CSCI 5409 Adv Topics in Cloud Computing

Term Assignment Report

Prepared by:

Tejas Pabbu – B00961226

Masters of Applied Computer Science
Faculty Of Computer Science
Dalhousie University

GitLab Repo Link:
https://git.cs.dal.ca/courses/2024-winter/csci4145-5409/pabbu/

Table of Contents

V. Data Security

VI. Estimated Cost for On-Premise Reproduction

VII. Cost Monitoring

VIII. Future RoadMap

IX. References

Part I

Introduction

1. Introduction:

In the hustle and bustle of our daily lives, it's easy to lose sight of our dreams and aspirations. But what if you could receive a message from your past self, reminding you of your journey, your goals, and your potential for greatness?

Dream Dash: *Reviving Dreams, One Capsule at a Time*

Dream Dash presents a unique approach to personalized motivation and self-reflection through its innovative time capsule delivery system. At its core, Dream Dash seeks to bridge the gap between our past, present, and future selves, offering a tangible reminder of our aspirations and commitments at crucial junctures in our lives.

The premise is simple yet profound: users craft their goals and aspirations, and Dream Dash facilitates the delivery of capsules from their past selves in the form of personalized motivation messages. These capsules serve as poignant reminders of their journey, offering encouragement, guidance, and reflection precisely when needed most.

Crafted at the intersection of modern web technologies and AWS cloud services, Dream Dash is more than just a platform—it's a tool for empowerment. Harnessing the potent combination of technology and psychology.

1.1 Purpose of the Application

The primary feature of the application is the creation of personalized capsules, allowing users to compose messages tailored to their goals and aspirations. Users can specify delivery dates for these capsules, which are then automatically sent out as motivational reminders at the designated times. Additionally, the platform offers scheduling capabilities, enabling users to plan and organize their capsule deliveries efficiently. Furthermore, the application provides a user-friendly interface for managing capsule content, facilitating seamless customization and editing.

1.2 Technology Stack

The project utilizes a React-based frontend, offering dynamic and user-friendly interfaces across multiple pages including login, registration, and capsule creation. These frontend components communicate seamlessly with a Flask-based backend, responsible for user authentication, registration, and capsule creation functionalities. Additionally, the backend manages data interactions and orchestrates integrations with various AWS services for capsule delivery scheduling and management.

1.3 Target Users

The application targets individuals seeking a unique way to stay motivated and inspired on their journey towards success. It caters to users from diverse backgrounds including students, professionals, and individuals pursuing personal development goals. The application offers an intuitive and accessible interface, accommodating both tech enthusiasts and those less familiar with advanced digital platforms.

1.4 Performance Targets

The application is engineered to deliver a seamless, interactive user experience, prioritizing swift response times, particularly during the process of image analysis and transformation of those images to text. The objective is to render precise and valuable outcomes promptly to users, ideally in a matter of seconds. The system is also designed for robustness, targeting a high-availability benchmark to guarantee consistent, uninterrupted access for all users. Such performance metrics will underpin the application's reliability and user satisfaction.

1.5 Functionalities:

Table 1.1.: Feature Descriptions with Workflow

Features	**Description**
User Authentication	Users can Register and Sign In into the application .
Capsule Creation & Delivery	The capsule creation form allows users to specify the date and time of capsule delivery, select the occasion or purpose of the capsule, and customize a personalized message. An innovative feature enables users to upload motivational quotes as images. The system automatically extracts text from these images, ensuring the quotes are included in the capsule delivery without the need for manual typing.
Notifications	Capsule delivery is instantiated at the specified date and time for subscribed user via mail where user receives extracted message from the image with personalized message given from user at the time of capsule creation.

Part II

Requirement Analysis

2. AWS Services and its Alternatives:

2.1 Requirements in Assignment

As per the requirements mentioned in the term assignment description, I have used the following list of AWS services in different categories for various features of my web application:

Table 2.1.: AWS Services Usage

AWS Service Category	AWS Service Used	Count
Compute	AWS EC2 , AWS Lambda	2
Storage	AWS S3, AWS DynamoDB	2
Network	AWS Event Bridge	1
General	AWS Textract AWS SNS	2

2.2 AWS Service Mappings for Implemented functionalities

Frontend: AWS EC2 – (Login Page, Registration Page, Capsule Creation Page)
Backend: AWS EC2 – (Login module, Registration Module, Capsule Creation Module)
Serverless Backend: AWS Lambda – (Capsule Scheduling and Capsule Delivery)

Table 2.2: AWS Services Mappings with features

Features	Service Mapping's Justification
User Authentication	AWS DynamoDB is used for validation and storing the user information in User Registration Table.
Capsule Creation	AWS Textract is used for image-to-text conversion and AWS S3 for storage of provided images as backup and AWS DynamoDB for Capsule metadata storage.
Capsule Delivery	Based on AWS Event Bridge Rule created when new capsule is created. AWS Lambda gets triggered for delivery.
Notifications	AWS SNS is used for Capsule delivery at the specified date and time for subscribed user.

