DALHOUSIE UNIVERSITY

Title: Course Report

Class: CSCI 5902 – Advanced Cloud Architecting

Professor: Lu Yang

Student: Jaivik Chetankumar Tailor

Banner ID: B00915987

Date: 30th July 2023

Application Overview:

I made To-Do List Maker Web application for my last Cloud Computing course. A To-Do List Maker application is a software tool designed to help users keep track of their daily tasks and responsibilities. It allows users to create and organize their to-do lists, set due dates and reminders, and track progress toward completion. I used some cloud services in different categories. That I have mentioned below.

List of services I have used to create application:

Compute: AWS Elastic Beanstalk and AWS Lambda
Storage: AWS DynamoDB
Network: AWS Virtual Private Cloud (VPC)
General: AWS SNS and Amazon Textract

Compute:

1. **What cloud services did you use?**

   I used **AWS Lambda** and **AWS Elastic Beanstalk** in my application under Compute category for different purposes.

   - **Why did you use those cloud services?**

     To **host my Node.js backend** in the application, I chose **Elastic Beanstalk** as a managed **Platform-as-a-Service (PaaS)** offering. Elastic Beanstalk was chosen because of its ability to facilitate the deployment and management of web applications and backend services. By using this service, I was able to **automate infrastructure provisioning and scaling**, which dramatically reduced operational costs and assured a smooth deployment process [1]. Because Elastic Beanstalk supports Node.js apps, it was the obvious choice for my backend deployment, removing the need to manage servers, load balancers, and other infrastructure components manually. With Elastic Beanstalk handling the underlying infrastructure, I could focus more on developing the core features of the application and rely on the service's automated scaling to accommodate varying levels of user demand. This decision proved to be efficient and streamlined, allowing me to deploy and manage the backend with ease while ensuring a reliable and scalable foundation for the application's functionality.

     In order to enable the nightly email trigger functionality, I used **AWS Lambda as the serverless computing service in my application**. I was able to execute code in response to events by using AWS Lambda, which relieved me of the hassle of managing servers. I used Lambda to activate a nightly function that sent email notifications to users, asking them to set their to-do lists for the next day. Lambda's serverless feature was useful, ensuring **cost-effectiveness and scalability** because I only paid for the compute time used during the function's execution [2]. This corresponds precisely with the infrequency of the email sending task, allowing me to perform the process efficiently without incurring additional fees or maintaining dedicated server instances. With AWS Lambda, I could focus on implementing the desired functionality while benefiting from its

serverless capabilities, creating a reliable and cost-efficient solution for nightly email notifications in my to-do list maker application. AWS Lambda is highly scalable, which allows me to handle a large number of email requests without managing the underlying infrastructure. This is essential in a To-Do List application to send email to users to set their next To-Do-List.

- **After reviewing your project, are you going to use the same services? Why?**

  After analyzing my application and project, in the development of my to-do list application, I have encountered some limitations with **Elastic Beanstalk** that I believe can be overcome by utilizing **EC2 instances**. One key drawback is the limited control over the underlying infrastructure when using Elastic Beanstalk. This abstraction can restrict my ability to customize server configurations and install specific software as per my application's requirements. By opting for EC2 instances, I can gain complete control and flexibility, allowing me to tailor the server environment precisely to meet the unique demands of my application [3].

  Additionally, while Elastic Beanstalk provides a straightforward deployment model, it may not offer the same level of performance optimization that can be achieved with EC2 instances. By leveraging EC2, I can fine-tune the instance types, network settings, and other parameters to maximize performance and efficiency, resulting in a more responsive and robust application [3].

  Furthermore, Elastic Beanstalk's simplified deployment approach might not be the most suitable option for complex application architectures. With EC2 instances, I have the freedom to design and implement a more customized and intricate architecture, integrating load balancers, multiple servers, and dedicated database instances to enhance overall performance and scalability [3]. In future, to do list maker might require complex architecture that can’t be manage by Elastic Beanstalk.

  Moreover, although Elastic Beanstalk offers automatic scaling, configuring scaling policies for specific use cases can be less straightforward. By managing EC2 instances directly, I can have greater flexibility and control over Auto Scaling settings, allowing me to tailor scaling rules precisely to match my application's workload patterns.

  Lastly, with EC2 instances, I can implement cost-saving measures more effectively. Elastic Beanstalk abstracts certain cost-related decisions, which may result in suboptimal resource allocation. However, by directly managing EC2 instances, I can take advantage of cost optimization strategies, such as scheduling instances to run only during peak times or utilizing cost-effective spot instances [3].

  In conclusion, using EC2 instances instead of Elastic Beanstalk addresses these limitations and empowers me to create a more customized, high-performance, and cost-efficient infrastructure for my to-do list application, ultimately providing an enhanced user experience and better application performance.

On the other hand, I will continue with AWS Lambda for my serverless architecture of to do list maker application's regular email trigger. There are several compelling reasons to continue using AWS Lambda for the nightly email trigger in my to-do list application [2]. The serverless architecture of AWS Lambda relieves the burden of operating servers while also ensuring cost-effectiveness by charging only for compute time consumed during function execution. Its event-driven design allows for automatic and dependable implementation of the email trigger without the need for manual intervention. Furthermore, Lambda's intrinsic scalability dynamically adjusts to fluctuating workloads, handling any number of email messages efficiently. Integration with other AWS services, such as Amazon SNS for email sending and Amazon DynamoDB for user data storage, speeds up development and promotes a unified solution. Furthermore, the managed infrastructure does not require maintenance and provides great availability and reliability. These all mentioned benefits will the best choice regarding my to do list maker application.

2. **Did you apply any security measures to the services?**

Yes, I have applied some security measures for Elastic Beanstalk and AWS Lambda. I have used **Amazon Virtual Private Cloud (VPC)** to isolate my Elastic Beanstalk environment from the public internet. This helps in **securing my backend** services and prevents unauthorized access.

However, several significant procedures can be done to improve the security of my **Elastic Beanstalk system**. To begin, using an **Application Load Balancer (ALB)** is advantageous because it efficiently distributes incoming traffic to the Elastic Beanstalk instances while also providing intrinsic security features such as **SSL/TLS termination**, which protects data during transit. **Enabling Auto Scaling** within the Elastic Beanstalk environment also ensures that the number of instances adjusts dynamically based on demand, offering high availability and fault tolerance.

AWS Identity and Access Management (IAM) can be used to create precise permissions for different components of the Elastic Beanstalk environment to further strengthen security. The danger of unauthorised access is reduced by providing least privilege access to IAM roles and users. Secure communication methods such as HTTPS should be used to protect data transmission between clients and Elastic Beanstalk instances [4].

Moreover, implementing **Network Security Groups (NSGs) and Network Access Control Lists (NACLs)** aids in controlling inbound and outbound traffic at both the network and subnet levels, adding an extra layer of protection. Regularly updating and patching the Elastic Beanstalk environment is critical to mitigate known vulnerabilities and ensure the system's overall security integrity [4].

By implementing these security measures, I can significantly strengthen the security posture of your Elastic Beanstalk environment, safeguarding my application's sensitive data and ensuring a safe and reliable user experience.

To ensure robust security for **Lambda functions** in the to-do list application, several essential measures can be implemented. Firstly, assign an **IAM execution** role to the **Lambda function** with the least privilege required to access other AWS services. This restricts access to resources, preventing unauthorized users from accessing sensitive data. Additionally, **implement encryption** for both data at rest and in transit using **AWS Key Management Service (KMS) and SSL/TLS certificates**, providing an extra layer of protection for your data [5].

Monitoring and logging are crucial aspects of Lambda security. Set up **CloudWatch Alarms** to monitor Lambda function performance and security, enabling timely detection of any potential anomalies or security breaches. Regularly reviewing logs helps in identifying and mitigating suspicious activities.

I can implement versioning for Lambda functions. It is important for maintaining a reliable and secure application. With versioning enabled, I can easily roll back to a previous version in case of issues or security incidents, ensuring a smooth and efficient recovery process that I had not done before.

By adhering to these security best practices, My Lambda functions in the to-do list application will benefit from enhanced protection against potential security threats, ensuring the safety and integrity of my users' data and the overall application performance.

