Cloud Computing and Edge-Cloud Continuum Course

GCP Project description

This part has a 60% portion of the final grade.

The students need to prove their understanding of cloud technologies and architectures. They are supposed to understand the design, development, and management of dynamic solutions that improve the quality of services.

The objectives are:

- To get familiar with the case and consider which non-functional requirements are needed.
- To get familiar with GCP technologies and think about which quality attributes they provide.
- Deploying the solution allows you to get more hands-on with these technologies and learn how to use them.
- Adopting too many GCP technologies is not the central goal to achieve, but you should rather focus on understanding the requirements and technologies.

Use-case: Emergency Routing

Company Overview

OdenseEmergency (imaginary) is a provider of emergency handling software used by governments/organizations for their end users: first responders, firefighters, and citizens. They provide their software as a service to organizations globally.

Solution Concept

Due to rapid global warming and fires that occur around the world, this business has been growing exponentially year over year. The company provides a mobile app and needs to ensure good performance in real-time while considering the users' device limits. In addition, scaling its environment, adapting its disaster recovery plan, and rolling out new continuous deployment capabilities to update its software quickly are desirable if the resources allow them. Google Cloud has been chosen to complete/replace their current colocation facilities.

Existing technical environment

OdenseEmergency developed an application that routes people to safe areas in disaster. The application captures location and human emotions data and processes the data to change the application interface. The change is in line with helping people to make better decisions and evacuate safely and quickly. The application captures six types of basic emotions (joy, surprise, fear, anger, disgust, sadness) plus neutral, uses various adaptation techniques (such as rule-based, reinforcement learning (RL)-based, etc.), and performs four types of UI adaptation (change in background color, font size, street vs. satellite view, pop-up information). It is worth mentioning that the target emotions (that the app wants to push the users to them) are joy, surprise, and fear, and they get a reward in RL. The other emotions get punishment, and neutral is set to zero.

OdenseEmergency thinks that local processing and storage would create **performance** and **scalability** issues (the main non-functional requirements), but they are not sure about it. The managers are unsure about a suitable architecture. They would like to make their decisions based on the above-mentioned non-functional requirements.

Executive statement

An on-premises strategy may work but has required a significant resource investment. Many outages have resulted from misconfigured systems and inadequate capacity to handle spikes in traffic. We want to use Google Cloud to leverage a performant and scalable platform that can span multiple environments seamlessly and provide a consistent and stable user experience that positions us for future growth. However, a lot of pre-assessments are required.

The abilities to assess:

- Design and plan cloud solution architectures
- Design for performance and scalability [you could discuss energy efficiency (optional), but a test for that is not required]
- Analyze technical and business processes
- Manage implementations of cloud architectures
- Verify solution and operations feasibility/quality

Summary of what to do:

- Deploy the app as the instruction says and assess the performance, using maybe Jmeter
- Provide architecture solution. You could provide one conceptual architecture plus one simplified architecture corresponding to GCP deployment practically.
- Deploy the app on GCP and perform load testing (see the example provided on AWS)
- Fill in the report template as requested, including reporting the experiment's findings.
- Deliver a (maximum) 5-page report, a 5-minutes video explaining your deployment and architecture, and a link to your code.
- I appreciate group work but make sure to write your report individually. In your report, you should mention the names of persons you worked with as a group.

More Instructions:

An app is given (it is a PWA, so forget about it being a mobile app), and you need to deploy it both locally and on GCP. The instruction file is provided, and a template is uploaded.

The first step would be carefully reading the use case description and analyzing various specifications, desires, and requirements. Then you need to analyze each problem/requirement, keeping in mind how a cloud mechanism/architecture could handle that issue. You may suggest solutions based on what you learn in the lectures, GCP labs, and talks, or you could have a domain background check.

Choice of technology can be used as an argument for fulfilling an NFR, e.g., using a Redis Memory Cache allows fast, low latency access to data. See the deployment example below regarding testing the NFRs.

Now, based on what you understood, suggest an architecture. You may now deploy the app based on your architecture (if you did not already).
What is essential is to deploy the app on your PC and GCP and assess the non-functional requirements. In other words, deploy the app, perform a load test with maybe 2000 instances (see the example below), and report the results.
You preferably consider other aspects you analyzed that could be regarding computing, networking, edge-cloud, etc., and may impact your solution's quality.
Presenting your tests' results, you may refine your architecture, discuss your results, and mention what you learned.

What I assess when grading:
1. Sound architecture solutions;
2. *Deployment of the app;*
3. Using some mechanisms and features discussed in the lectures/talks/ lab sessions;
4. Testing the requirements;
5. *Discussion on lessons learned.*

What I suggest you deliver:
1st deliverable: An analysis of the use case, and its requirements, together with a primary architecture solution (4 Nov)
2nd deliverable: More details regarding architectures, deployment, and how you would test the requirements. (9 Dec)
Final Submission: The full report, including the final architecture, requirements analysis, testing, discussions, etc. [you could improve/change the content of previous reports]. Do not forget the demo video and link (29 Dec).

Example of the App Deployment on AWS

1. Using the Elastic Container Service

For orchestrating the containers which make up the service (frontend, backend, and admin dashboard), the Elastic Container Service (ECS) is used. ECS was chosen as, with ECS, it is possible to do auto-scaling, where the amount of instances of the image is increased based on the load (resource utilization) of the containers. In addition to this, the usage of AWS Fargate for orchestrating the containers within ECS enables the shutting down of containers when they are not used, thus also saving on deployment costs.