2.3 Comparison with Alternatives

Table 2.3: Comparison between chosen AWS Services with Alternatives

Services	Alternatives	Description
AWS S3	AWS EFS	AWS EFS (Elastic File System) provides a simple, scalable file storage for use with Amazon EC2 instances.
AWS DynamoDB	AWS RDS	AWS RDS (Relational Database Service) provides an alternative to DynamoDB for applications that require a relational database structure.
AWS SNS	AWS SQS	AWS SQS (Simple Queue Service) offers a secure, durable, and available hosted queue that lets you integrate and decouple distributed software systems and components. It can serve as an alternative to SNS for certain use cases.
AWS Lambda	AWS Batch	AWS Batch is a fully managed service by AWS for efficiently running batch computing workloads without managing infrastructure. It automates job scheduling, scaling, and management, ideal for tasks like data processing and scientific simulations.
Amazon Event Bridge	AWS Kinesis	AWS Kinesis provides real-time data streaming capabilities for ingesting, processing, and analysing large volumes of data. It's suitable for scenarios where you need to process and react to continuous streams of data in real-time

2.4 Reasons for Using Mentioned Services over alternatives

- **AWS S3 over AWS EFS:** I used S3 buckets for storing my image files processed by Amazon Textract due to its durability, scalability, and cost-effectiveness. S3 is purpose-built for object storage and ideal for handling image files EFS, while useful for certain use cases, may be overkill for simply storing small image files and could incur unnecessary costs. Moreover, EFS also provides scalable file storage, it is designed to be used with EC2 instances and may not provide the same level of accessibility and cost effectiveness as S3 when used to serve static content over the internet.

- **AWS Event Bridge over AWS Kinesis:** I opted against AWS Kinesis primarily due to its focus on real-time data streaming rather than event-driven workflows. While Kinesis excels in scenarios requiring real-time data processing and analytics, it involves more manual setup and management compared to EventBridge for event routing and integration across AWS services. Additionally, Kinesis may introduce additional complexity and operational overhead, especially for less experienced users, whereas EventBridge offers a more streamlined and intuitive approach to managing event-driven architectures.

- **AWS DynamoDB over AWS RDS:** I used DynamoDB for storing User registration and Capsule Metadata services due to its seamless scalability and fast, consistent performance. While RDS provides a relational database structure, it may not perform as well as DynamoDB under heavy load and may require complex queries to retrieve data. Moreover, RDS is a powerful relational database service, it may not provide the same level of performance for read-intensive workloads, and it may require more management overhead compared to DynamoDB.

- **AWS SNS over AWS SQS:** I used SNS mainly for its ability to send notifications to a large number of subscribers with customizable message, including distributed systems, end-user emails, SMS, and HTTP endpoints. The Capsule Delivery which is main feature is done via email medium on the specified date and time. While SQS is a robust queuing service for decoupling and scaling microservices, distributed systems, and serverless applications, it doesn't natively support message fan-out to multiple subscribers, making SNS a more suitable choice for my project's capsule delivery use case.

- **AWS Lambda over AWS Batch:** I selected AWS Lambda for its serverless architecture, which facilitates effortless scalability and cost-effectiveness for my project's event-driven tasks. Functions such as capsule delivery and event bridge rule triggers benefit from Lambda's automatic scaling, ensuring optimal performance during peak loads. Conversely, AWS Batch's container-based deployment would entail added management complexity and might not deliver equivalent scalability and cost-efficiency. Lambda's pay-per-use pricing model suits the variable workload of my application, making it the preferred option over Batch for my project Dream Dash.

Note: These rationales stem from my personal encounters and understanding of the mentioned services, which could differ in various contexts for different project use cases. However, I opted for these services for my Dream Dash project due to the outlined justifications.

2.5 Cost Comparison Table

Table 2.4: Comparison between chosen AWS Services with Alternatives in terms of pricing

Service	Pricing	Usage Cost per Month	Reason for Selection
User Authentication			
DynamoDB	Pay per provisioned read/write capacity units and storage	For "Read Request Units (RRU)": $0.25 per million read request unit. For "Write Request Units. (WRU)": $1.25 per millionwrite request units. For "Data Storage": First 25 GB storedper month is free using the DynamoDB Standard table class $0.25 per GB-month thereafter.	Scalable, low-latency, and cost-effective for user data storage
Amazon RDS	Pay per instance type and storage	Standard Instances - db.t4g.micro = $0.016 /hr General Purpose SSD storage (single AZ) (gp2) - Storage $0.115 per GB- month [12]	Overkill for simple user data, cost-effectiveness concerns

Image Upload and Storage			
AWS S3	Pay per storage and data transfer	S3 Standard General-purpose storage for any type of data,typically used for frequently accesseddata First 50 TB / Month $0.023 per GB [13]. Data Transfer IN = $0, Data Transfer OUT = $0, (since no data is transferred out)	Ideal for handling image files, durable and cost-effective
AWS EFS	Pay per storage and throughput	Effective storage Price ($0.043/GB-Month) - One Zone*[14]	Better for shared access to files, not primary requirement

Capsule Delivery			
AWS Lambda	Pay per request and duration	First 6 Billion GB – Seconds / month $0.000016667 for every GB-second $0.20 per 1M requests [16].	Suitable for event driven applications.
AWS Batch	Pay per vCPU and memory usage	There is no additional charge for Amazon Batch. You only pay for what you use, as you use it [17].	More suited for large-scale batch processing , not real-time

Event Driven Application			
AWS Event Bridge	Pay per number of events ingested and number of event schema discovery API calls.	Standard Event Ingestion rate is $1.00 per million events. Custom Event bus Ingestion is priced at $1.00 per million events plus $0.10 per million schema discovery API calls.	Easy event routing and integration across AWS services
AWS Kinesis	Pay based on shard hours and PUT payload units.	Kinesis Data Streams cost is $0.015 per shard hour, & PUT payload unit pricing starts at $0.014 per PUT payload unit.	Focuses on real-time data streaming rather than event-driven workflows.