3. **Was your application monolithic?**

Yes, my application can be considered monolithic since I have implemented the whole application using a **single codebase project**. In a monolithic architecture, all functionalities and components of the application are tightly integrated into a single codebase, making it a single unit that is deployed and managed together. My all functionalities were tightly coupled with each other.

Even though I have utilized various AWS services such as SNS, Lambda, DynamoDB, and VPC for specific functionalities, the fact that all the code is combined into a single codebase suggests a monolithic architecture. After analyzing my past development, I will better suggest to adopt a **microservices approach**. In this approach, I will create separate microservices to handle **user authentication, to-do list management, email notifications, and image processing using Amazon Textract**. To streamline communication with these microservices, I will implement an API Gateway as a centralized entry point for the application. This **API Gateway** will efficiently route requests from the front-end application to the appropriate microservices.

In order to give myself the freedom to leverage **AWS Lambda for serverless tasks or EC2 instances** for finer control over the infrastructure, I will deploy each microservice as a standalone application. To-do list data will continue to be stored in DynamoDB for data management, with each microservice having a separate database or sharing data as needed through APIs and events.

To implement an event-driven architecture, I will **use AWS SNS and AWS Lambda**. For instance, instead of directly triggering the nightly email function from a Lambda function, I will have the to-do list microservice publish an event to an SNS topic. The email notification microservice can then subscribe to that topic and handle the email sending accordingly.

To ensure secure access, I will implement a centralized authentication service like **AWS Cognito**, which will handle user authentication and provide access control to the microservices. To achieve a decoupled front-end, I will ensure that the front-end application interacts with the microservices solely through the **API Gateway**. Standardized communication protocols, such as RESTful APIs or GraphQL, will be used to facilitate seamless data exchange.

Lastly, I will implement **centralized monitoring and logging** solutions to gain comprehensive visibility into the performance and health of each microservice, allowing for effective monitoring and maintenance of the entire application.

4. **Did you have a disaster plan?**

No, my to-do list maker application **does not have a disaster plan** specifically for Elastic Beanstalk and Lambda in your to-do list maker application. A disaster plan is a crucial aspect of any application's architecture, as it ensures business continuity and minimizes the impact of potential disasters or outages.

To create a disaster plan for Elastic Beanstalk and Lambda, I will consider the following aspects:

Firstly, I will consider deploying my **Elastic Beanstalk instances and Lambda functions across multiple availability zones (AZs)** to ensure high availability and fault tolerance. By distributing my resources across different AZs within the same region, I can significantly improve the **resiliency of my application** [2]. In case of a failure or outage in one availability zone, my application can seamlessly switch to resources in another zone, avoiding service disruptions and ensuring continuous availability for my users [1]. **This approach provides a robust disaster recovery strategy and helps maintain a reliable user experience even during unexpected events.**

Additionally, it enhances the performance of my application by reducing latency and improving response times for users in different geographical locations. For **Lambda functions specifically**, I will design them to be **stateless and leverage AWS's built-in fault tolerance**, which automatically replicates functions across multiple data centers [2]. This ensures that Lambda functions can handle failures and continue processing requests without disruption. By introducing these redundancy and failover measures, my application will be better equipped to handle unexpected failures and maintain high availability for users, providing a smooth and uninterrupted experience.

Furthermore, implementing a robust **disaster recovery (DR)** strategy is crucial to ensure the safety and continuity of my data and services. I will configure automated snapshots for my Elastic Beanstalk instances and Lambda functions to capture their state and configurations regularly. By establishing well-defined recovery procedures and conducting periodic testing and drills, I can confidently mitigate potential risks and quickly restore the functionality of my to-do list application in case of any unforeseen events.

In addition to this, to proactively identify and handle any **potential problems or anomalies**, I will integrate strong **monitoring and alerting methods for both Elastic Beanstalk and Lambda** in my application. I'll set up in-depth monitoring for vital performance indicators like

CPU, RAM, and request latency for Elastic Beanstalk instances and Lambda functions using **AWS CloudWatch** [6]. I can make sure that the application is operating smoothly and effectively by closely monitoring these indicators. Aside from that, I'll set up **CloudWatch** alarms so that they send out notifications right away if any crucial thresholds are crossed or if the environment has any issues or failures.

By incorporating these elements into my disaster plan for Elastic Beanstalk and Lambda, I can enhance the resilience and availability of my to-do list maker application, ensuring it can withstand and recover from various disaster scenarios effectively.

5. **Were the services highly available and reliable?**

Currently, both Elastic Beanstalk and Lambda **are not highly available and reliable** in my application but I can configured to be highly available and reliable in my application by implementing load balancing and setting up backup and replication mechanisms.

- **Load Balancing for Elastic Beanstalk:** In front of your Elastic Beanstalk instances, I'll introduce an **Application Load Balancer (ALB)** [1]. Incoming traffic will be evenly distributed among several instances via the ALB, preventing any one instance from becoming overloaded and improving fault tolerance. Users will see the fewest interruptions possible if traffic is immediately routed to healthy instances whenever one instance goes down.
- **Cross-Region Replication for Lambda Functions:** I will explore replicating Lambda functions across several AWS regions to improve stability and availability. Cross-region replication ensures that even if one AWS region goes down, my functions can still be executed from another, ensuring uninterrupted service availability [2]. AWS Lambda offers cross-region replication by replicating function code and configurations to various regions utilising services such as AWS Lambda@Edge or cross-region S3 triggers.
- **Backup and Restore Strategies:** I will implement regular backups of your Elastic Beanstalk environment configurations and Lambda function code. Having backup snapshots allows me to quickly restore my environments or functions in case of accidental changes, failures, or data corruption.

In addition to this, With Elastic Beanstalk same like disaster recovery, I can deploy application across multiple availability zones (AZs) to ensure high availability and fault tolerance. This multi-AZ deployment means that if one AZ experiences issues, my application can automatically switch to a healthy AZ, minimizing downtime and ensuring continuous operation. Additionally, Elastic Beanstalk's auto-scaling feature can dynamically adjust the number of instances based on demand, providing scalability and ensuring that application can handle varying workloads [2] [1].

6. **Did you consider the cost of the services?**

Yes, I have considered the cost of both AWS Elastic Beanstalk and AWS Lambda in my to-do list application. I have calculated the estimated cost for running an EC2 t2.micro instance with Elastic Beanstalk, including the EC2 instance cost, Elastic Beanstalk platform fee, and storage cost. The total estimated cost for running Elastic Beanstalk continuously for a month without additional services was $26.43. I have not provided an exact cost estimate for AWS Lambda, I

was aware that there might be a possibility of incurring charges if my usage exceeds the free tier limits. I can further improve the cost effectiveness of **AWS Lambda and Elastic Beanstalk** by implementing various optimization strategies.

To achieve **cost effectiveness** for **AWS Lambda**, I will implement several optimization strategies. Firstly, I will carefully assess each Lambda function's resource requirements and adjust the **memory allocation accordingly** [2]. By finding the **optimal memory size** for each function, I can minimize **the cost per invocation** without compromising performance. Additionally, I will optimize the execution time of Lambda functions **by refining their code** to complete tasks efficiently and quickly, leading to lower costs. **For functions experiencing fluctuating traffic**, I will leverage **provisioned concurrency** to reduce startup time and enhance response times, ensuring cost efficiency during varying workloads. **To avoid unnecessary function invocations**, I will fine-tune the trigger frequency of Lambda functions that rely on events. Lastly, I will utilize **AWS Lambda Layers** to store reusable code, thereby reducing the size of deployment packages. This approach will facilitate faster function startup times, ultimately contributing to lower costs while maintaining the application's performance and responsiveness [2].

To improve the cost effectiveness of **AWS Elastic Beanstalk** in my application, I will implement several strategies. I will carefully examine my application's workload and select the appropriate **EC2 instance type and size to right-size the instances** [1]. I may analyze resource utilization and identify cost-cutting options with **AWS Cost Explorer** [7]. For long-term workloads, I will use **AWS Reserved Instances**, which provide discounted hourly rates, further lowering costs. I will configure Elastic Beanstalk's **auto scaling** settings to dynamically alter the number of instances depending on actual traffic to guarantee efficient resource allocation. By having the appropriate amount of instances operating at any one time, I may minimize excessive charges during periods of low demand.

