

PI- Reporting

Group (8) – Project Title: CovidHub

System Analysis:

Component	Technology	Reason
Services	Elastic Container Registry (ECR)	Repository for images
	ECS and Fargate	Serverless to run microservices
	VPC (Virtual Private Cloud)	Isolated network on the internet
	Load Balancer	Application Load Balancer for high availability
Database	Aurora MySQL	Serverless Storage
Backend	Node.js and Go	Languages for microservices
Frontend	React Framework	Serves user interface files to client

System Design:

Overview

A microservices architecture on a VPC (Virtual Private Cloud) using ECS (Elastic Container Services) and Fargate (a serverless compute engine that works with ECS). Three services Authorization, Text, and Email are run as tasks using Fargate and communicate with a serverless MySQL Aurora Database cluster.

Architecture (See Figure 1)

Two public subnets (10.0.1.0/24 and 10.0.2.0/24) exist in a VPC (10.0.0.0/16) accessible through an API gateway. These subnets are called availability zones. Inbound traffic is routed through the API gateway to the Load Balancer which forwards the request to the most available fargate task in the ECS cluster. Resources and monitoring of tasks are available through Cloud Watch.

Microservices: The client sends requests to Authorization, Text, or Email services within the cluster. Services can easily be added as the application grows.

High performance: Fargate tasks vCPU and memory resources are scaled based on policy. Less limitations on CPU, memory, and maximum run time than Lambda. We performed benchmarked testing. For each of the services we have sent 1000 requests on a certain endpoint via 1, 3 and 5 threads. The results are shown in figure 2.

Highly Availability: If a task fails or has low available resources, inbound requests are routed to a task that is available. Our Application Load Balancer redirects the client request to the available machine to ensure the all-time availability of our web application.

Security: The web application has some added components for different features such as login, signup, subscribe and unsubscribe. An API gateway is used to separate out the business logic from the features. Requests are only redirected to and from the client so that the other services are not directly available. The web application has two security groups for all services to allow traffic to or from its associated instances. The transfer of data into the RDS goes through the security group. For each security group, all traffic is allowed in outbound rules and only HTTP traffic is allowed in inbound rules.

Test Procedures:

1. Login (figure 3.a) with a user we have created (Username: "cmpt474test@gmail.com ", Password: "66jHKn5%4*"). This tests the authentication service.

2. Once logged in, the user receives a “successfully logged in” notification popup and it takes the user to Dashboard.
3. In the dashboard (Figure 3.c), we have two different services - text and email
4. User can choose for the service, he/she wants to subscribe, and it will redirect the user to the respective page.
5. Once a user knows which service to subscribe to, he/she can fill in the countries, frequency and email/phone.
6. The user can subscribe or unsubscribe to the services by choosing the respective option.
7. To test the working of the email and text service, visit the cloud-watch service in AWS, and select the respective service to view it's logs.

Goal measures & deployment considerations (Max 350 words):

1. Create a microservices based application which can be easily scaled (Fargate tasks) and managed (Cloud Watch).
2. Allow services to securely access a database which is also scalable (MySQL Aurora Serverless). Alternatively, we could have had each service talk to its own DB instance within its container.
3. Setup services so that they are all accessible within the subnet. Alternatively we considered setting up the ECS cluster with private subnets and a NAT gateway for added security.
4. Deploy the application manually to AWS (i.e. ECS cluster / Fargate, Aurora, VPC and subnets). Alternatively, we could have set this up as infrastructure as code for easy deployment and teardown.

Lessons Learned:

Member Name	Michael Zhagi	Manan Maniyar	Raghav Mittal	Maheepartap Singh	Manmeet Singh
Contribution (%)	20%	20%	20%	20%	20%
Individual Ref.	http://covid-project-s3-website-us-east-1.amazonaws.com/	http://ec2co-ecsel-1jni8txjq1cve-122992530.us-west-2.elb.amazonaws.com:3000/	EC2Co-EcsEl-YU443X08NCGI-351691330.us-east-1.elb.amazonaws.com:3000	http://covid-hub-lb-525281288.us-east-1.elb.amazonaws.com:3000/	http://lb-covidhub-1230398115.us-east-1.elb.amazonaws.com:3000/

Comments about Teamwork:

Teams divided into application development and deployment. Work as a group for deployment. Group could have had better communication with deadlines and progress.