Notification Handling			
AWS SNS	Pay per Email send	Email/Email-JSON - $2.00 per 100,000 notifications [11].	Simplicity, direct communication with users
AWS SQS	Pay per request	First 1 million requests/month - FREE	Suitable for message queuing, but not ideal for real-time notifications

Part III

Delivery and Deployment Models

3. Delivery & Deployment Model

3.1 Deployment Model

For the Dream Dash project, selecting a Hybrid Cloud deployment model offered a suite of compelling benefits that are particularly aligned with its goals and operational blueprint of the project:

- **Data Management and Compliance**: The application demands stringent data management practices due to its existing security vulnerabilities. By utilizing a hybrid cloud, I ensure that sensitive information is secured on-premises or within a private cloud, adhering to compliance mandates and industry standards, while still being able to harness the computational power of the public cloud for less sensitive operations.

- **Customized Security Posture**: Recognizing the sensitive nature of data encapsulated in a capsule, I am planning to maintain a customized security posture with a hybrid model.

- **Controlled Scalability:** Although the current implementation doesn't employ a load balancer, it's imperative to provision for scalability. The hybrid model is agile, granting me the ability to scale resources in the public cloud segment while safeguarding core functions internally. This enables me to meet varying workloads and spikes in trafficwithout compromising on security or incurring unnecessary costs

- **Cost-Optimization:** Balancing costs without compromising on service quality is crucial. I find that a hybrid cloud deployment is still feasible for my application. By carefully managing resource allocation and leveraging the pay-as-you-go pricing model of the public cloud, I can optimize costs while benefiting from the advantages of cloud computing, such as easy provisioning and dynamic resource allocation.

- **Integration and Interoperability:** My application may need to integrate with existing systems that are kept on-premises for various reasons. Choosing a hybrid cloud model affords seamless integration and interoperability between cloud-based services and on-premises infrastructure, facilitating a cohesive ecosystem.

- **Performance and Latency:** Given my project is not currently geographically distributed, the hybrid model allows me to optimize for performance by hosting the application in proximity to my user base, reducing latency that can occur with fully public cloud services located in distant regions.

- **Technology Stack Alignment:** The current tech stack of AWS services, as above, showcases a blend of AWS services that support both on-demand scalability and dedicated resource control. The hybrid model complements this by allowing me a selective placement of services within the public or private sphere, in alignment with technological needs and organizational strategy.

In summary, the decision for a Hybrid Cloud deployment model is a strategic choice that aligns with the project's operational, security, and financial prerogatives. It allows me to address my specific security concerns, efficiently handle potential growth, and make the most of cloud resources while maintaining a level of control that aligns with my application's unique needs.

3.2 Delivery Model

I have decided to adopt the **SaaS** delivery model for the Dream Dash Time Capsule application, as it resonates with my aim to offer a seamless, robust, and economically viable digital time capsule service. As the developer, my concentration is on providing an unparalleled experience to users who wish to capture and preserve their memories, without the added complexity of handling infrastructure and ongoing maintenance.

Opting for the SaaS model permits me to capitalize on the expertise and dependability of the cloud service provider, ensuring that Dream Dash is consistently updated, secure, and available whenever users wish to access their cherished memories . The intrinsic scalability of the SaaS framework offers the agility needed to manage growing user interest in the application, guaranteeing that Dream Dash can sustain increased activity during notable periods when users are most likely to create time capsules, such as holidays or significant personal milestones. Furthermore, the cost efficiency of the SaaS model empowers me to curtail expenditures and judiciously distribute resources. This means that rather than allocating funds to physical servers and software licenses, I can invest more into enhancing Dream Dash's functionalities and user engagement strategies.

Considering the nature of the cloud application, the SaaS delivery model aligns well with the goals of providing image analysis capabilities to a broad range of users. The model ensures easy accessibility, continuous updates, and a cost-effective solution for users while allowing the development team to maintain and improve the application efficiently.

Reasons for Choosing SaaS Delivery Model

- **Ease of Access and User Convenience**: The chosen SaaS delivery model excels in providing superior access and user convenience. With an internet-enabled device and web browser, users can effortlessly engage with the application, negating the need for local installations and enabling ease of access from any location.

- **Swift Deployment and Updates**: The central hosting of SaaS application streamlines the deployment of updates and new features. This centralization removes the need for user-end update distribution, leading to a seamless evolution of the application and an enriched user experience.

- **Inherent Scalability**: The infrastructure's scalability is seamlessly managed by the SaaS provider. This allows for an increase in the user base without any interruption to service or the need for hands-on scaling procedures.

- **Economical Approach**: Embracing the SaaS model translates into cost savings for both provider and users. Providers can realize economies of scale by allocating resources across a broader user base, while users benefit from the elimination of initial investment costs, paying only for their actual usage.

- **Maintenance and Technical Support**: Responsibility for application upkeep, including routine maintenance and addressing technical concerns, falls to the SaaS provider. This alleviates users from the burdens of software maintenance, freeing them to concentrate on leveraging the application for their personal or professional use.

- **Centralized Security and Data Safeguarding**: The application's security architecture is centralized within the SaaS framework, ensuring comprehensive protection of user data. Trust in the application's ability to handle sensitive information is bolstered by the proactive monitoring and threat response mechanisms.

- **Worldwide Reach**: The web-based nature of the SaaS application means that geographical boundaries do not restrict access, allowing a global audience to utilize the application, thereby expanding its reach and potential market [21].

Rationale for Inclusion of Hybrid Components in Application

My application architecture incorporates some hybrid components , which involve use of EC2 and S3 and there are specific benefits which my application would receive from the integration of EC2 and S3 which are :

1. **Modular Scaling and Control**: EC2 delivers the necessary control and flexibility over the computational environment, ensuring the application's modules can be scaled in response to user demand while maintaining optimal performance.