I will also analyze the application's availability requirements. If it does not need to run continuously, I can put up a **schedule to shut down the Elastic Beanstalk** environment during **off-peak hours**, saving on instance running expenses while maintaining functionality. In addition, I will examine my application's **storage consumption** on a regular basis and optimize storage configurations by eliminating any unneeded data or applying storage optimization strategies to save costs. **By putting these safeguards in place**, I can ensure that my AWS Elastic Beanstalk setup is cost-effective while still providing the performance and scalability that my to-do list application requires.

7. **Did you use monitoring and/or logging tools to give you graphs, messages, metrics, or alerts of the services?**

I have not used any monitoring and/or logging tools before for Elastic Beanstalk and Lambda. However, I will consider the following tools:

AWS CloudWatch: CloudWatch is a crucial monitoring and logging service provided by AWS. It allows me to collect and track metrics, set up alarms for threshold breaches, and monitor the performance of my Elastic Beanstalk instances and Lambda functions. CloudWatch provides detailed insights into the health and resource utilization of my services, enabling me to detect and respond to any issues promptly [6]. It also offers logging capabilities, allowing me to centralize

and analyze log data for troubleshooting and auditing. It provides graphs and charts to help me understand the resource utilization, request rates, and other key performance indicators. Additionally, CloudWatch allows me to set up alarms based on predefined thresholds or custom metrics, and it sends alerts via email or other notification channels when these thresholds are breached.

AWS CloudTrail: CloudTrail is essential for logging all API calls and actions taken on AWS resources, providing an audit trail of activity within my AWS account. By enabling CloudTrail, I can track changes made to your Elastic Beanstalk environment and Lambda functions, helping me maintain security and compliance. It's a valuable tool for monitoring and logging any unauthorized access attempts or suspicious activities [8]. The recorded logs can be used to audit and investigate any modifications, providing an additional layer of security and compliance.

By incorporating these monitoring and logging tools into your to-do list application, you can gain visibility into the performance and health of my Elastic Beanstalk and Lambda services. CloudWatch and CloudTrail help you track metrics, set alarms, and monitor changes.

Networking:

1. **What cloud services did you use?**

   I used **VPC (Virtual Private Cloud)** for networking in my to-do list maker application under networking category.

   - **Why did you use those cloud services?**

     In my to-do list application, I utilized AWS VPC (Virtual Private Cloud) to bolster the security and isolation of the resources within the **AWS cloud environment**. By creating a **private network using VPC**, I ensured that all the components and data of my application **remained shielded** from the public internet, effectively minimizing exposure to potential threats and unauthorized access [9]. This added layer of security greatly enhances the overall protection of my application.

     Furthermore, VPC allowed me to logically segment my application into distinct subnets, including backend, and database. This network segmentation facilitated a **clear separation of functionalities**, ensuring that **sensitive backend services** were not directly exposed to the internet. As a result, I experienced enhanced security measures and simplified network management. The clear segregation of components within different subnets **not only mitigated the risk of potential vulnerabilities** but also made it easier to **identify and resolve any issues** that might arise in specific areas of the application. I just wanted to secure my application database from the additional layer as I have many personal details of user and their tasks that I can connect using VPC by providing private and public access using subnets [9].

     In building my to-do list application, I made use of AWS VPC (Virtual Private Cloud) to take advantage of its **controlled network configuration** capabilities. With VPC, I had complete control **over defining IP address ranges, routing tables, and network**

gateways, which allowed me to design a customized and secure network topology specific to the requirements of my application. This **level of control** was crucial in ensuring that my application's resources were **appropriately isolated** and that only **authorized traffic could access** the different components**.**

Overall, AWS VPC offers a robust and secure network foundation that complements my to-do list application's architecture. By using VPC, I have established a reliable and well-protected network environment that fosters seamless communication between different services while safeguarding sensitive data and resources from potential threats.

- **After reviewing your project, are you going to use the same services? Why?**

For network category, I will continue to use VPC (Virtual Private Cloud) for my to do application. There are several reasons to continue to use VPC. In my to-do list application, I have implemented AWS VPC to enhance security and protect sensitive data from potential external threats. By continuing to use VPC and creating private subnets, I ensure that critical components, like the backend database where users' task details are stored, remain isolated from the public internet. This ongoing security measure **effectively shields the database from direct exposure to the internet**, reducing the risk of unauthorized access or data breaches [9].

Furthermore, by establishing **Network Access Control Lists (NACLs)** and **Security Groups** within the VPC, I may strengthen the security of my application even further. NACLs enable me to regulate inbound and outbound traffic at the subnet level, specifying particular rules for granting or refusing resource access [10]. In contrast, Security Groups enable me to restrict traffic at the instance level by declaring which protocols and ports are open to inbound and outbound traffic.

By continuously leveraging VPC's capabilities and making effective changes like implementing NACLs, Security Groups, my to-do list application will maintain a **robust security posture**. Critical data, such as user login credentials and task details, will remain protected from external threats, fostering user trust and confidence in the application's security measures.

By **logically segmenting** my application into different subnets, such as frontend, backend, and database, I create clear separation between these components. This ongoing network segmentation enhances the security of my application by isolating sensitive backend and database resources from the frontend, reducing the potential attack surface.

With this network segmentation, I can implement fine-grained access controls through Network Access Control Lists (NACLs) and Security Groups [10]. **For example,** I can configure the frontend subnet to only allow incoming traffic on HTTP and HTTPS ports from the internet, while the backend subnet can only accept traffic from the frontend subnet and specific IP addresses of trusted services. This prevents **unauthorized access** between different parts of the application and adds an additional layer of protection against potential threats.

Continuously using VPC endpoints and private subnets helps optimize costs by reducing data transfer fees and eliminating the need for unnecessary internet gateways. This ongoing cost optimization contributes to overall financial efficiency. By persistently using VPC, I benefit from a well-organized and manageable network infrastructure. The ongoing network design and configuration enable easier monitoring, troubleshooting, and maintenance of the application.

In summary, continuously using AWS VPC in my to-do list application ensures ongoing security, scalability, and cost optimization. The ongoing implementation of VPC features and best practices enables me to maintain a robust and efficient network architecture, meeting the evolving needs of the application and its users.

2. Did you apply any security measures to the services?

Yes, I used security measures designed expressly for AWS VPC in my application. I have taken steps to secure my application's network architecture by using AWS VPC and configuring my Elastic Beanstalk environment in a public subnet with an internet gateway. This method allows my application to be accessed from the internet while maintaining backend resources, such as the DynamoDB database, in private subnets within the VPC.

Additionally, I have utilized AWS endpoints to connect various services within the VPC, ensuring that communication between different components of my application remains secure and isolated from external networks.

While the security measures implemented for AWS VPC in to do list maker application are a good starting point, they may not be sufficient on their own to ensure comprehensive security. VPC provides a strong foundation for network isolation and protection, but it's important to consider additional security measures to further enhance the overall security posture.

Some potential new security measures that I can apply to VPC are:

Network Access Control Lists (NACLs): By offering fine-grained control over inbound and outbound traffic at the subnet level, NACLs can add an extra layer of protection [10].

Network Flow Logs: Enabling VPC Flow Logs allows me to collect data on IP traffic to and from network interfaces in my VPC. This information can be utilised to identify potential security concerns and to fix network faults [11].

AWS PrivateLink: Utilizing AWS PrivateLink, I can securely access AWS services over private connections within the VPC, avoiding exposure of sensitive data to the public internet [11].

AWS Transit Gateway: Implementing AWS Transit Gateway enables centralized management and routing of traffic between VPCs, simplifying the network architecture and improving security by reducing the number of internet-facing entry points.

Distributed Denial of Service (DDoS) Protection: Enabling AWS Shield, which provides DDoS protection, helps safeguard my application from potential DDoS attacks including VPC.

AWS Config: Enabling AWS Config can provide continuous monitoring and assessment of resource configurations, helping to maintain compliance and identify security risks.

By combining these additional security measures with the existing ones, I can create a more robust and secure VPC environment for my to-do list application.

3. **Was your application monolithic?**

Yes, my application is monolithic. To decouple application, I will begin by creating private subnets within my VPC to host the microservices instead of deploying my Elastic Beanstalk in a public subnet. By doing so, I ensure that my microservices are not directly accessible from the internet, which will significantly improve the security of my application. I will place my backend Node.js application running on Elastic Beanstalk in one of these private subnets, ensuring that it is only accessible within the VPC.

