CSCI 5409 Term Assignment

Written Report

Sushank Saini
B00922727

Contents

Overview

The **Fourth Generation (4G) Task Manager** has been adapted from the concept of the 4 Quadrants time management tool introduced by Stephen R. Covey in his book "The 7 Habits of Highly Effective People." Covey's time management matrix is a framework that helps individuals prioritize tasks based on their urgency and importance. Urgent tasks are those that require immediate action or attention while important tasks are the ones with high significance or value to your goals. Accordingly, tasks are categorized into four quadrants:

1. **Urgent and Important (Quadrant I):** Tasks in this quadrant are both urgent and important. They require immediate attention and often relate to critical issues or emergencies. These tasks need to be dealt with promptly.
2. **Not Urgent but Important (Quadrant II):** Tasks in this quadrant are important but not urgent. These tasks contribute to long-term goals, personal development, and meaningful activities. It's essential to focus on Quadrant II tasks to prevent them from becoming urgent in the future.
3. **Urgent but Not Important (Quadrant III)**: Tasks in this quadrant are urgent but not important. They may seem pressing, but they don't contribute significantly to your long-term goals or priorities. Delegating or minimizing these tasks is often the best approach.
4. **Not Urgent and Not Important (Quadrant IV)**: Tasks in this quadrant are neither urgent nor important. They are time-wasting activities, distractions, or trivial matters that don't contribute to personal or professional growth. It's best to minimize or eliminate these tasks from your routine.

Covey's time management matrix encourages individuals to focus on Quadrant II activities, as they are crucial for personal and professional development [1].

Through this term assignment, I have tried to give a concrete shape to this abstract concept of the 4G time management matrix by creating the **Fourth Generation Task Manager web application using AWS cloud services**.

The application provides a brief description of what the 4G Task Manager is about(**Figure 1**).

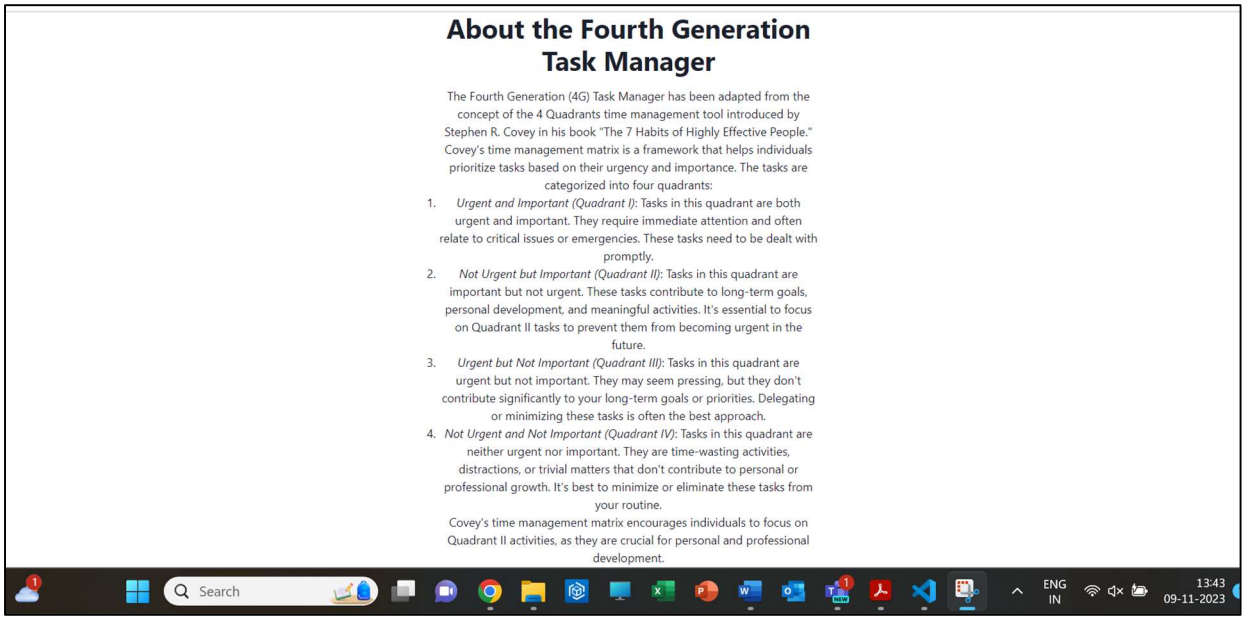

Figure 1: About section of the app.

The application then takes the user input from the user (**Figure 2**). Once the task is added successfully, the user can view the tasks by clicking on "View Tasks" button (**Figure 2**).