2. **Targeted Storage Solutions**: S3 provides a robust platform for storing diverse data types, such as multimedia content and analytical outputs, ensuring high availability and cost-efficiency in data handling.

Though the core of the application is provided as SaaS, the inclusion of EC2 and S3 plays a pivotal role in bolstering the application's infrastructure. These components add vital IaaS and Storage-as-a-Service (STaaS) capabilities that support the SaaS model, leading to an integrated solution that excels in processing and managing data within a responsive and scalable cloud architecture.

Scope of Migration towards FaaS Delivery Model

While the Dream Dash Time Capsule project currently operates on a Software as a Service (SaaS) model, which is well-suited for delivering a fully integrated service to end-users, there is scope for incorporating Function as a Service (FaaS) for specific backend functionalities. FaaS would not replace the SaaS model but would complement it by providing additional benefits for particular services. Here are some advantages the project could realize from integrating FaaS:

- **Cost Savings on Compute Resources**: FaaS operates on a pay-per-use basis, which means the project would only incur costs when functions are executed in response to events, potentially reducing overhead during periods of low activity.

- **Enhanced Scalability for Event-Driven Functions**: For functionalities like sending notifications or processing user-generated content, FaaS can offer instantaneous and automatic scaling, ensuring that these services can handle high loads without manual intervention or pre-provisioning.

- **Increased Development Efficiency**: Developers can focus on writing individual functions for specific tasks without worrying about the underlying server infrastructure, leading to faster deployment of new features and updates.

- **Simplified Operational Overhead**: With FaaS, the burden of server maintenance, patching, and management is shifted to the cloud provider, allowing the Dream Dash team to concentrate on innovation and improving application functionality.

- **Improved Fault Tolerance**: Cloud providers typically ensure high availability and fault tolerance for FaaS, enhancing the reliability of the services it powers within Dream Dash without additional complexity [22].

3.3 Conclusion

Although the Dream Dash Time Capsule project could benefit from a shift towards a Function as a Service (FaaS) model for certain aspects, it's important to acknowledge that the Software as a Service (SaaS) delivery model remains ideal for the overarching framework of this project. The SaaS model provides a cohesive and comprehensive user experience, offering the full functionality of the time capsule service directly to end-users.

Nevertheless, integrating FaaS into specific services within the backend infrastructure can harness the strengths of both models. Services that could particularly benefit from FaaS include event-triggered notifications, media processing for user-uploaded content, and on-the-fly analytics. By selectively applying FaaS to these components, the Dream Dash project can capitalize on the scalability and cost-effectiveness of serverless computing, while the SaaS framework continues to deliver a robust, user-centric product. This dual approach allows for a harmonious balance between maintaining a seamless end-user experience and optimizing backend processes

Part IV

Final Architecture

4. Final Architecture

4.1. Architecture Diagram

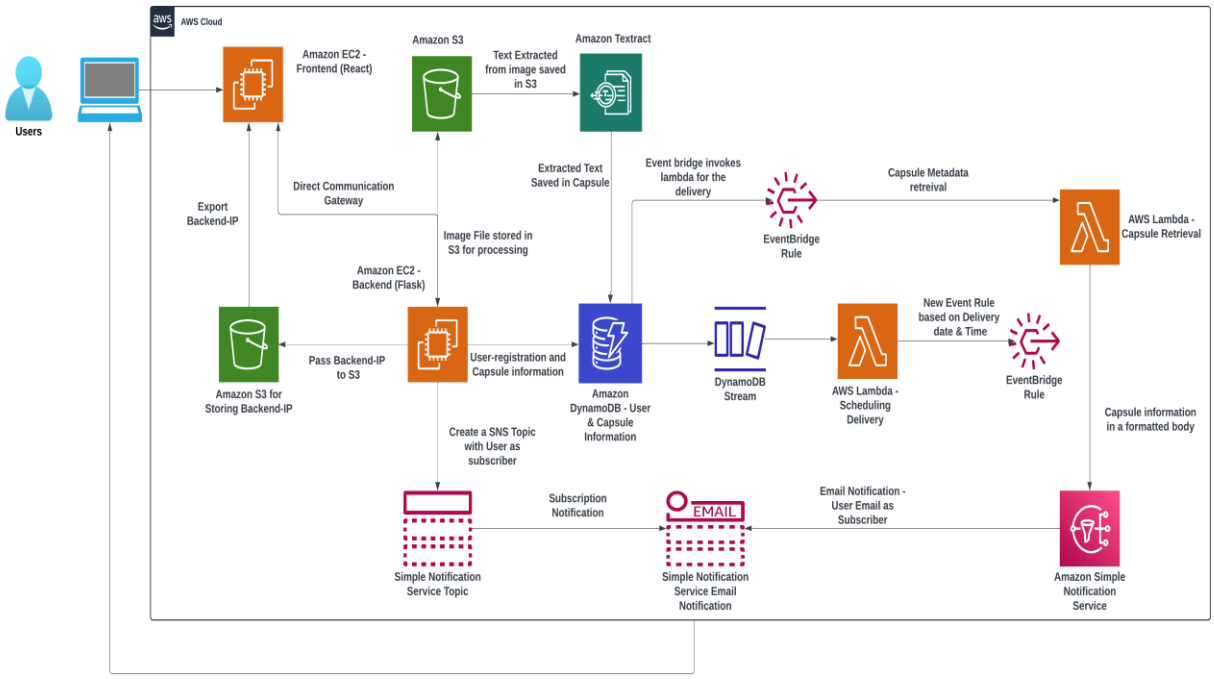

Figure 4.1: AWS Architecture Diagram of Dream Dash Application [20]

4.2. Cloud Mechanisms and Integration Fit

Based on the above provided architecture for the Dream Dash Time Capsule project, here's an overview of each service used, its purpose, and how it fits into the project's integration:

- **Amazon S3**: Amazon S3 provides reliable, scalable object storage and is utilized for storing user-uploaded image content for AWS Textract processing and backup purposes. It ensures data durability and easy accessibility, serving as the backbone for user content management in Dream Dash. Furthermore, it is also used to store Backend-IP addresses of the EC2 instance which is retrieved by Frontend EC2 for creating a communication gateway between them.