Comments on Problem-Solving Approach:

AWS tutorials: Leverage tutorial content to help with deployment

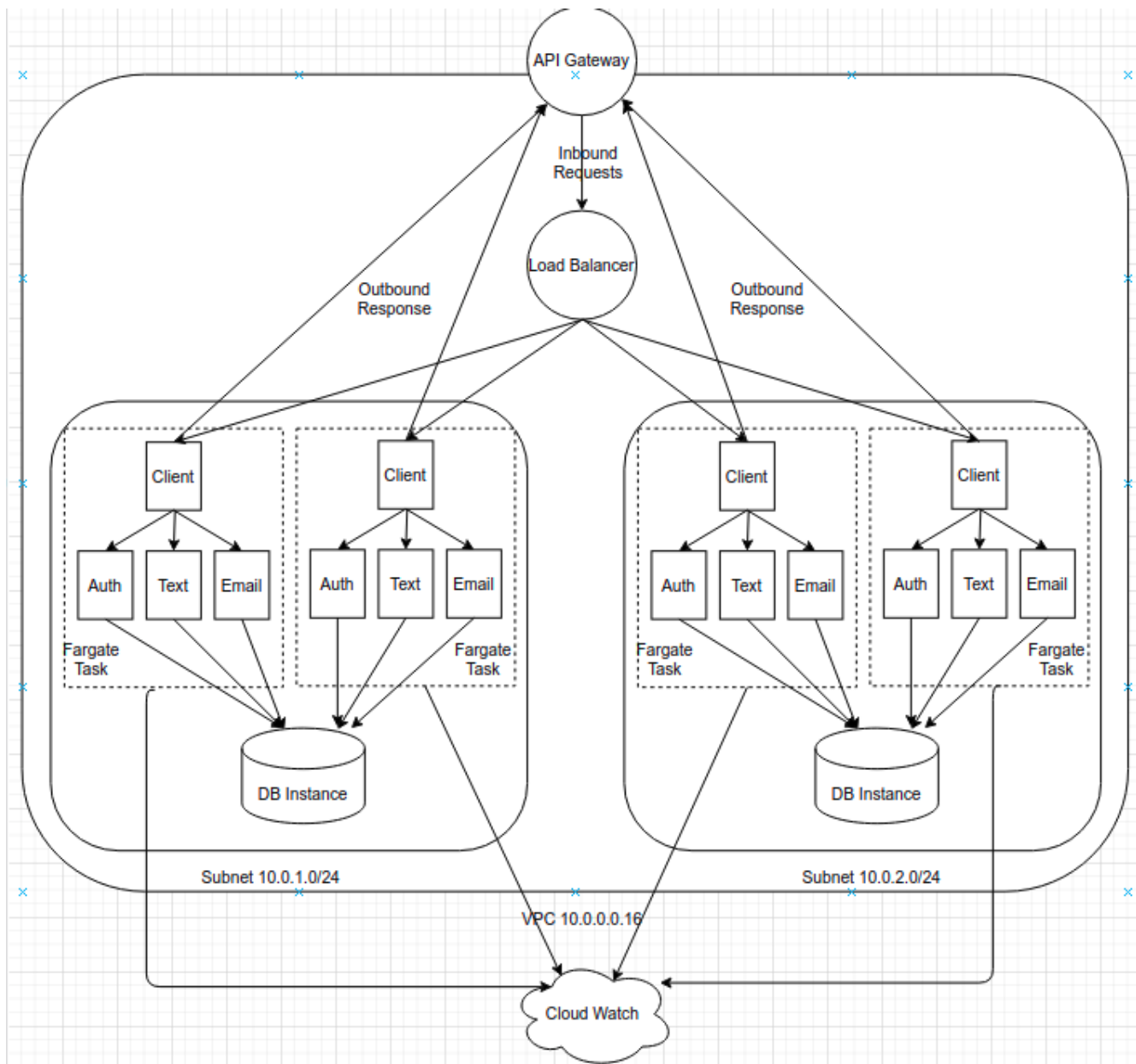
Adding a Load Balancer: To help with meeting project requirements

S3 (static) to Fargate (serverless): Possible to reduce fargate tasks and serve production static files.