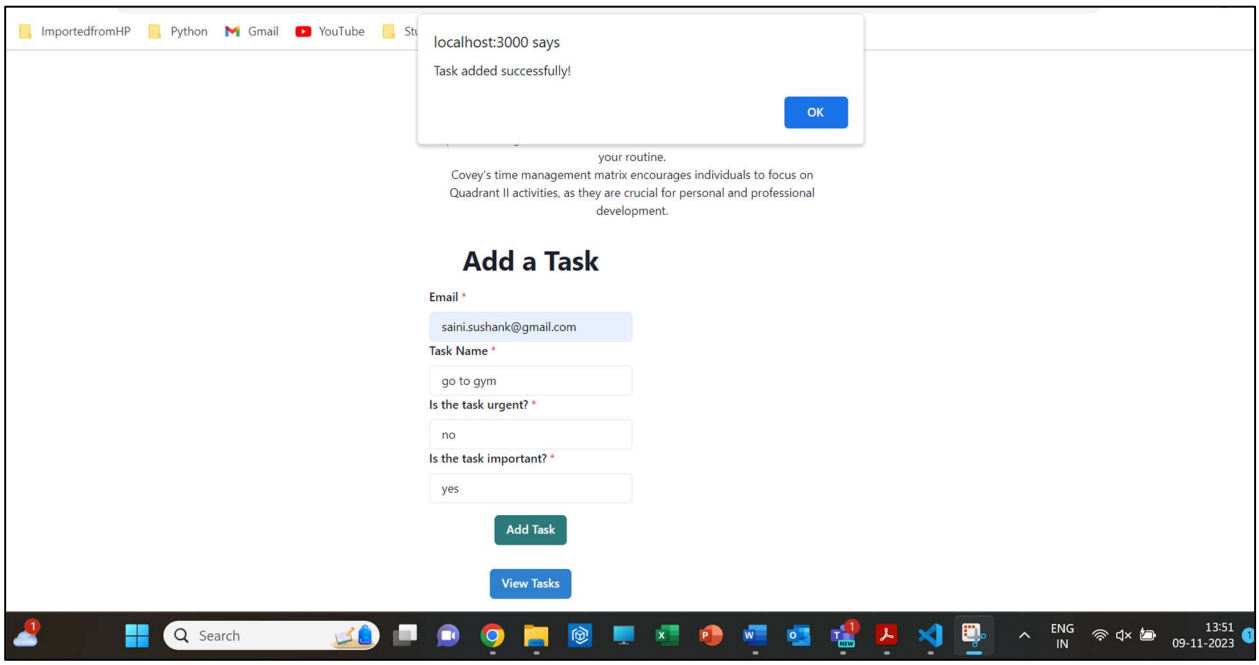

Figure 2: User adds a task and views it using the "View Tasks" button.

On the "View Tasks" page, the user can enter their email address which was used to add the task and see their tasks categorized into four quadrants based on the task's urgency and importance (**Figure 3**).

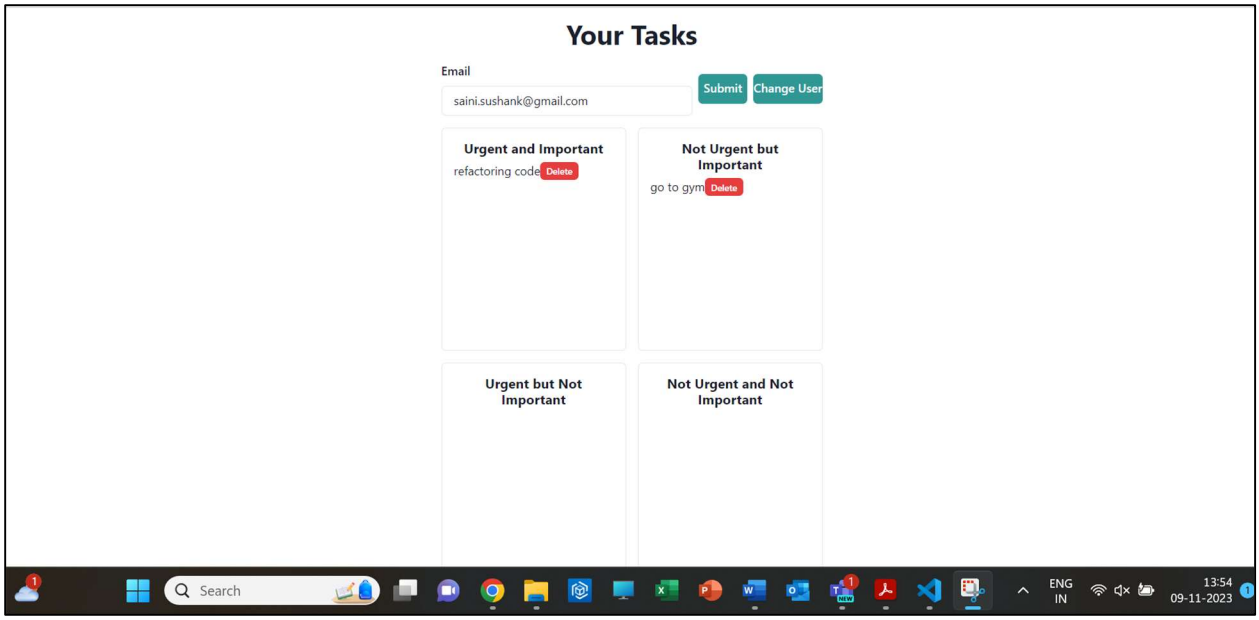

Figure 3: User views their tasks categorized into four quadrants.

If the task has been completed, then the user can delete the task by clicking on the “delete” button next to the task (**Figure 4 and Figure 5**).