- **Amazon EC2**: Amazon EC2 instances host the core application logic of Dream Dash. They provide the computational environment for the frontend React application and backend Flask services, ensuring responsive user interactions and efficient processing of backend tasks.

- **Amazon DynamoDB**: Serving as the primary database, DynamoDB stores user profiles with credentials, time capsule records with specific delivery dates, and turns on Dynamo DB stream accessibility for event triggers. Its low-latency and high-performance capabilities support the application's needs for quick data retrieval and updates, vital for a seamless user experience.

- **AWS Lambda**: These serverless compute resources execute code in response to events, such as scheduled time capsule openings or user notifications. AWS Lambda functions integrate with other AWS services to perform tasks without the need for a continuously running server, optimizing resource use and reducing operational costs.

- **Amazon Textract**: Used for processing and extracting text from uploaded images or documents within the time capsules. Textract's OCR capabilities automate the data extraction process, enhancing the time capsule's content management and searchability features.

- **Simple Notification Service (SNS)**: Amazon SNS manages the delivery of notifications related to time capsule events. It handles the distribution of messages across various channels, ensuring that users receive timely updates about their time capsules.

- **EventBridge (formerly CloudWatch Events)**: This service detects and reacts to changes in the AWS environment. In Dream Dash, EventBridge schedules and triggers events that initiate time capsule-related workflows, like the release of a capsule's content on a specified date.

Each of these services is integral to Dream Dash, collectively offering a robust, scalable, and efficient cloud-based environment that underpins the time capsule experience. They are carefully orchestrated to ensure that the application remains responsive, cost-effective, and capable of handling the varied demands of a dynamic user base.

4.3. Data Storage:

My Dream - Dash application data is stored in 3 main locations:

- **Textract S3 Bucket:** All my images which contains motivational quotes and which needs processing from AWS Textract are first saved into a S3 bucket after the user submits the request to for Capsule Creation

- **User Registration DynamoDB Table:** All the User Profile information like name, email and password for the specific user will be saved in this table for validating users when they login again to the application.

- **Capsule Metadata DynamoDB Table:** Information related to Time Capsule which includes occasion/purpose of the capsule , the ARN of the subscribed SNS Topic and most importantly the delivery date and time of the capsule information is stored in the table , which then will be retrieved by Capsule Delivery Lambda at the time of delivery of capsule

4.4. Choosing Right Programming Language

- **Backend**: The backend of the application is implemented using Flask, a Python-based web framework. Python is chosen for its simplicity, ease of use, and extensive libraries, making it suitable for developing the application's server-side logic and integrating with AWS services.

- **Frontend**: The frontend is developed using React.js, a popular JavaScript library known for its flexibility, component-based structure, and efficient rendering. React.js enables the creation of dynamic and interactive user interfaces, enhancing the user experience.

- **Integration:** The Fetch API in the Dream Dash project facilitates communication between the frontend and backend hosted on EC2 instances. It enables asynchronous HTTP requests to retrieve or send data, ensuring a smooth user interaction by exchanging information with the server without reloading the webpage. This integration is key to providing a responsive and seamless experience as users navigate the time capsule functionalities.

4.5. System Deployment on the Cloud

I will deploy this Dream Dash web application to the cloud using the services provided by Amazon Web Services (AWS). The deployment process may include the following steps:

- **Code Deployment:** Docker images are utilized for both the frontend and backend components of the application code deployment. These Docker images facilitate the containerization of the application, ensuring consistency and portability across different environments. The Docker images are hosted on public repository in my personal docker account, allowing for easy management and distribution of the application containers.

- **Serverless Deployment:** Lambda functions for capsule delivery scheduling and capsule delivery via SNS are triggered automatically by new record insertion in AWS DynamoDB Table, which is Capsule Metadata Table in my case through DynamoDB Streams Service. These Lambda's, along with their dependencies are packaged into deployable units.

- **AWS Infrastructure Setup:** AWS CloudFormation is used to define the application's infrastructure as code. This includes setting up EC2 instances for frontend and backend for handling user authentication and capsule creation with automatic code deployment into EC2 instances at boot time through EC2 instance's User Data Script, creating DynamoDB tables and S3 buckets, and Lambda functions and SNS triggers.

4.6. Architecture Comparison and Rationale

The architecture that most resembles my Dream Dash Time Capsule project from the architectures taught in the coursework is the **Dynamic Scaling Architecture**.

Here's why this fits my project:

- **Automated Scaling**: The use of AWS Lambda for Capsule Delivery Scheduling and Delivery through SNS suggests a system designed for dynamic horizontal scaling. Lambda functions scale automatically, both horizontally (by running more instances of your functions) and vertically (as individual functions can automatically handle more requests), akin to the dynamic scaling listener described in the architecture.

- **Cost-Effectiveness**: Lambda and other managed services like Amazon S3 and Amazon DynamoDB are billed based on usage, not pre-provisioned capacity, which aligns with the cost-effective aspect of the Dynamic Scaling Architecture. I only pay for the compute time and storage you use, with no costs associated with idle resources.