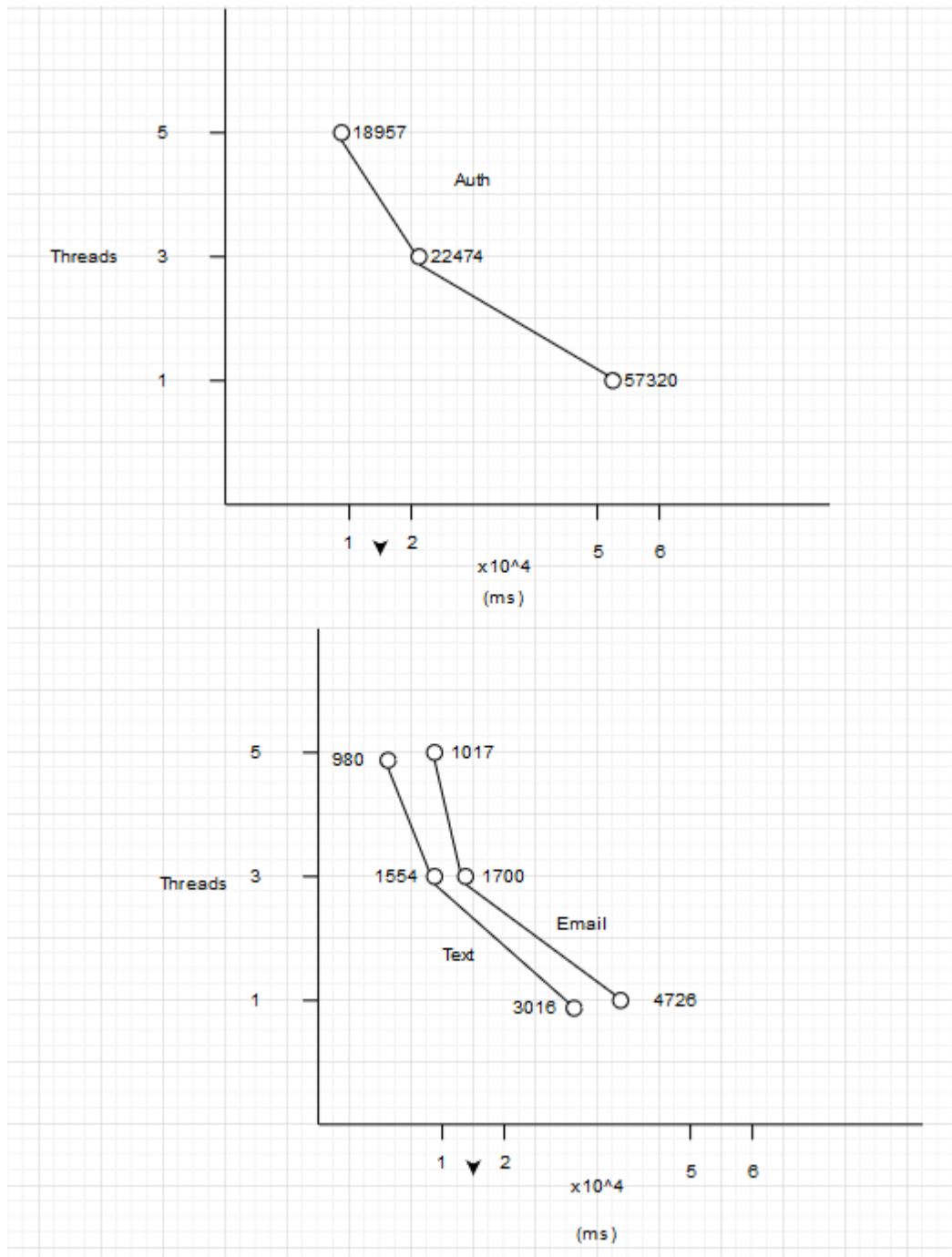
Comments on Choice of Technology:

- The reason to choose Microservices architecture is that different parts of the applications are developed and managed by different team members. It helped us to manage all the services in isolation and have communication between services using HTTP REST.
- To run serverless containers, our team used AWS Fargate which reduced the work of managing the services and helped in scaling smoothly. We used Application Load Balancer to manage the traffic and make the application scalable for future use.


- To make the applications interface we used React framework which is responsive for all types of screens such as desktop, tablet and mobile. To make the application more reactive we used Material UI libraries. Backend of the web application is written in Go and Node.js.



(Figure 1)



(Figure 2)



Sign in

[SIGN IN](#)

[Don't have an account? Sign Up](#)

(Figure 3.a)

Register

+

REGISTER

(Figure 3.b)

Service Information

E-MAIL

TEXT

Email

lal@gmail.com

Phone Number

+19999999999

Country(s)



Information Type *



Frequency *



SUBSCRIBE

GET INSTANT NOTIFICATION

If you wish to unsubscribe, click [here](#)

(Figure 3.c)