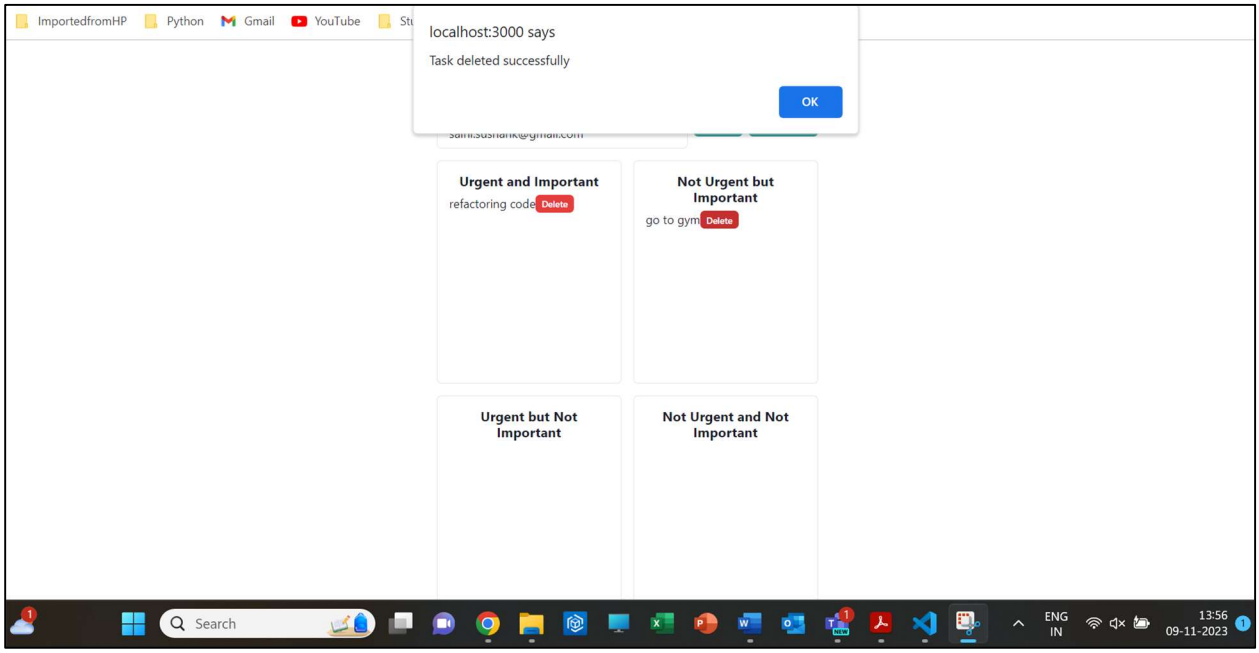

Figure 4: User deletes a task when completed.

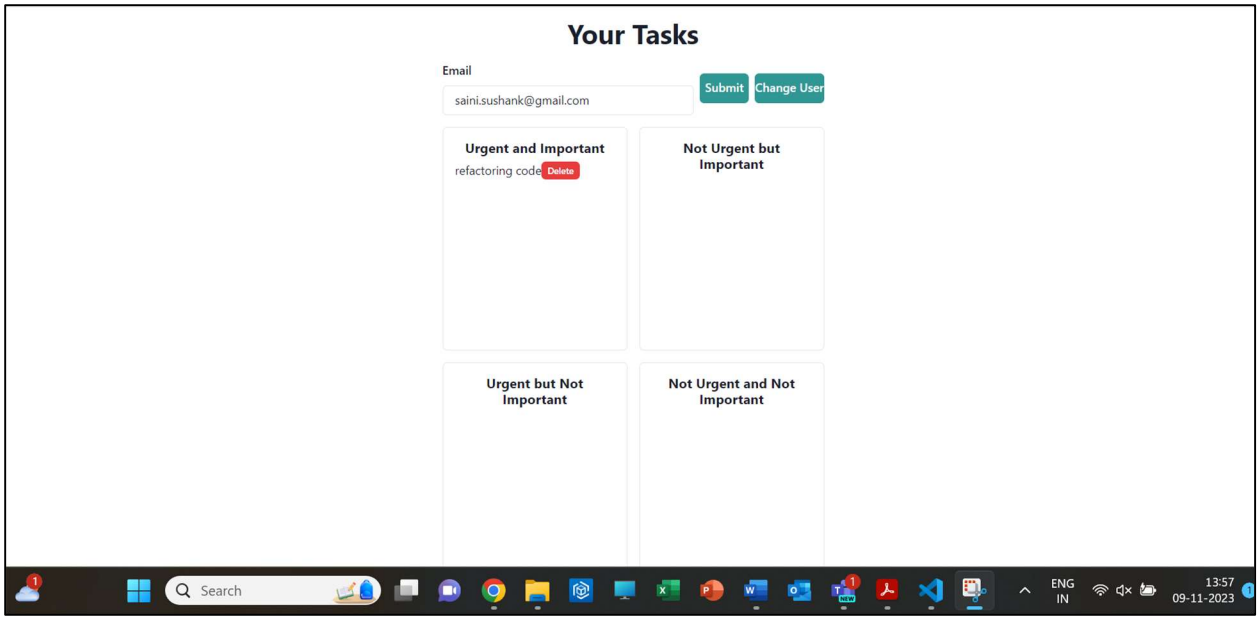

Figure 5: View after the task is deleted.

The application gives the option to the users to subscribe to the newsletter of the 4G Task Manager (**Figure 6**). Once the user confirms the subscription, they will receive push notifications from the app.

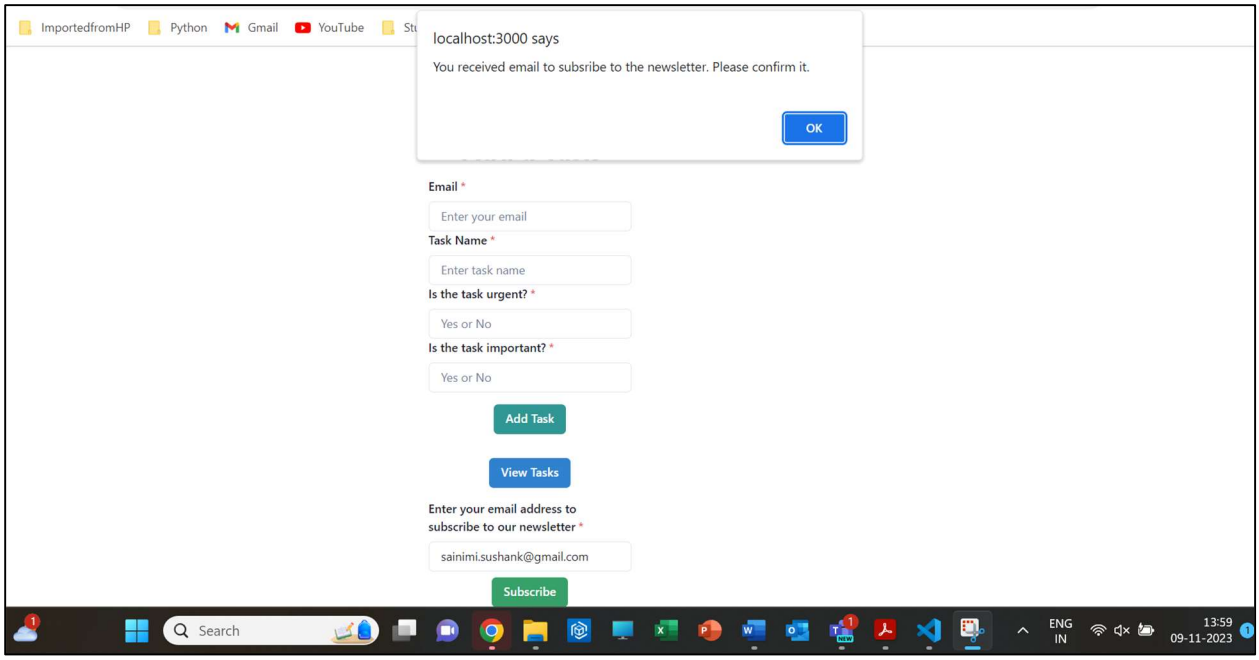

Figure 6: User subscribing to the newsletter.

