to a minimum.

To deploy my application services to NetLab, I had to create an Ubuntu VM which has Kubernetes installed and configured. I use Microk8s as the Kubernetes cluster orchestrator since it provides the needed functionalities for creating and running a cluster on a Linux based system. Once I had Kubernetes configured and ready to work, I started deploying my application to the cluster. The process follows similar steps as I have described in part 4.4 in this document. Now that my application is deployed, I had to install k6 so that it can be used to load test my cluster. With this, the main configurations were complete and I could start doing load testing on my application.

To do the testing I created a JavaScript test file which specifies what the test is going to do, what it is going to record and monitor and what are the expected results from it. Since the test creation is done using JavaScript it can be customized to the specific scenarios quite easily. In my case, I wanted to test the performance of my application when using the main functionality of the search\_service, which gets information about the songs and then returns it in the response. With this I have a load test created that can performed on the NetLab deployment of HeardIT.

The final step of the load testing procedure was to automate the load testing process from the CI/CD pipeline. This is the final and yet the trickiest part of the load testing research since I spent the majority of my time, working on establishing the connection between the GitHub runners and the Fontys NetLab server. To connect to NetLab, it is required to use a VPN connection and then in order to actually run my load tests on the server, I need to establish a SSH connection between the GitHub runner and NetLab.

To do the VPN connection I install a VPN connector on the GitHub runner and then authenticate with NetLab. This way the required VPN access in done and the runner has access to the server. However, it still cannot execute any commands or actions on my test server.

To enable the runner to run my test commands, I had to create an SSH key firstly locally on my development system (work laptop), then SSH into the NetLab server and authorize the connection. Then I allow the GitHub runner to use this SSH key and then SSH into the NetLab server. This way now my runner is allowed to actually run the load tests that I want to do on the Fontys server.

An important concern that is worth mentioning is the reliance on the Fontys NetLab environment. If my VM server is not running, GitHub runners cannot access it which means that they cannot run load tests on it. This is a part of the accepted risks that are present.

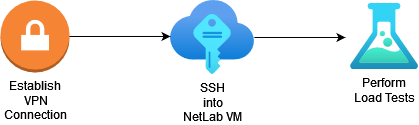
With this the load testing process of the application is completed. I will not go into analyzing the results of the load tests in this document as it is not the focus subject that is being discussed here. In **Figure 2** you can see the process of doing load testing visualized.

Figure 2 - Load Testing Process

Methods used:

* Literature study – this method was used to determine how to perform load testing
* Document analysis – this method was used to determine the capabilities of the load testing tools, how to deploy to NetLab and how to allow my runners to connect to NetLab
* Community research – this method was used to see what other users have done when doing load tests
* Peer review – this method was used to see how my colleagues have approached the load testing
* Non-functional test – this method was used when doing the load testing

1. Conclusion

To conclude this research, I am going to deploy the HeardIT application using GitHub Actions as my CICD pipeline automation tool and I am going to automatically deploy it to a cloud service using Google Cloud Provider. These tools and services have been chosen because of their compatibility with my application’s requirements, project needs and my personal decisions.

**References:**

* Slingerland, C. (2023, December 15). 11 Top Cloud Service Providers Globally In 2024. CloudZero. <https://www.cloudzero.com/blog/cloud-service-providers/>
* Chand, M. (n.d.). Top 10 cloud service providers (2023). <https://www.c-sharpcorner.com/article/top-10-cloud-service-providers/>
* Cloud computing services - Amazon Web Services (AWS). (n.d.). Amazon Web Services, Inc. <https://aws.amazon.com/?nc2=h_lg>
* Cloud Computing Services | Microsoft Azure. (n.d.). <https://azure.microsoft.com/en-us/>
* Cloud Computing Services | Google Cloud. (n.d.). Google Cloud. <https://cloud.google.com/?hl=en>
* Features • GitHub Actions. (2024). GitHub. <https://github.com/features/actions>
* Google Kubernetes Engine (GKE). (n.d.). Google Cloud. <https://cloud.google.com/kubernetes-engine?hl=en>
* Introduction - Kubernetes Gateway API. (n.d.). https://gateway-api.sigs.k8s.io/
* Envoy proxy - home. (n.d.). https://www.envoyproxy.io/
* Load testing for engineering teams | Grafana k6. (n.d.). <https://k6.io/>
* Joecolantonio, & Joecolantonio. (2024, March 24). 16 Top load testing software Tools for 2024 (Open- Source Guide). Test Guild - Automation Testing Tools Community. https://testguild.com/load-testing-tools/