Next, I will implement an API Gateway in a public subnet to decouple the frontend from the backend microservices. The API Gateway will serve as the entry point for all client requests and will route them to the appropriate microservices running in private subnets. This way, my frontend code in React will communicate only with the API Gateway, which will act as a reverse proxy for the microservices. This separation will allow me to manage and scale each microservice independently, promoting flexibility and ease of maintenance.

To securely access DynamoDB from my microservices in private subnets, I will use AWS PrivateLink. This feature enables me to access DynamoDB directly within the VPC without exposing the database to the public internet. This enhanced security measure will protect sensitive data and prevent unauthorized access to the database.

VPC peering will be used to enable smooth connectivity between various microservices running in distinct private subnets. This will allow for direct and efficient communication between the microservices without the need for public internet access, boosting the privacy and security of my application even further.

When implementing a Lambda function triggered by SNS to send emails, I will place this function in a private subnet along with a NAT Gateway or NAT instance. This setup will allow the Lambda function to access external resources like the internet for sending emails while operating within the private VPC network. This ensures that my Lambda function remains isolated and secure.

By following these steps and making these changes to my VPC structure and application architecture, I will effectively decouple the components and successfully transform my monolithic application into a scalable, modular, and more manageable microservices-based architecture. This approach will provide me with greater flexibility, easier maintenance, and improved scalability for my to-do list maker application.

4. **Did you have a disaster plan?**

   No, I did not make any disaster plan for VPC in my application as that was in initial stage. To ensure the resilience and availability of your application in the event of unforeseen failures or disasters, it is crucial to have a disaster recovery plan specifically designed for the AWS VPC.

   Here are some steps you can take to create a disaster recovery plan for my VPC:

   Automated Snapshotting: I will enable automated snapshotting of my VPC configurations, including subnets, security groups, and routing tables. These snapshots will be used to quickly restore the network setup in case of any unintended changes or disruptions, ensuring the integrity and availability of my VPC [11].

   VPC Peering and Transit Gateway: I will set up VPC peering or utilise a transit gateway to facilitate secure connectivity between multiple VPCs across regions to improve redundancy and backup capabilities. This will give me with an alternate communication method in the event that one of my VPCs becomes unreachable, ensuring that my application maintains continuous connectivity [9].

   Monitoring and Alerts: To proactively detect and address possible issues, I will use AWS CloudWatch to implement thorough monitoring. By configuring relevant alerts, I will be notified of any crucial events or odd behaviour, allowing me to take immediate action to keep my VPC healthy and performing well.

   Disaster Recovery Testing: To make sure my disaster recovery strategy is reliable and successful, I will test it frequently. I will test my ability to successfully recover my VPC and application in the event of a disaster through simulations and drills, giving me the assurance that my VPC is robust and well-prepared [12].

   Documentation and Communication: I will document my disaster recovery plan thoroughly, detailing step-by-step procedures and including contact information for key personnel. By clearly defining roles and responsibilities during a disaster, I will ensure efficient and coordinated actions. Moreover, I will communicate the plan to all relevant stakeholders, ensuring everyone is aware of their roles and actions in case of an emergency, promoting a cohesive response to any potential disruptions.

   Combination of these all services will be the great recovery or disaster plan for VPC. By combining these all to-do list application remains available and reliable even in challenging situations.

5. **Were the services highly available and reliable?**

   I have taken steps to ensure high availability and reliability for my VPC. But, that was limited and can be make highly available and reliable by some steps.

Redundant Internet Gateways: I will consider setting up redundant internet gateways in different Availability Zones (AZs). This will ensure that even if one internet gateway becomes unavailable, my VPC can still maintain internet connectivity through the other gateway, providing high availability for my application. I used only one internet gateway in an initial stage that I will change here [13].

Elastic IP Addresses (EIPs): As part of my disaster recovery strategy, I suggested to use Elastic IP addresses to provide a static and public IP address to resources within my VPC. This will enable me to easily remap EIPs to different instances in case of instance failure, allowing for quick recovery and minimizing downtime. That will enhance availability and reliability [14].

Route 53 Health Checks and Failover: I will utilize Amazon Route 53 health checks to monitor the health of my application endpoints that I did not use before. By configuring failover routing policies, I can direct traffic to alternative endpoints in case of endpoint or application failures, ensuring seamless user experience and minimal disruptions.

Multiple NAT Gateways: I will set up multiple NAT gateways in various AZs to improve outbound traffic continuity from my private subnets. This will eliminate a single point of failure and provide consistent internet connection for my private subnets at all times [10].

Backup VPC Configuration: As a proactive measure, I will regularly back up my VPC configuration using AWS CloudFormation or AWS Config. Storing the configuration in versioned templates will enable me to quickly recreate the VPC in case of accidental deletion or misconfiguration, reducing the impact of potential issues.

Security Group Best Practices: To strengthen the security of my VPC, I will apply security group best practices. This includes using separate security groups for different tiers of my application and limiting access to only necessary ports and IP ranges, minimizing the attack surface [15].

Along with my disaster management, I will take mentioned steps to improve availability and reliability for my VPC in to do list maker. These all action was not taken by me previously.

6. **Did you consider the cost of the services?**

Yes, I considered cost of VPC and NAT Gateway but that was approximate value for my application.

As the developer of the to-do list application, I will implement several cost optimization strategies to effectively manage expenses related to the AWS Virtual Private Cloud (VPC) without compromising on the application's reliability and performance.

Firstly, I will carefully review the resource allocation for my **VPC components**, such as Elastic Beanstalk, DynamoDB, and NAT gateways. By ensuring that these resources are appropriately sized to match the actual workload requirements, I can avoid overprovisioning and optimize

costs. This means that I will provision resources based on the application's actual usage and adjust them as needed to avoid unnecessary expenses.

Additionally, I will leverage **AWS PrivateLink** to securely access services like DynamoDB and SNS from my VPC. AWS PrivateLink allows me to establish private connections within my VPC to these services, avoiding data transfer costs over the public internet. This not only reduces data transfer charges but also lowers bandwidth costs and enhances security without incurring additional expenses [16]. The simplified network architecture provided by PrivateLink further contributes to cost optimization, while the granular control over data flow ensures that only authorized resources access specific services, reducing the risk of unauthorized usage and associated costs.

To monitor and manage data transfer costs, I will keep a careful eye on the price of data transfers between the VPC and other AWS services, as well as within the VPC itself. I can lower outbound data transfer costs by leveraging caching, content delivery networks, and minimizing data transmission where feasible. Additionally, I will enable **VPC Flow Logs** to capture network traffic for monitoring and troubleshooting purposes. However, I will ensure that I am recording the necessary data without generating an excessive amount of log volume, which could result in higher log storage expenses [17].

To proactively manage costs, I will utilize **AWS Cost Explorer** to analyze my VPC-related expenses and create cost alerts. By proactively monitoring costs and receiving notifications when they go above predetermined criteria, I can quickly identify potential cost overruns and take the necessary actions to optimize spending [7].

By implementing these cost optimization strategies, I can ensure that my to-do list application benefits from a cost-effective infrastructure within the AWS VPC. The application will continue to deliver a seamless user experience, while the optimized cost structure ensures efficient resource utilization and budget management.

7. **Did you use monitoring and/or logging tools to give you graphs, messages, metrics, or alerts of the services?**

Previously, I did not focusing on any monitoring and/or logging tools, but after learning Cloud architecture course I understand the value of monitoring and analyzing pattern of each service. I have decided to use 5 different tools for analyzing and monitoring VPC in my application.

- Amazon CloudWatch
- VPC Flow Logs
- AWS X-Ray
- AWS CloudTrail
- AWS Config

By using Amazon CloudWatch, I will monitor the performance and health of my **AWS VPC in real-time**. I will set up CloudWatch alarms to receive notifications when specific metrics, such as CPU utilization or network traffic, exceed defined thresholds [6]. This proactive monitoring approach will allow me to quickly identify and address any performance issues or anomalies,

ensuring my VPC operates optimally and maintains high availability for my to-do list application. Additionally, I will create custom CloudWatch dashboards to visualize key metrics and gain deeper insights into the overall health of my VPC.