- **Load Balancing Not Central**: Although there isn't an explicit mention of a traditional load balancer in your setup, the use of AWS services inherently distributes the load. For example, S3 and DynamoDB automatically handle incoming requests, distributing them across their vast networks to manage the load.

- **Event-Driven and Horizontal Scaling**: My architecture is designed to respond to events (via EventBridge) and scales based on those, which fits within the concept of dynamic scaling where resources are adjusted as needed.

The architecture of my project does not involve fixed pools of resources (excluding the Resource Pool Architecture), does not specifically mention cloud bursting or redundant storage devices as primary features, and does not rely on persistent disk provisioning strategies. Instead, it focuses on responsive scaling and management of computing resources and storage, central to the Dynamic Scaling Architecture.

4.7. Conclusion

My cloud application architecture is well-designed by combining the advantages of serverless computing with EC2 instances for frontend and backend hosting. It effectively leverages the dynamic scalability architecture with AWS Lambda for Capsule Delivery Scheduling and Delivery. The usage of Amazon S3 for object storage ensures redundancy and data durability, even though it does not strictly align with the elastic disk provisioning architecture. Overall, my choices for the architecture are justified, as the architecture optimizes performance, cost, and user experience for the image analysis application.

Part V

Data Security

5. Data Security

5.1. Current security in the application

In the architecture of my Dream Dash application, data security is well taken care to a limit. I have deployed a range of strategies to fortify the protection of data across the application like:

- **Authentication**: User authentication is handled through secure processes where passwords are hashed before storage, ensuring that even if data storage is compromised, the actual passwords remain protected.
- **Encrypted Communications**: To maintain the integrity and confidentiality of data as it moves between the mobile app and backend services, HTTPS is used for all communications. This encryption protocol guards against interception and maintains the privacy of data in transit.
- **Access Control**: Adhering to the principle of least privilege, I've structured AWS IAM policies to restrict access rights. Users and services are endowed with only the essential permissions they require, significantly limiting the potential damage in the event of a security breach.
- **Isolated Lambda Environments**: AWS Lambda provides a secure execution environment for serverless functions. Each Lambda function runs in an isolated container, ensuring that code from one function cannot access data from another function.

5.2. Data Security Vulnerabilities & Resolution in Architecture

Table 5.1: Security Concerns in Application with appropriate resolution

Security Concern	Security Concern Reason	Vulnerability	Resolution
Public Buckets	One notable security concern would be the presence of public buckets in my application which is Textract bucket used for storing images provided by user.	Making buckets Public can expose sensitive data to unauthorized access, making it a potential target for malicious actors. Public buckets can be accessed by anyone with bucket's URL, leading to potential data leaks and unauthorized data modification	To address this vulnerability, it is crucial to avoid making buckets public whenever possible. Instead, implement fine-grained access control using AWS IAM (Identity and Access Management) and bucket policies.

Exposed Backend	My frontend is exposed to the internet while the backend is only accessible to the frontend EC2 instance, there may still be potential risks.	If a malicious actor gains access to the frontend EC2 instance, they could potentially leverage it to attempt unauthorized access to the backend.	Implement additional security layers, such as API authentication mechanisms between the frontend and backend EC2 instances. Additionally, I can consider using a (VPC) to isolate the backend EC2 instance from the internet, allowing access only through private IP addresses or VPN connections.
DDOS Attack	DDoS attacks can overwhelm my application with traffic, potentially causing downtime and service disruption, which can erode user trust and incur financial loss.	A potential vulnerability to DDoS attacks could be the direct communication gateway from internet to the Amazon EC2 instances. These attacks could overwhelm the servers by flooding them with traffic, leading to denial of service for legitimate users.	I can plan to leverage AWS Services which offers DDOS Protection for my application. Few of AWS services which provide the above services are AWS Shield and AWS WAF to mitigate such attacks.

.

Part VI

Estimated Cost for On-Premise Reproduction

6. Cost Estimation

To reproduce my Dream Dash web application architecture in a private cloud, I would need to carefully consider required hardware and software components with software licenses to achieve a similar level of availability as my cloud implementation. The exact costs can vary based on various factors, I will provide a rough estimate which an organisation need to purchase to achieve a secure, scalable, and high-performing on-premise environment for the application.

6.1 Tools Used for Cost Estimation

The AWS Pricing Calculator allows you to estimate the cost of AWS services based on your usage. While you may not be deploying in the cloud, it can give you an idea of the costs associated with different services, which can help in comparing with on-premise solutions. You can find it on the official AWS website.

6.2 Hardware and Software Requirements

Hardware : For assessing the costs of servers and various hardware elements necessary for an on-premises setup, visit the official websites of hardware suppliers such as Dell, HP, and Lenovo. These sites typically offer tools for customization, enabling you to tailor hardware to your needs and get an estimated price for your specified configurations.

Software Licensing Costs : I would reach out to the respective software vendors (e.g., Docker, image-to-text processing software providers) to inquire about their licensing costs for the products. They usually provide pricing information on their websites or through their sales representatives.

Object Storage Costs : I would need on-premise storage solutions, such as Network-Attached Storage (NAS) or Storage Area Network (SAN), to store static product images, user data, and other application-related files.

Database Server Costs : I would require a dedicated database server to host the data stored in my application. Depending on the database technology used, I might need to purchase licenses for commercial database software or consider open-source alternatives. I'd need a NoSQL database system to replace DynamoDB. There are open-source options like Apache Cassandra, but they come with significant setup, maintenance, and scaling challenges.