List of AWS services used

Compute

I used **AWS Lambda** for my **backend services** and **AWS Elastic Beanstalk(EBS)** to **host my frontend.**

My application is event driven ,stateless and serverless . Accordingly, AWS Lambda, a **Function as a Service (FaaS)** fits these requirements. AWS Lambda makes it easy to execute code in response to events, such as updates to an Amazon DynamoDB table. Also, with Lambda, we **do not have to provision our own instances** as Lambda performs all the operational and administrative activities on our behalf, including capacity provisioning, applying security patches to the underlying compute resources, deploying code, monitoring, and logging code in CloudWatch. In addition, AWS Lambda **provides easy scaling and high availability** to our code **without additional effort** on our part. On the contrary, with **Amazon EC2,** which is **Infrastructure as a Service (IaaS)**, we are **responsible for provisioning capacity**, **monitoring** fleet health and performance, and designing for **fault tolerance** and **scalability**[2].

For hosting my frontend, EBS provides **pre-defined ready-to-use environment** comprised of already deployed and configured IT resources[3], as it is **Platform as a Service(PaaS).** In addition, EBS **automates the deployment details of capacity provisioning, load balancing, auto-scaling**, and application health monitoring, running a version of the application. As a developer, we **are freed from deployment-oriented tasks**, such as provisioning servers, setting up load balancing, or managing scaling[4]. However, these **same features are not provided by AWS EC2** as we must configure everything ourselves.

Since I am **following serverless architecture and not microservices** architecture, there was **no need to use containerized applications**. Therefore, services like Docker with EBS, AWS Elastic Container Service & Elastic Container Registry and Amazon Elastic Kubernetes Service have not been chosen. Likewise**, AWS Step Functions has not been used** as it is used to coordinate the components of distributed applications and microservices using visual workflows[5].

My application is a web application which does not have a use case for using simple devices and therefore, AWS IoT 1-Click has not been used.

Storage

I chose **AWS DynamoDB** for storage.

I picked this service because DynamoDB provides management of database software and the provisioning of the hardware needed to run it. It **automatically scales throughput capacity** to meet workload demands and partitions and repartitions the data as the table size grows. Also, DynamoDB synchronously replicates data across three facilities in an AWS Region, **providing availability and data durability**[6]**.**

Moreover, I wanted a **schema-less (NoSQL), non-transactional** database to get more flexibility with my data. Therefore, relational database services like AWS Aurora and AWS Relational Database Service were not picked.

In addition, the data in my application is **stored as key-value pair** and **not as object/file** whereas AWS S3 is a simple key-based object store[7] which does not fulfill the purpose of my

application. Consequently, AWS Athena was not used as there was no need to query any data in S3.

My application does not have any connected data which rules out using a graph database like AWS Neptune. Also, my application uses a single database and hence, there is no requirement to consolidate data from multiple databases using AWS AppSync.

Lastly, my application does not run on simple devices and hence, there is no need for analyzing IoT data using AWS IoT Analytics.

Network

I utilized **AWS API Gateway** for my APIs.

I picked this service as API Gateway is a fully managed service that makes it easy for developers to publish, maintain, monitor, and **secure APIs at any scale**. It acts as a "front door" for applications to access data, business logic, or functionality from back-end services. It handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, **including traffic management, authorization, and access control**[8].

On the other hand, even though **AWS Virtual Private Cloud** provides traffic management and security, it needs to be provisioned manually by the user.

General

I have picked **AWS Simple Notification Service(SNS) and AWS CloudFormation**.

I picked SNS service as SNS provides a highly scalable, flexible, and cost-effective capability to publish messages from an application and immediately deliver them to subscribers. It has several benefits such as instantaneous push-based delivery (no polling required), inexpensive pay-as-you-go model with no up-front costs and simple APIs & easy integration with applications[9] due to which I have chosen this service for broadcasting newsletter emails to users.

AWS CloudFormation is **Infrastructure as Code(IaC)** provisioning tool to provision and manage AWS resources in an orderly and predictable fashion[10]. This helps in quick deployments without manual intervention, error reductions and improved infrastructure consistency[11].

Deployment Model

I have chosen the **public deployment model**. The main consideration for using this deployment model is the **low cost**. Having a dedicated on-premises cloud would have cost more than the cost-effective pay-as-you-go pricing model of public cloud like AWS. Also, there is **no additional overhead** from investing in the **hardware and managing the infrastructure.** In addition, a public deployment model ensures the **highest degree of availability** relative to other deployment models[12].

Delivery Model

I have chosen **Function as a Service(FaaS)** delivery model for my **backend**, implemented using **AWS Lambda functions**. Using this delivery model helped me just focus on writing code and **not worry about provisioning or managing servers** which includes capacity provisioning,

applying security patches to the underlying compute resources, deploying code, and monitoring and logging my code. FaaS such as AWS Lambdas also provides **easy scaling and high availability** of the code without any extra efforts [2]. Additionally, my backend functionality is **event-driven, stateless, and short-lived** which perfectly fits with the use cases of FaaS [3].

To host my **frontend**, I have used **Platform as a Service (PaaS)** delivery model through the **AWS Elastic Beanstalk**. PaaS provided me with the **pre-defined ready-to-use environment** to deploy my frontend [3]. Also, using PaaS, the headache of **maintaining, scaling, and load-balancing** is passed to the cloud provider from the developers [4].