By enabling **VPC Flow Logs**, I will monitor **the network traffic within my VPC.** I will analyze the flow log data to track the source and destination of traffic, the protocols used, and any denied or rejected connections. This monitoring will help me troubleshoot connectivity issues, detect potential security threats, and ensure compliance with my network policies. With VPC Flow Logs, I will have a comprehensive view of the **network activity within my VPC**, allowing me to maintain a secure and efficient network environment for my application [17].

By utilizing **AWS X-Ray**, I will monitor the **performance of my application and its individual components**. X-Ray will enable me to trace requests as they flow through my application, providing insights into the latency and performance of each service. With X-Ray, I will be able to pinpoint any bottlenecks or errors within my application, allowing me to optimize its performance and enhance the user experience. This detailed monitoring will help me identify areas for improvement and make data-driven decisions to ensure my to-do list application operates efficiently.

By using **AWS CloudTrail**, I will monitor all **API activity and changes made to my VPC** and other AWS resources. CloudTrail will provide a comprehensive audit trail of actions performed on my VPC, including who performed the actions and when. This monitoring will allow me to detect any unauthorized access attempts or changes to my VPC configuration, enhancing the security and compliance of my application. With CloudTrail, I will have a **reliable record of all VPC-related activities**, providing me with valuable insights into the changes and events occurring in my environment [8].

By leveraging **AWS Config**, I will continuously monitor the **configurations of my VPC and related resources**. AWS Config will automatically assess and evaluate the state of my VPC against my desired configurations and best practices. If any resource configurations drift from the desired state, **AWS Config will alert me, allowing me to quickly remediate any non-compliant configurations**. This proactive monitoring will help me maintain a consistent and secure VPC environment for my to-do list application, ensuring that it aligns with my organization's policies and reducing the risk of configuration errors.

Using these monitoring and logging tools together will provide me with comprehensive insights into the performance, health, and security of my VPC in the to-do list application. The combination of CloudWatch, VPC Flow Logs, AWS X-Ray, AWS CloudTrail, and AWS Config will help you proactively identify and address any issues, optimize performance, and enhance the overall reliability of my VPC.

Database:

1. **What cloud services did you use?**

   I have used **AWS DynamoDB** for data management in my to-do list maker.

   - **Why did you use those cloud services?**

I have used **DynamoDB** for data management, DynamoDB's flexible data model allows for storing structured, semi-structured, and unstructured data, which makes it easy to store and retrieve data for my application. DynamoDB provides **low latency access** to data, which means that users can retrieve their To-Do-List items quickly and efficiently. DynamoDB is **a highly secure** service that provides several security features, including encryption at rest and in transit, access control, and auditing [18]. While S3 and Aurora are both excellent storage systems, they might not be the greatest option for my application. S3 is primarily intended for storing and retrieving huge files such as photos, movies, and documents, and may lack the capabilities required for storing structured data. In contrast, Aurora is a relational database service that may be more difficult to set up and manage than DynamoDB. Overall, AWS DynamoDB is a better fit than S3 or Aurora for this application.

In addition to that, DynamoDB is a fully managed NoSQL database service provided by AWS. As a managed service, it takes care of the underlying infrastructure, automatic scaling, data replication, and backups, allowing me to focus on building my application's features rather than managing the database infrastructure. Since my application was hosted on AWS and I was already using other AWS services like SNS and Lambda, DynamoDB integrates well with the AWS ecosystem. This makes it easy to store and retrieve data from DynamoDB tables in a secure and efficient manner.

One of the reasons I chose DynamoDB for my to-do list application is because it efficiently handles unstructured data. In a to-do list application, the tasks created by users can vary widely in content and structure. Some tasks may only have a simple title and description, while others might include additional attributes like due date, priority level, tags, and attachments. This variability in data structure makes it challenging to use a traditional relational database that requires a fixed schema. DynamoDB, being a NoSQL database, allows me to store unstructured data without the need to define a rigid schema beforehand. I can easily add new attributes to tasks as needed, and each task can have a unique set of properties. This flexibility is crucial as it enables me to adapt the data model of my application on-the-fly and accommodate any future changes in user requirements or application features without disrupting the database [18]. With Dynamo DB’s seamless support for unstructured data, I can efficiently manage the diverse information associated with tasks in my to-do list application, providing a dynamic and scalable solution for storing and retrieving task-related data.

- **After reviewing your project, are you going to use the same services? Why?**

  I will continue to use DynamoDB to store data in my application as that has numerous advantages, one of the key benefits of DynamoDB is its exceptional scalability. Being a fully managed NoSQL database, it automatically scales to handle any amount of data and traffic without requiring manual intervention. This means that as my application grows and attracts more users, DynamoDB can seamlessly handle the increasing workload, ensuring that your application remains responsive and performs well even during periods of high demand.

Another compelling reason to stick with DynamoDB is its flexibility. In a to-do list application, user requirements and data structures may evolve over time. DynamoDB's flexible schema allows me to easily add, modify, or remove attributes from my data without the need for complex database migrations. This adaptability makes it easier to accommodate changes and implement new features in my application without disrupting the existing data or causing downtime [18].

DynamoDB also excels in performance, providing low-latency read and write operations. For a to-do list application, where tasks are frequently added, edited, and viewed by users, DynamoDB's fast response times ensure a smooth and efficient user experience. This real-time data access is essential for keeping the application responsive and delivering an excellent user experience.

Furthermore, DynamoDB seamlessly integrates with AWS Lambda, which I have already utilized in my application. This integration allows me to build serverless architectures efficiently, reducing operational overhead and costs. By leveraging AWS Lambda with DynamoDB, I can create a highly scalable and cost-effective backend for your to-do list application.

In case my application's user base is distributed globally, DynamoDB's multi-region support comes into play. This feature allows me to replicate data across different AWS regions, ensuring low-latency access to data for users located in various parts of the world. As a result, users from different regions will experience minimal delays in accessing their to-do lists, providing a better overall user experience.

Lastly, DynamoDB being a managed service handles administrative tasks such as hardware provisioning, setup, and maintenance. This relieves me from the burden of managing the database infrastructure, allowing me to focus on developing new features and improving the functionality of my to-do list application.

Considering all these benefits, DynamoDB proves to be a highly suitable and practical choice for to-do list maker application [18]. It offers the scalability, flexibility, performance, global distribution, and managed service features required to meet the evolving needs of my users while streamlining database management and ensuring a seamless user experience.

2. Did you apply any security measures to the services?

No, I did not apply any security specific measures to the DynamoDB. I just used VPC endpoints to connect all service and to make sure about security. As VPC endpoint provide me security layer, I have not applied specific security to DynamoDB.

As I continue to develop and enhance my to-do list maker application, I will use various security measures to safeguard the data and ensure the privacy and integrity of user information stored in DynamoDB. Firstly, I will define appropriate IAM roles and policies to control access to DynamoDB [19]. By configuring least privilege access, I can restrict permissions to only the

necessary actions and resources within the database, minimizing the risk of unauthorized operations.

I will activate encryption at rest for DynamoDB tables using AWS Key Management Service (KMS) keys to offer an extra degree of security. As a result, all data saved in DynamoDB will be encrypted, ensuring that even if someone gains access to the underlying storage, the data will remain secure and unreadable in the absence of the necessary decryption keys. For secure data transmission, I will ensure that my application communicates with DynamoDB over encrypted connections using HTTPS [20]. Encryption in transit will prevent data interception and tampering during transmission, guaranteeing data integrity and confidentiality.

As I have already connected my services using endpoints in my AWS VPC, I will also utilize VPC endpoints for DynamoDB. This approach allows my application to access DynamoDB securely without relying on the public internet, reducing the exposure to potential external threats.

By diligently implementing these security measures, I can enhance the overall security of my to-do list maker application, instill user confidence in the privacy of their data, and maintain a safe and trustworthy environment for all users. Security is a paramount concern, and I will continuously update and improve my security measures to adapt to new threats and ensure the long-term integrity of my application.

3. Was your application monolithic?

Yes, my application's database management DynamoDB is monolithic. To decouple the components in my solution and convert from a monolithic to a microservices architecture in terms of DynamoDB, I will implement the following changes and improvements:

Firstly, I will create separate DynamoDB tables for each microservice in my application. This approach ensures that each service has its own dedicated data store, providing better data independence and reducing the risk of data access conflicts between different microservices. For example: User Profile management will have one table, another table with tasks associate with users etc.