Serverless Frameworks: For replicating the serverless features of AWS Lambda on-premise, you can explore open-source serverless frameworks like OpenFaaS, Knative, or Apache OpenWhisk [18].

Networking Equipment: A robust and redundant networking infrastructure is crucial for ensuring seamless communication between different components of the application. This includes network switches, routers, firewalls, load balancers, and other networking equipment.

Notification System: Replacing SNS would require a messaging system like RabbitMQ or an email service. These can be free or have small costs but need maintenance.

6.3 My Estimation Report Analysis

To estimate the upfront and ongoing costs for running my application on-premise with 10-100 requests a day, we will consider the following components:

1. Upfront Costs for servers and networking equipment : Servers for Frontend and Backend: Cost will vary based on the specifications and quantity of servers we purchase. Let's assume that we will buy the server: E3-1240 v5 (3.50 GHz), 64GB RAM, 1000GB SSD, with an estimated upfront cost of $2,748.24 for the required servers. Routers, switches, load balancers, firewalls, and other networking gear would cost another $2,000 for the required network connections and initial setup.
2. Ongoing Costs (Monthly):
    a) Power and Cooling: Let's assume an estimated monthly cost of $500 for power and cooling for the servers.
    b) Internet Connectivity: Let's assume an estimated monthly cost of $200 for internet connectivity to support the application's traffic.
    c) Amazon Textract License: Textract's pricing can be as low as $0.015 per page for extracting text from tables and $0.05 per page for forms. To give you an idea, processing 5,000 pages of documents with both tables and forms could cost around $325 for the month [15].
    d) MongoDB database server -$57/month for a Dedicated server [10].
    e) Email: SNS- First 1 million Amazon SNS requests per month are free, $0.50 per 1 million requests thereafter. Hence, no cost for those 300,000 requests/month. [11]
    f) Serverless cost - entirely depends on no. of requests, time of execution and memory allocated. So, cannot give an estimate.
3. *Total cost = **5,830.24** (approx.. for the 1st month only, including upfront and monthly costs)*

Note: It's essential to acknowledge that forecasting the expenses for an on-premises setup can be complex, and can fluctuate based on numerous elements such as hardware specs, the terms of software licenses , energy consumption, and operational expenditures. Unlike cloud services which typically operate on a pay-as-you-use basis, incurring costs for an on-premises infrastructure usually requires a significant initial outlay.

Part VII

Cost Monitoring

7. Monitoring AWS Lambda for Cost Control

Cloud mechanism which I feel would be most important to add monitoring to control costs is AWS Lambda because of the following below reasons:

1. **Performance Optimization**: Monitoring Lambda functions can provide insights into their performance, allowing you to optimize code for better efficiency and faster execution times, which can also impact costs.

2. **Error Tracking**: Lambda functions can fail silently. Monitoring can help in identifying, diagnosing, and alerting when errors occur so that they can be addressed promptly to maintain application reliability.

3. **Resource Utilization**: By monitoring, you can track metrics like memory usage and execution duration, which can help in fine-tuning the allocated resources for each function to match usage patterns without over-provisioning.

4. **Security**: Monitoring access and execution patterns can help detect and respond to unauthorized or malicious use of your Lambda functions [8].

Justification:

In the given architecture, the AWS Lambda service, particularly the functions responsible for 'Scheduling Delivery' and 'Capsule Retrieval,' holds the most potential to escalate costs unexpectedly. Lambda pricing is based on the number of requests for my functions and the time your code executes, which scales with the size of the workload. If the application experiences a sudden increase in usage or if there's an inefficient loop within the code, it could trigger a vast number of Lambda invocations or lengthy execution times, resulting in higher costs.

Furthermore, AWS Lambda is event-driven, which means that it works in concert with Amazon DynamoDB Streams and Amazon S3 triggers. An increase in the write/read operations on DynamoDB or the number of objects stored and retrieved from S3 can lead to an uptick in Lambda executions, thereby increasing costs. These interactions exemplify the cascading effect one service can have on another within a cloud architecture, highlighting the importance of closely monitoring Lambda alongside DynamoDB and S3 for cost optimization.

It's essential to implement monitoring and set up alarms using services like AWS CloudWatch to track the function's execution and control the associated costs effectively. By monitoring these services, I can not only avoid bill shocks but also optimize the performance and cost-efficiency of your application.

Part VIII

Future RoadMap

8. Future Enhancement in the Application

To further develop the Dream Dash capsule delivery project, I would envision the application evolving in several ways:

1. **Text-to-Speech Feature with AWS Polly**: To add a more personalized touch, I would integrate AWS Polly to allow users to add voice messages to their capsules. Polly's lifelike text-to-speech service could enable users to record messages in their chosen voices, which can then be played back when the capsule is opened.

2. **User Authentication with AWS Cognito**: To enhance security and provide a seamless user experience, AWS Cognito would be used for user authentication. Cognito would allow the application to offer user sign-up, sign-in, and access control, including options for social sign-on and multi-factor authentication.

3. **Image and Sentiment Analysis with Amazon Rekognition and Amazon Comprehend**: These services could analyze the content of the images and text for sentiment, providing insights into the emotional tone of the capsules. Users could get a summary of the overall sentiment of their reflections over time.

4. **Interactive Time Capsule with Amazon Lex and Amazon Connect**: By using Amazon Lex, the application could provide an interactive chatbot for users to interact with their capsule content. Amazon Connect could even enable phone-based interactions, allowing users to add to or access their capsules via a conversation.