Defining a task in ECS. A task is defined in ECS. A task is a unit of work that is deployed in an ECS Cluster; a task can be likened to an instance of a single container. In the task specification, ports and environment variables are defined.

Defining a service in ECS. A service in ECS defines how a task should be deployed and scaled. It is also defined which load balancer should be used.

Health Checking ECS does health checking of containers. This is done through endpoints. This is done easily for the frontend and dashboard as they already expose */* and */dashboard*. However, parameters need to be supplied to the backend to return a response. Thus a */health* route was added to the backend server to reply to ECS. This ensures stability and that a suitable running container will not be shut down, because it is not responding. Adding this fixed issues with downtime.

Auto-scaling. The configuration of auto-scaling is done on a per-service basis. The decision to scale is based on the average CPU load registered by ECS. The scaling will occur when CPU usage is > 80%. The metrics were collected in CloudWatch is done in 1-minute intervals. This means that the scaling latency will be a minimum of a 1-minute interval + the time to spin up a new instance. To be able to scale quicker, the scale-in period is set to 45 seconds, such that. To be able to scale quicker, the scale-in period is set to 45 seconds. For application load balancing, listeners are set up for different ports, e.g., forcing HTTPS, forwarding dashboard traffic to the dashboard group, and forwarding the rest of traffic to the backend API.

2. Using Application Load Balancer

A load balancer is used to direct traffic to the different target groups. A target group is the pool of instances provided by the service defined in the ECS cluster. In the Application Load Balancer (ALB), listeners are set up for different ports.

- Port 80 directs to port 443 for the forcing of HTTPS
- With port 443, the path of */dashboard** forwards traffic to the dashboard group. All other traffic is directed to the frontend (PWA) group.
- With port 3000, all traffic is forwarded to the backend API.

One benefit of using the ALB is the use of HTTPS offloading. Here the ALB handles the HTTPS connection and communicates with the target groups by HTTP. This enables the services to only be concerned with HTTP communication. This is done by registering a certificate to the relevant listeners.

To enable the usage of HTTPS, a domain was used. To point the domain to the load balancer, a CNAME record was used.

3. Using DevOps for Deployment

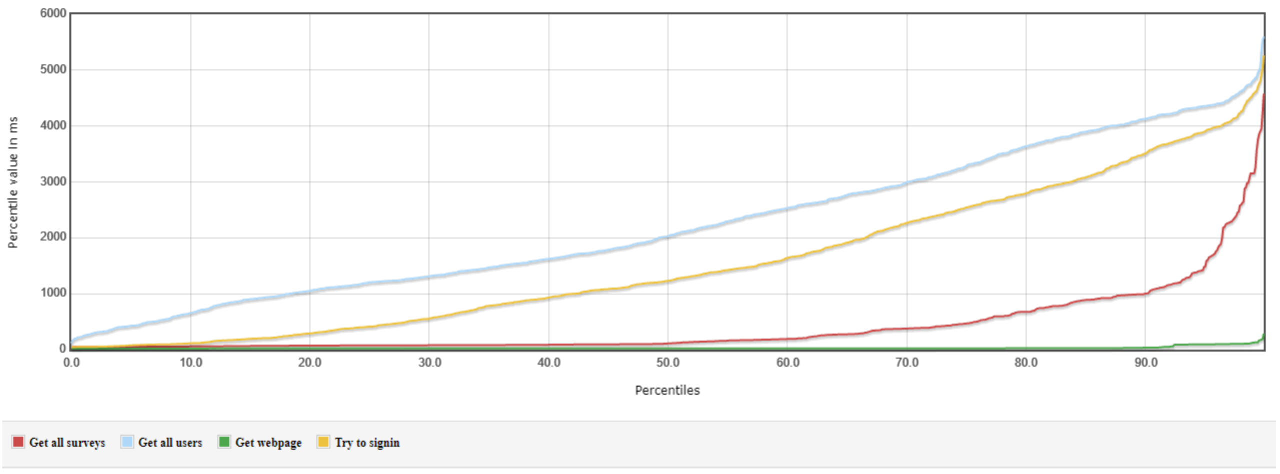
For the different components of the application, pipelines were created in GitHub Actions. The pipelines enable quick deployment of code without the developer's interaction.

The pipelines each build containers for the respective components. For the dashboard and PWA-app, the applications are built such only that the static files remain; in the container, these are server by a lightweight webserver, nginx. After building the image, then it is pushed to Amazon Elastic Container Registry (ECR). To deploy the image to service, then a task definition is changed to use the new image. This task definition is automatically deployed within the service. The pipeline run is completed when the new image has been deployed and is running healthily.

4. Some AWS Tests

We have created some performance tests based on querying the backend and getting it to scale to *> 1* instance of the backend. We set a failure to log in deliberately, just to stress the server. We observed no dropped connections, even when the server was pinned at *100%.* The service is somewhat slow to scale up, as the decision is based on a rolling average, but the scaling action does happen in due time. We also observed that when the usage is low, the desired target server count is scaled down.

The figure below shows that when all four instances are scaled to 2000, the response times remain within an acceptable level. For instance, when the system gets users' data, it experiences the highest response time of 2 seconds for the 50 percentile. For getting all surveys data, the response time of 1 second for the 90 percentile is recorded.



Percentile response time value in ms.