For **persistence storage,** I have used **DynamoDB** which is **PaaS** [13]**.** As a PaaS, DynamoDB offloads the administrative burdens of operating and scaling distributed databases from the users to AWS so that they don't have to worry about hardware provisioning, setup and configuration, throughput capacity planning, replication, software patching, or cluster scaling [6].

For configuring the **APIs**, I have used **API Gateway** which is a **Software as a Service** (**SaaS)** [8]. As a SaaS, API gateway reduces the cost and tasks of managing all aspects of creating and operating robust APIs for application back ends [14].

To **broadcast emails to the subscribers**, I have used **SaaS** through **SNS.** As a SaaS, there is no need for maintenance or management overhead and with pay-as-you-go pricing. Amazon SNS gives developers an easy mechanism to incorporate a powerful notification system with their applications [9].

Architecture

My application has a **serverless architecture** (**Figure 7).** In this architecture there is abstraction of backend servers as server management is done by the cloud provider. In AWS, this is realized through stateless, and event driven **Lambdas** which let us run the code without provisioning or managing servers. The **code for backend** is written in **Python** because it is simple and has readable syntax, making it easy to write code quickly. More importantly, AWS provides official Software Development Kits for various programming languages, including Python(boto3). These SDKs simplify the integration of Lambda functions with other AWS services.

In this architecture, we have a **thick frontend**, an **API Gateway** to publish & manage APIs used by client applications and **Backend as a Service** like databases and storage. In my application, **frontend** is written in **React JS** and **data is stored in DynamoDB**.

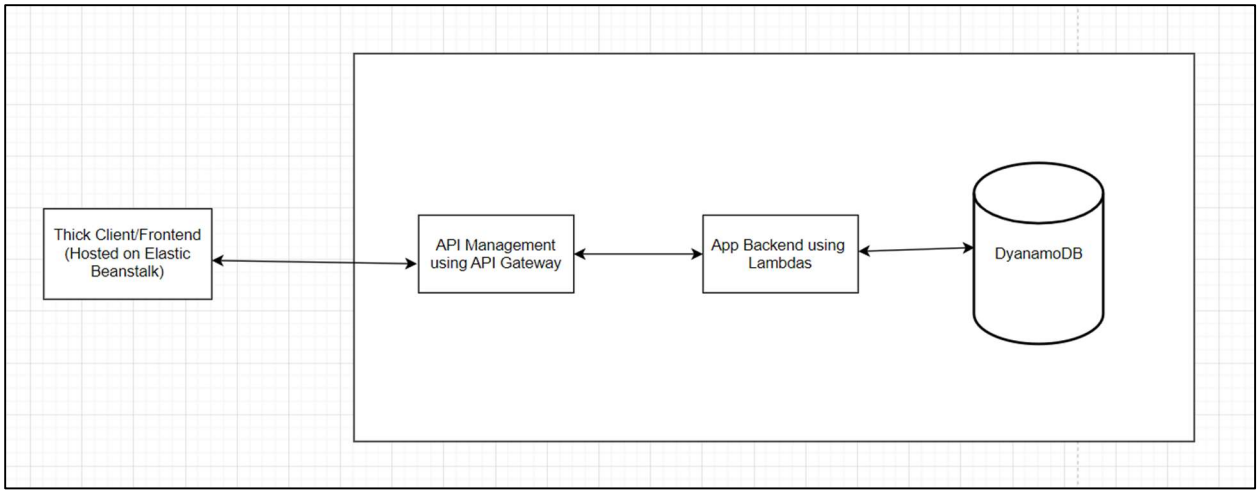

Figure 7 : Application Architecture.
Source : Adapted from [15] using draw.io.

Security Analysis

My frontend application is publicly hosted on AWS Elastic Beanstalk. However, incoming traffic can be controlled by changing the EC2 security groups settings, making it more secure and application security can be enhanced by enabling HTTPS protocol on the load balancer [4].

For the backend, the **security for data in transit is provided by AWS API Gateway** as any interaction to the AWS services including the Lambdas and DynamoDB is being done through APIs. API Gateway provides traffic management, authorization and access control, monitoring, and API version management. Additionally, API Gateway can verify signed API calls. Using custom authorizers written as AWS Lambda functions, API Gateway can also help verify incoming bearer tokens, removing authorization concerns from backend code[14]. Moreover, AWS Lambda function runs in its own isolated environment, with its own resources and file system view to provide security and separation at the infrastructure and execution levels. AWS **Lambda stores code in Amazon S3 and encrypts it at rest**. AWS Lambda performs additional integrity checks while the code is in use [2].

For **data at rest** , DynamoDB provides **encryption at rest by default**. It **uses AWS Key Management Service (KMS)** to manage the encryption keys[16].

Cost Analysis

The upfront, on-going, and additional costs to build this system have been calculated using the **AWS Pricing Calculator** (**Figure 8, 9 and 10**). For AWS Elastic Beanstalk, there is no additional charge. We pay for AWS resources (e.g., EC2 instances or S3 buckets) we create to store and run our application. There are no minimum fees and no upfront commitments. Accordingly, I have included cost estimates for an EC2 instance used in my AWS Elastic Beanstalk configuration.