5. **Enhanced Data Visualization with Amazon QuickSight**: To give users an overview of their journey, QuickSight could be utilized to create visualizations and insights from the data collected over time, such as frequency of capsule creation, types of quotes included, and sentiment trends.

6. **Personalized Recommendations with Amazon Personalize**: This service could analyze users' interactions and content to recommend personalized quotes or content for future capsules, enhancing user engagement.

7. **Expanded Notification Services with Amazon Pinpoint**: Beyond just email, Amazon Pinpoint could manage a broader set of user notifications, including SMS and push notifications, to alert users about their capsule's status or remind them to create new ones.

8. **Collaborative Features using AWS AppSync**: For shared capsules or memories, AWS AppSync could facilitate real-time updates across user devices, allowing friends or family members to collaboratively contribute to a shared capsule.

These features, built on AWS's robust suite of cloud services, would significantly enrich the user experience by making the capsules more interactive, personalized, and secure, potentially increasing user engagement and retention.

Part IX

References

References

[1] Amazon Web Services, ”AWS::S3::Bucket Properties,” *AWS CloudFormation User Guide*, [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws- properties-s3-bucket.html, [Accessed: April 8, 2024].

[2] Amazon Web Services, ”AWS::EC2::Instance Properties,” *AWS CloudFormation User Guide*, [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-ec2-instance.html, [Accessed: April 8, 2024].

[3] Amazon Web Services,” AWS::DynamoDB::Table Resource,”*AWS CloudFormation User Guide*, [Online]. Available: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource- dynamodb-table.html, [Accessed: April 8, 2024].

[4] Pallets Projects, ”Flask Quickstart,” *Flask Documentation*, Version 2.3.x, [Online]. Available:https://flask.palletsprojects.com/en/2.3.x/quickstart/, [Accessed: April 8, 2024].

[5] React, ”Learn React,” *React* [Online]. Available: https://react.dev/learn, [Accessed: April 8, 2024].

[6] Stack Overflow, ”How to Enable CORS in Flask?,”, *Stack Overflow* [Online]. Available: https://stackoverflow.com/questions/25594893/how-to-enable-cors-in-flask, [Accessed: April 8, 2024].

[7] Amazon Web Services, ”Serverless Architectures with AWS Lambda,” *AWS*, [Online]. Available: https://aws.amazon.com/lambda/serverless-architectures-learn-more/, [Accessed: April 8, 2024].

[8] Ernesto Marquez, ”Use These Tools to Keep your AWS Lambda Cost Under Control,” *Concurrency Labs*, [Online]. Available: https://www.concurrencylabs.com/blog/aws-lambda-cost-optimization-tools/ , [Accessed: April 8, 2024].

[9] Amazon Web Services, ”What Is Infrastructure as a Service (IaaS)?,” *AWS*, [Online]. Available: https://aws.amazon.com/what-is/iaas/, [Accessed: April 8, 2024].

[10] MongoDB, ”MongoDB Pricing,” *MongoDB* , Online: https://www.mongodb.com/pricing , [Accessed: April 8, 2024].

[11] Amazon Web Services, ” Amazon SNS Pricing,” *AWS*, Online: https://aws.amazon.com/sns/pricing/, [Accessed: April 8, 2024].

[12] Amazon RDS MySQL Pricing: Amazon Web Services. ”Amazon RDS MySQL Pricing,” *AWS Pricing*, [Online]. Available: https://aws.amazon.com/rds/mysql/pricing/?pg=pr&loc=2, [Accessed: April 8, 2024].

[13] Amazon S3 Pricing: Amazon Web Services. ”Amazon S3 Pricing,” *AWS Pricing*, [Online]. Available: https://aws.amazon.com/s3/pricing/?nc=sn&loc=4, [Accessed: April 8, 2024].

[14] Amazon EFS Pricing: Amazon Web Services. ”Amazon EFS Pricing,” *AWS Pricing*, [Online]. Available: https://aws.amazon.com/efs/pricing/, [Accessed: April 8, 2024].

[15] Amazon Textract Pricing. ”Amazon Textract Pricing,” *AWS Pricing*, [Online]. Available:https://aws.amazon.com/textract/pricing/, [Accessed: April 8, 2024].

[16] AWS Lambda Pricing: Amazon Web Services. ”AWS Lambda Pricing,” *AWS Pricing*, [Online]. Available: https://aws.amazon.com/lambda/pricing/, [Accessed: April 8, 2024].

[17] AWS Batch Pricing: Amazon Web Services. ”AWS Batch Pricing,” *AWS Pricing*, [Online]. Available: https://aws.amazon.com/batch/pricing/ , [Accessed: April 8, 2024].

[18] StackShare. ”Apache OpenWhisk vs. OpenFaaS - Comparison,” *StackShare*, [Online]. Available: https://stackshare.io/stackups/apache-openwhisk-vs-openfaas, [Accessed: April 8, 2024].

[19] Nirmal Ganesh Yarramaneni, ”Dynamic Scalability Architecture,” *LinkedIn*, Online: https://www.linkedin.com/pulse/dynamic-scalability-architecture-nirmal-ganesh-yarramaneni/, [Accessed: April 8, 2024].

[20] “draw.io”, *draw.io* [Online]. Available: https://draw.io/, [Accessed: April 8, 2024].

[21] Salesforce, ”What is SaaS?” Salesforce. [Online]. Available: https://www.salesforce.com/saas/. [Accessed: April 8, 2024].

[22] IBM Cloud, ”What is Function as a Service (FaaS)?” IBM Cloud. [Online]. Available: https://www.ibm.com/cloud/learn/faas. [Accessed: April 8, 2024].