Next, I will adopt an event-driven architecture instead of tightly coupling my Lambda functions with specific tables. Whenever data in DynamoDB is modified, I will trigger events using DynamoDB Streams. These events can then be processed by Lambda functions or other microservices to take relevant actions. This will promote loose coupling and allow for easier scaling and maintenance of individual components.

I will also carefully analyze the access patterns of each microservice and design DynamoDB tables accordingly. By using appropriate partition keys and secondary indexes, I can ensure efficient data retrieval and improve overall performance. To handle varying workloads on each DynamoDB table as I transition to a microservices architecture, I will enable auto scaling for the tables. This feature will automatically adjust read and write capacity based on demand, allowing each microservice to scale independently based on its own requirements.

For certain microservices with sporadic or unpredictable workloads, I will consider using DynamoDB's On-Demand mode. This mode automatically scales the capacity to handle the actual request rate, which can be cost-effective for microservices that experience varying levels of activity.

By implementing these changes and improvements in the context of DynamoDB, I can effectively decouple the components in my architecture, promote better scalability, enhance fault tolerance, and achieve a more efficient and manageable microservices-based application. These changes will enable each microservice to operate independently and provide me with more flexibility in scaling and maintaining my to-do list maker application.

4. **Did you have a disaster plan?**

No, I did not have a disaster plan. As part of my disaster recovery plan for DynamoDB in my to-do list maker application, I will implement regular data backups using AWS Backup. **By setting up automated backups**, I can ensure that my DynamoDB tables are regularly backed up, providing a safety net in case of any data loss or corruption. These backups will enable me to restore the database to a previous state, allowing me to recover from accidental deletions or other data-related issues efficiently.

In addition to regular backups, I will also enable **point-in-time recovery for my DynamoDB** tables. This feature will provide an extra layer of protection against data loss by allowing me to restore the tables to any specific point within a specified recovery window. With point-in-time recovery, I can go back to a specific moment in time, further safeguarding my data against potential errors or failures [21].

To ensure the effectiveness of my disaster recovery plan, I will conduct **periodic testing and drills**. By simulating disaster scenarios and performing recovery drills, I can evaluate the performance of my recovery procedures and identify any weaknesses or gaps in the plan. This proactive approach will enable me to make necessary improvements and adjustments to enhance the overall resilience and reliability of my disaster recovery strategy.

By taking these measures, I can be confident that my DynamoDB data is well-protected, and my application is prepared to recover swiftly from any unforeseen events. A robust disaster recovery plan will not only ensure the continuity of my to-do list maker application but also provide peace of mind in handling potential data-related incidents.

5. **Were the services highly available and reliable?**

I have not configured options to make DynamoDB highly available. To ensure that DynamoDB is highly available and reliable, I will implement several strategies to enhance its performance and resilience.

Firstly, I will utilize DynamoDB's built-in features for high availability. DynamoDB is designed to be highly available by default, with data automatically distributed across multiple Availability Zones (AZs) within an AWS region. This means that even if one AZ becomes unavailable due to hardware or network issues, DynamoDB will continue to operate without any disruption. By

leveraging multi-AZ deployments, I can significantly reduce the risk of data loss and downtime.

Additionally, I will enable DynamoDB's Global Tables to achieve global replication of data. This feature allows me to replicate DynamoDB tables across multiple AWS regions, providing low-latency access to data for users located in different parts of the world. Global Tables not only improve data availability but also reduce latency for geographically distributed users, enhancing the overall user experience [22].

To further enhance reliability, I will implement DynamoDB auto-scaling. With auto-scaling, DynamoDB can automatically adjust its provisioned capacity to handle fluctuations in read and write requests. During periods of high demand, the database will automatically scale up to accommodate increased traffic, ensuring optimal performance for users without the need for manual intervention [18].

I will also implement robust error handling and retries in my application code to handle any temporary failures or throttling issues that may occur with DynamoDB. By implementing exponential backoff and retry mechanisms, I can ensure that my application can gracefully handle any transient issues and continue to function smoothly.|

Moreover, Furthermore, I will use Amazon CloudWatch to check DynamoDB's performance and health on a regular basis. CloudWatch enables me to configure alarms and notifications for important performance parameters like read and write capacity units, allowing me to proactively identify possible issues and take corrective steps [6].

I can ensure that DynamoDB remains highly available and reliable in my to-do list maker application by following these methods and best practises. The combination of multi-AZ deployments, Global Tables, auto-scaling, error handling, and proactive monitoring will result in a robust and resilient database system that will provide a consistent and dependable user experience.

6. Did you consider the cost of the services?

Yes, I have consider some cost to use DynamoDB but that was rough estimate using Cost Explore**.** Several tactics will be used to improve the cost effectiveness of DynamoDB in my to-do list builder application. First, I will carefully examine the consumption patterns of my application and change the supplied read and write capacity accordingly. I may avoid overprovisioning and modify capacity to match actual demand by closely monitoring the workload, hence optimising expenses. In addition, I plan to use DynamoDB's on-demand option, which allows me to pay per request rather than pre-provisioning capacity. This could be more cost-effective for my application, especially if it sees erratic traffic.

Furthermore, I will implement caching mechanisms to reduce the number of read operations on DynamoDB. By using in-memory caching solutions like **Amazon ElastiCache** or client-side caching, I can minimize the need for frequent and costly data requests to DynamoDB, resulting in potential cost savings [23]. I will also pay close attention to my data model and ensure its optimization to reduce unnecessary attributes and indexes that could lead to increased read operations. An efficient data model will help lower the amount of data read and written,

contributing to overall cost optimization.

As part of my cost optimization strategy, I will regularly monitor my DynamoDB usage and analyze CloudWatch metrics. This will help me identify any inefficient queries or data requests and make necessary optimizations to my application code. By being proactive in identifying and addressing inefficiencies, I can minimize read and write operations, ultimately reducing DynamoDB costs [6].

Finally, I will consider archiving infrequently accessed data to a more cost-effective storage solution like Amazon S3. By offloading rarely used data from DynamoDB, I can further reduce the number of read operations, leading to additional cost savings. Overall, these strategies will enable me to make the most cost-effective use of DynamoDB while maintaining the reliability and performance of my to-do list maker application.

7. **Did you use monitoring and/or logging tools to give you graphs, messages, metrics, or alerts of the services?**

No, I have not used any monitoring or logging tools, to ensure the optimal performance and reliability of DynamoDB, I will use specific monitoring and logging tools to gain valuable insights into the database's operations. Firstly, I will use Amazon CloudWatch to monitor DynamoDB tables and metrics. By integrating CloudWatch with DynamoDB, I can track important metrics like read and write capacity units, throttling events, and latency. This will enable me to understand the database's workload and identify any potential bottlenecks or performance issues. CloudWatch alarms will be set up to alert me when certain thresholds are breached, allowing me to take proactive measures to maintain the database's efficiency.

In addition to CloudWatch, I will enable DynamoDB Streams, which will serve as a valuable logging tool. DynamoDB Streams captures a time-ordered sequence of item-level modifications made to the tables, such as inserts, updates, and deletions. By enabling DynamoDB Streams, I can implement real-time data analysis, trigger Lambda functions based on changes in the database, and track the history of data modifications. This will be particularly useful for auditing and maintaining a detailed record of all user interactions with the to-do list application.

Furthermore, to consolidate and analyze the logs generated by DynamoDB and other AWS services, I will use Amazon CloudTrail. CloudTrail records all API calls made to DynamoDB, providing a comprehensive audit trail of actions taken within the database. By configuring CloudTrail, I can monitor the access to my DynamoDB tables and identify any unauthorized access attempts or suspicious activities [8].

For advanced analytics and visualization of the log data, I will integrate CloudTrail with Amazon Elasticsearch and Kibana. This combination will allow me to create interactive dashboards, conduct detailed searches, and visualize the log data in real-time. This level of analysis will help me quickly identify any potential security threats, gain insights into usage patterns, and make data-driven decisions to optimize the application's performance.

By utilizing these monitoring and logging tools for DynamoDB, I will have a comprehensive

view of the database's operations, enabling me to proactively address any issues, maintain data integrity, and ensure the reliability of the to-do list maker application.

Storage:

1. **What cloud services did you use?**

   In my to-do list maker, I did not use any specific storage service for my application as I only managed my data using DynamoDB.