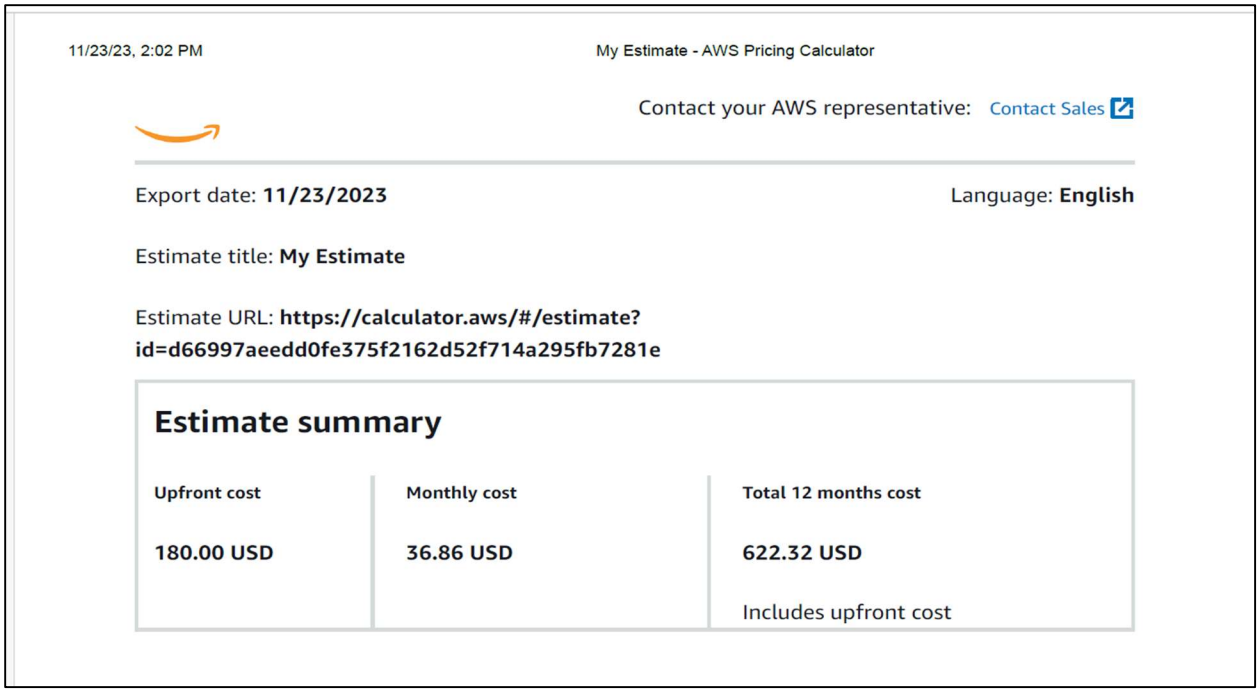

Figure 8: Estimate summary of the cost

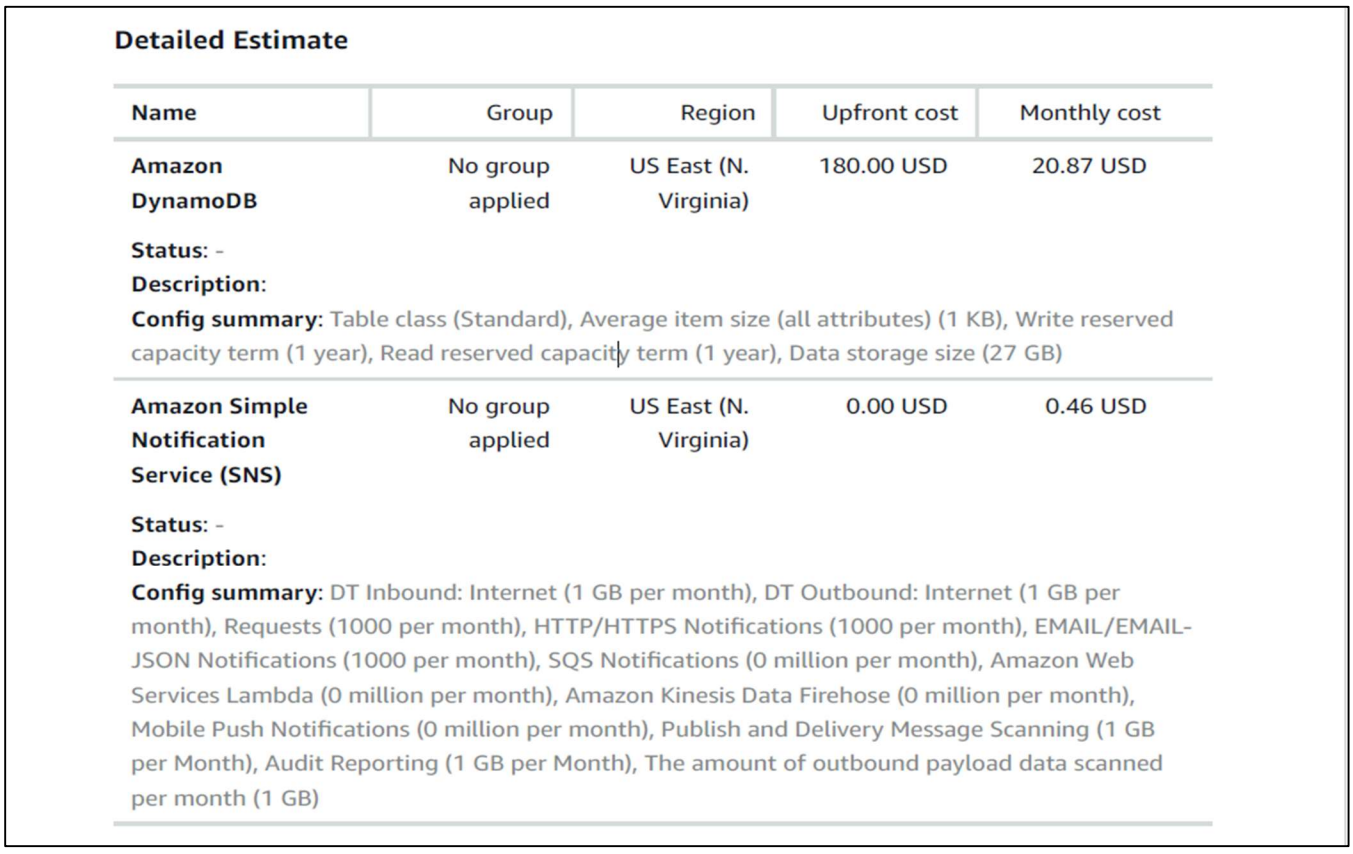

Figure 9: Detailed estimate (1/2)

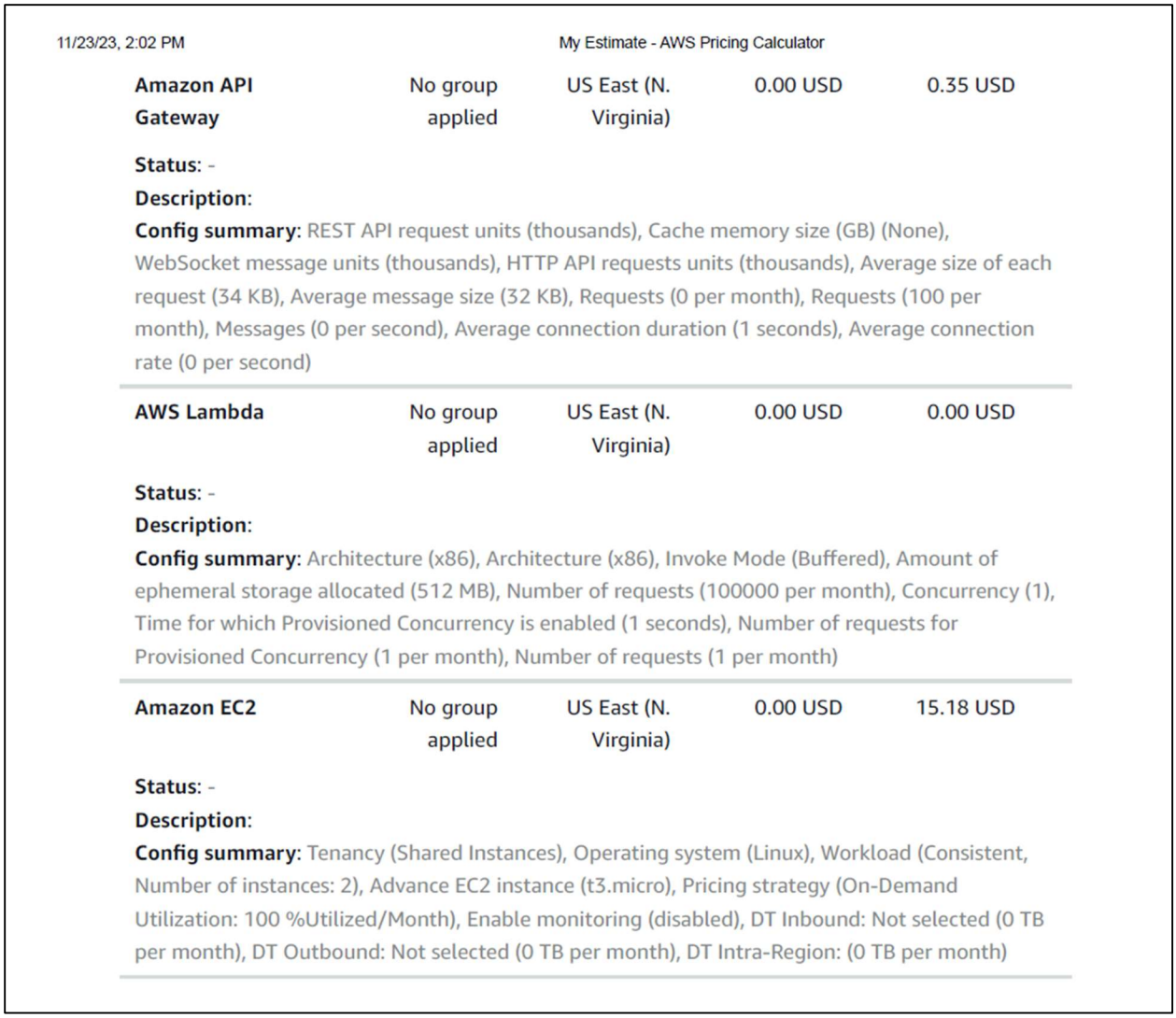

Figure 10: Detailed estimate (2/2)

There is no alternative approach for serverless and hence, Lambdas will have to be used which will incur costs only on the number of requests received. Also, for hosting my frontend, Beanstalk does not incur any additional costs but only for the EC2 instances. This cloud service will escalate the budget unexpectedly and hence, monitoring alerts need to be added using AWS Budgets.

An alternate approach for network would be to use VPC instead of API Gateway. However, it increases the cost manifold (**Figure 11 and 12**).