   - **If you didn't use any services in this category, why? If you think now you will use some services in this category, what are they and why?**

     In my to-do list maker application, I used Amazon DynamoDB as the primary storage service. DynamoDB is a fully managed NoSQL database provided by AWS, and it was well-suited for storing all the details of users and their tasks. I chose DynamoDB because it can handle flexible, semi-structured data and automatically scales to meet the demands of the application as the user base grows. Moreover, DynamoDB provides low-latency access to data, which is crucial for real-time interaction with the to-do lists.

     While I mentioned other storage services such as Amazon S3, Amazon RDS, Amazon ElastiCache, and Amazon EFS in my previous response, I want to clarify that I did not use these specific services in the current version of my application. The decision not to use these services was based on the current feature set and use cases of the to-do list maker. DynamoDB fulfilled the data storage needs adequately, and the other mentioned storage services were not directly relevant to the current functionality of the application.

     I will suggest using Amazon S3 in my to-do list maker application to enhance its capabilities and allow users to upload and store attachments or images related to their tasks. With S3, I can provide a feature that enables users to easily attach files or images when creating or editing their to-do list items. For instance, if a user wants to add a document related to a specific task or attach an image for reference, they can do so by uploading the file through the application's interface. This will help for to set to do list from photos features, previously I did not save any photos to create task list but using S3 I can make a record of photos that can help later.

     By implementing Amazon S3, I will create a dedicated S3 bucket for the application to store these user-generated files securely. Each file will be associated with the corresponding to-do list item, allowing users to access and download the attachments whenever they need them. This will not only enhance the functionality of the to-do list maker but also provide a seamless and organized way for users to manage supporting documents or visual references for their tasks.

     Additionally, Amazon S3's high scalability and durability ensure that the files uploaded by users will be reliably stored and accessible at any time. S3 provides a robust infrastructure for data storage, backup, and retrieval, making it an ideal choice for handling user-generated content in the to-do list maker application [24]. With this

integration, users will have a more comprehensive and versatile tool to manage their tasks efficiently, making the application even more valuable and user-friendly.

General questions:

1. **Assuming your application will be commercialized and your users will be globally distributed, what would you do to enhance the security, quick and easy access, and performance of the application? Pick one or two cloud services to explain for security, quick and easy access, and performance, respectively.**

   I am suggesting 2 cloud services that I haven't used yet in my application for security, quick and easy access, and performance but that services will enhance mentioned things for application.

   Amazon CloudFront for Performance: In a globally distributed application like my to-do list maker, performance is crucial to provide a seamless user experience for users across different regions. Amazon CloudFront, as a Content Delivery Network (CDN), excels at improving performance by reducing latency and accelerating content delivery [25].

   As users access my application from different parts of the world, CloudFront caches static assets like images, CSS, and JavaScript files in its edge locations closest to the users. This means that subsequent requests for the same content can be served from nearby edge locations, reducing the round-trip time and minimizing the latency**.**

   For example, if a user in Europe accesses my to do list application, CloudFront will cache the relevant assets in an edge location in Europe such as to-do list, task details, deadline and reminder date. If another user from Asia accesses the same assets, they will be served from an edge location in Asia. This distribution of content significantly speeds up the delivery process, resulting in faster load times and a more responsive application. They will get result of their "to do list" in faster way. To do list is one which people want to see at any time and with less time of loading.

   By integrating CloudFront with my React frontend hosted on Heroku and the backend on Elastic Beanstalk, I can accelerate the delivery of my application's assets and data. CloudFront can cache the frontend files and API responses, reducing the need for repeated requests to the backend servers. Additionally, CloudFront supports real-time content updates using cache invalidations or Time To Live (TTL) settings, ensuring users always access the latest version of my application.

   Amazon Cognito for Security and Quick User Access: Amazon Cognito is an excellent choice for enhancing the security and user access management of my to-do list application. Security is paramount, especially when users' task lists and personal information are involved.

   I can develop safe user authentication and authorization by integrating Amazon Cognito without having to create elaborate authentication systems from start. Cognito supports a number of authentication techniques, including my personal user directories and social identity providers like Google and Facebook [26].

This versatility allows users to sign in using their existing credentials from popular platforms, streamlining the sign-up process and increasing user adoption. Quick and easy access to the application fosters a positive user experience, reducing friction during the login process and increasing user engagement.

Furthermore, Amazon Cognito offers features like Multi-Factor Authentication (MFA) and User Pool Groups [26]. MFA adds an extra layer of security by requiring users to provide an additional authentication factor, such as a one-time password or fingerprint. User Pool Groups allow me to define different access permissions based on user roles, enabling me to control access to specific features or data within the application. I can define User Pool Groups based on user roles, such as regular users and admin users. For example, regular users can log in, add and manage their to-do lists, while admin users have additional privileges to manage user accounts and access more advanced features.

By leveraging the capabilities of Amazon CloudFront for performance and Amazon Cognito for security and user access management, my to-do list application will be well-equipped to handle the demands of a global user base. Users will experience faster load times, secure authentication, and an intuitive sign-up process, contributing to a positive and reliable user experience for my application.

2. **When you design your cloud architectures/solutions, should you try to choose managed or unmanaged services?**

When designing cloud architectures or solutions, the choice between managed and unmanaged services often involves striking a balance between simplicity, control, and cost-effectiveness. In many cases, a **combination of both managed and unmanaged services** can be the most practical approach to meet specific requirements and achieve desired outcomes and I try to achieve combination of both instead of choosing one type of service.

By thoughtfully combining managed and unmanaged services, I can leverage the best of both worlds. **Managed services** simplify the **management of critical components, freeing up time for application development and ensuring high availability**. **Unmanaged services**, on the other hand, **offer flexibility and customization** for specific needs, enabling me to tailor the infrastructure to meet unique requirements.

In the end, my cloud architecture's unique demands for each component should determine whether I use managed or unmanaged services. I can develop a well-optimized solution that yields the greatest outcomes for my cloud-based application by carefully weighing the trade-offs between simplicity of use, control, and cost.

- **Explain the pros and cons of managed and unmanaged services. You can Google the answers, but you must summarize using your own words.**

  Managed Service Benefits:

  Managed services in AWS offer numerous advantages that make them an attractive choice for many businesses and developers. One of the key benefits is **simplified**

management. With managed services, developers can focus on building applications without the need to worry about the underlying infrastructure. AWS takes care of routine tasks such as patching, backups, and scaling, which reduces the administrative burden and allows teams to be more productive and efficient.

Another significant advantage of managed services is their **high availability and resilience**. AWS manages the underlying infrastructure of these services, ensuring that they are designed to tolerate failures and provide better uptime compared to self-managed alternatives. This means that businesses can rely on these services to deliver consistent performance and availability, even during peak usage periods [27].

Another useful feature provided by many managed services is **automatic scaling**. The services will be able to automatically alter their resources in response to demand. The services can scale up to accommodate more traffic during busy periods, ensuring that users receive the best performance possible. In contrast, the services can scale back to reduce costs during slow periods, making them more effective and economical [27].

Security and compliance are paramount considerations for any business, and managed services in AWS help address these concerns. AWS takes responsibility for the security of the underlying infrastructure and regularly updates services to address security vulnerabilities. Managed services often come with **built-in security features**, providing businesses with a strong foundation for their applications to meet industry compliance standards and regulatory requirements [27].

In terms **of cost-effectiveness**, managed services can be a compelling choice, particularly for small to medium-sized businesses. By using managed services, businesses can avoid the need to invest in hardware, maintenance, and operational staff [27]. They can also take advantage of the pay-as-you-go pricing model offered by AWS, which allows them to pay only for the resources they consume, making it a cost-efficient option for their specific needs.

Overall, managed services in AWS offer advantages like simplified management, high availability, automatic scaling, security, and cost-effectiveness that make them a strong and practical option for companies looking to use cloud technology without having to deal with the challenges of managing the underlying infrastructure.

Managed Service Drawbacks:

Despite their numerous advantages, managed services in AWS also come with some drawbacks that businesses should consider before making their architectural decisions. One of the main drawbacks is **limited customization**. While managed services are designed to be user-friendly and easy to deploy, this simplicity can sometimes result in a **lack of flexibility** when it comes to fine-tuning or customizing the service to meet specific requirements. Advanced configurations and settings may not be accessible, and users may find themselves constrained by the service's predefined parameters [28].