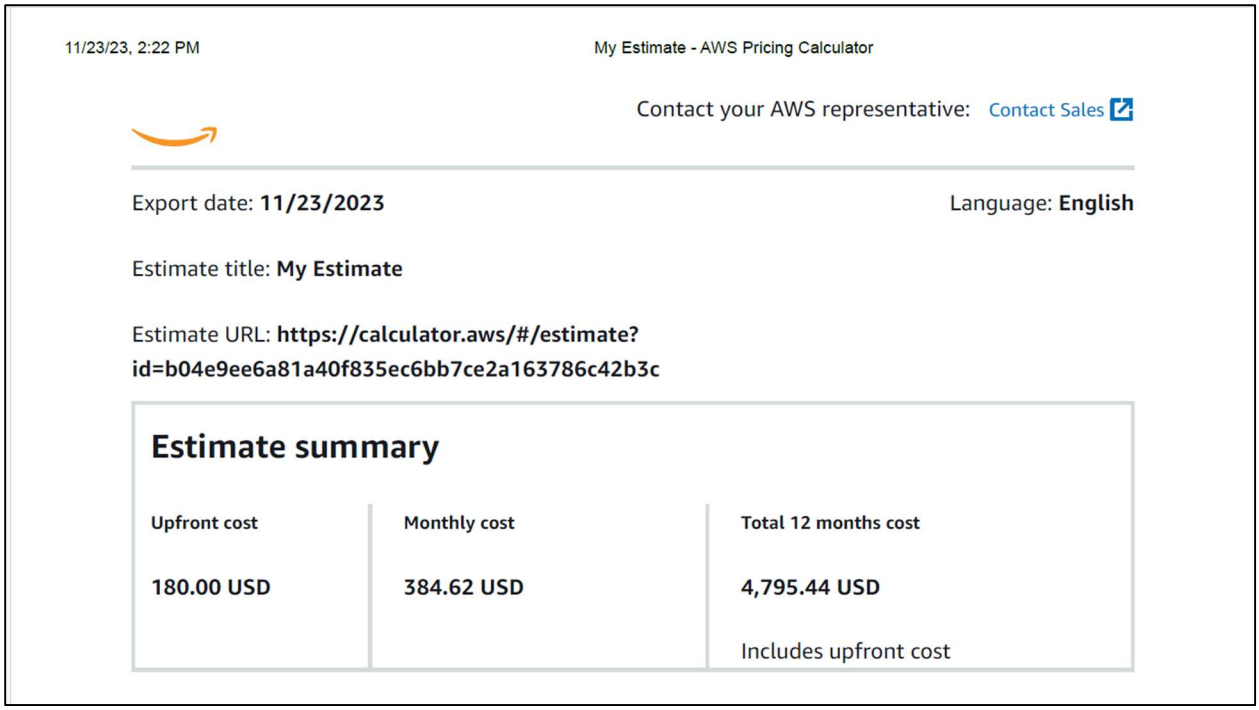

Figure 11: Estimate summary using VPC instead of API Gateway.

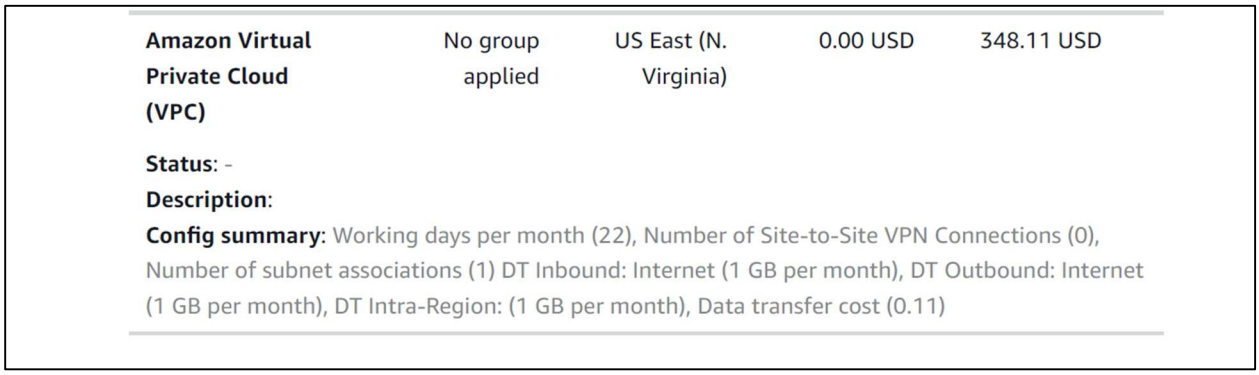

Figure 12: Detailed Summary for VPC

Given the design choices made and the options available, there is little scope of alternative approaches to the system used for the 4G Task Manager app.

Future Roadmap

The application will further be developed to have a user login so that each user can manage their tasks in isolation. Moreover, more functionalities will be added wherein user can add date and deadline of a task. The current design of cloud services considers these evolutions.

References

[1] Indeed Editorial Team, “The Covey Time Management Matrix Explained”, *Indeed* [Online]. Available:https://www.indeed.com/career-advice/career-development/covey-time-management-matrix,July 22,2022.[Accessed: November 10,2023].

[2] “AWS Lambda FAQs”, *AWS* [Online]. Available: https://aws.amazon.com/lambda/faqs/. [Accessed: November 17, 2023].

[3] Dr. Lu Yang (September 28, 2023), “Cloud Delivery Models”, CSCI 5409 Lecture, Dalhousie University. [PowerPoint slides].

[4] “AWS Elastic Beanstalk FAQs”, *AWS*[Online]. Available: https://aws.amazon.com/elasticbeanstalk/faqs/. [Accessed: November 17 ,2023].

[5] “AWS Step Functions FAQs”, *AWS*[Online]. Available: https://aws.amazon.com/step-functions/faqs/. [Accessed: November 17 ,2023].

[6] “Amazon DynamoDB FAQs”, *AWS*[Online]. Available: https://aws.amazon.com/dynamodb/faqs/. [Accessed: November 17 ,2023].

[7] “Amazon S3 FAQs”, *AWS*[Online]. Available: https://aws.amazon.com/s3/faqs/. [Accessed: November 17 ,2023].

[8] “What is AWS API Gateway”, *dashbird* [Online]. Available:https://dashbird.io/knowledge-base/api-gateway/what-is-aws-api-gateway/. [Accessed: November 17 ,2023].

[9] “Amazon SNS FAQs”, *AWS*[Online]. Available: https://aws.amazon.com/sns/faqs/. [Accessed: November 17 ,2023].

[10] “AWS Cloudformation FAQs”, *AWS*[Online]. Available: https://aws.amazon.com/cloudformation/faqs/. [Accessed: November 17 ,2023].

[11] Dr. Lu Yang (October 17, 2023), “Infrastructure as Code”, CSCI 5409 Lecture, Dalhousie University. [PowerPoint slides].

[12] Dr. Lu Yang (October 24, 2023), “Cloud Deployment Models”, CSCI 5409 Lecture, Dalhousie University. [PowerPoint slides].

[13] B.Kariuki, “Getting Started With AWS DynamoDB”, *Section*[Online]. Available: https://www.section.io/engineering-education/getting-started-with-aws-dynamodb/, January 08,2021. [Accessed: November 18,2023].

[14] “Amazon API Gateway FAQs”, *AWS*[Online]. Available: https://aws.amazon.com/api-gateway/faqs/. [Accessed: November 18,2023].

[15] Dr. Saurabh Dey(September 28,2023), “Serverless Data Processing”, CSCI 5410 Lecture, Dalhousie University. [PowerPoint slides].

[16] “DynamoDB encryption at rest”, *AWS*[Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/EncryptionAtRest.html. [Accessed: November 18,2023].

[17] “AWS Elastic Beanstalk Pricing”. *AWS* [Online]. Available: https://aws.amazon.com/elasticbeanstalk/pricing/. [Accessed: November 23,2023].