Vendor lock-in is another concern associated with using managed services. By adopting a specific cloud provider's managed services and APIs, businesses become tightly integrated with that provider's ecosystem. This can make it challenging to switch to another cloud provider in the future without significant rework and migration efforts. Consequently, businesses should carefully consider the potential long-term implications of vendor lock-in before fully committing to a particular set of managed services.

Another aspect to consider is the potential for **delayed feature updates**. Managed services may not always support the latest features or software versions as quickly as self-managed solutions [28]. Users might have to wait for the cloud provider to release updates or new features, which could delay the adoption of cutting-edge functionalities in their applications.

Managed **services may not be the most inexpensive choice for high-throughput, large-scale applications**, even though they can be cost-effective for smaller workloads and applications [28]. Self-managed alternatives or serverless architectures may present better potential for cost optimization in such circumstances. Organizations should carefully assess the financial costs of employing managed services for their unique workload and use patterns.

The reliance on the provision of services is another factor. Since AWS is in charge of the services' underlying infrastructure, any outages or disturbances there could have an immediate effect on the application's availability. Even though AWS offers reliable infrastructure, companies should still prepare for emergencies and think about creating multi-region designs to maintain high availability and lower the risk of service outages that could result in downtime.

In summary, while managed services in AWS offer numerous benefits in terms of simplicity, high availability, and security, they also come with certain trade-offs, such as limited customization, potential vendor lock-in, and delayed feature updates.

Unmanaged Services Benefits:

Full Control and Customization: Users in AWS have complete control over the underlying infrastructure thanks to unmanaged services. This degree of control enables highly specialized configurations and installations, which makes it perfect for applications with particular needs in terms of performance [29]. For instance, organizations with specialized and complicated workloads can need fine-grained management over their resources to operate at peak efficiency. They can precisely design and implement the infrastructure to suit their particular use cases by using unmanaged services.

Flexibility in Software and Configuration: The choice and configuration of software is very flexible with unmanaged services. The precise software and tool versions that users want to use can be chosen, ensuring that they are compatible with the requirements of their applications [29]. This flexibility enables the use of certain libraries, frameworks, or

distinctive software components that might not be provided by managed services. Developers are thus free to create and publish apps that are precisely tailored to meet their particular needs.

Avoidance of Vendor Lock-In: By opting for unmanaged services, users can avoid vendor lock-in, which is the risk of becoming dependent on a specific cloud provider's proprietary services and APIs [29]. This independence offers businesses the freedom to move their applications and workloads across different cloud providers or even adopt a multi-cloud strategy. It also provides a safety net against unexpected changes in cloud service offerings or pricing models.

Cost-Effective for Large Workloads: Unmanaged services may be more affordable, particularly for large-scale workloads with high resource requirements. Since users have complete control over the infrastructure, they may tailor resource use to the demands of the workload [29]. As resources can be distributed more effectively, reducing irrational costs associated with overprovisioning, this optimization may result in cost savings. The affordability of unmanaged services can be advantageous for businesses with significant computing needs.

Immediate Access to Latest Features: Users can have rapid access to the most recent software versions and features with unmanaged services. Users have the ability to upgrade software components as soon as new versions are published, in contrast to managed services, where updates are managed by the cloud provider. By doing so, they can benefit from the most recent developments and improvements without having to wait for the cloud provider's update cycle. Instant access to new features can enhance the functionality and performance of an application [29].

In summary, unmanaged services in AWS offer users unparalleled control, customization, and flexibility, making them suitable for applications with unique requirements and specific needs. They provide cost-effective solutions for large workloads and grant organizations the ability to avoid vendor lock-in, while also allowing access to the latest software features without delay. However, it's important to note that unmanaged services also come with the responsibility of managing and maintaining the infrastructure, which may require more expertise and effort on the part of users. Properly weighing the benefits and challenges of unmanaged services is crucial when designing cloud architectures and solutions.

Unmanaged Services drawbacks:

Increased Operational Complexity: Utilizing unmanaged services in AWS can lead to increased operational complexity. Unlike managed services that abstract away many operational tasks, unmanaged services require users to handle various aspects of infrastructure management, including patching, backups, and scaling. This added complexity can be challenging for organizations with limited cloud expertise or resources [30]. It may require dedicated IT staff with specialized knowledge to effectively manage and maintain the infrastructure, adding to operational costs and potential risks.

Responsibility for Patching and Maintenance: Users of unmanaged services are responsible for deploying patches, updates, and completing maintenance chores on the underlying infrastructure and software on a regular basis. This obligation necessitates careful attention and prompt action in order to keep the system secure and up to date. Failure to properly maintain the infrastructure exposes the application to security risks, performance concerns, or potential downtime, making patch management a vital part of using unmanaged services [30].

Potential for Human Error and Misconfiguration: The higher level of control provided by unmanaged services also introduces the potential for human error and misconfiguration. Since users have direct control over the setup and management of the infrastructure, any mistakes or misconfigurations can have significant consequences [30]. Misconfigurations could lead to security breaches, performance degradation, or application instability. Proper training, documentation, and adherence to best practices are essential to mitigate the risk of human errors in managing unmanaged services.

Higher Administrative Burden: Unmanaged services may have a higher administrative load due to the lack of automated administration and abstraction. Users must actively monitor, debug, and assure the infrastructure's high availability. This may necessitate more work and resources than managed services, which handle many basic activities automatically by the cloud provider [30]. The additional administrative workload may be a source of concern for organizations with limited IT employees or when core application development is a high priority.

Limited Built-In Security Features: While managed services often come with built-in security features and compliance measures, unmanaged services may lack some of these out-of-the-box security capabilities. Users must take additional measures to implement security controls and best practices to safeguard their applications and data effectively. This includes managing access controls, encryption, and ensuring compliance with industry regulations [30]. Failure to address security requirements adequately could expose the application to potential security breaches and data compromises.

To summarize, while AWS unmanaged services give customers unrivalled power and flexibility, they also come with greater obligations and possible issues. When deciding between managed and unmanaged services, organizations must carefully assess their operational capabilities and resource availability. Unmanaged services can offer bespoke solutions and cost-saving options to organizations with the skills and capacity to manage and safeguard their infrastructure properly. Those with limited resources and a preference for a hands-off approach, on the other hand, may find managed services to be a better fit for their cloud infrastructures.

- **Using your explanations in question 1), analyze if the services in your previous report were managed or unmanaged. If you think there are managed (unmanaged) services that should be replaced by unmanaged (managed) services, pick one or two to explain why. If there is none to change, pick one or two to explain why.**

Here is the analysis of the services in your application and whether they are managed or unmanaged:

- Managed Services:
  Amazon Elastic Beanstalk - Managed
  Amazon SNS (Simple Notification Service) - Managed
  Amazon DynamoDB - Managed
  Amazon Textract - Managed
- Unmanaged Services:

  React Frontend Code on Heroku – Unmanaged (The React frontend hosted on Heroku is considered unmanaged as I have control over the deployment and configuration of the frontend application. Heroku provides the platform, but I am responsible for managing the frontend code.)

  Node.js Backend Code on Elastic Beanstalk - Unmanaged (Although Elastic Beanstalk is a managed service, the Node.js backend code I have deployed on it is unmanaged. I am responsible for managing the backend code and its configurations.)

For my application, I will suggest to change the Node.js backend code from Elastic Beanstalk (managed) to Amazon EC2 instances (unmanaged).

By utilizing Amazon Elastic Beanstalk, I will get the benefits of a **fully managed platform-as-a-service (PaaS)** offering by AWS. It will abstract away the underlying infrastructure details, automating application deployment, scaling, and management. With Elastic Beanstalk, I only need to provide my application code, and the platform will handle the deployment, load balancing, auto-scaling, and other operational tasks. This ease of use will save me time and effort, allowing me to focus on developing and improving my to-do list application's features.

By switching to Amazon EC2 instances, I will gain complete control over the virtual machines running my backend Node.js code. This added control will allow me to customize the server settings, install specific software or services, and manage the entire infrastructure according to my exact requirements. This flexibility will be advantageous if I need to implement specific configurations that are not supported by Elastic Beanstalk or if I have advanced use cases that demand more granular control.

In short, I could consider changing the Node.js backend code from Elastic Beanstalk (managed) to Amazon EC2 instances (unmanaged). With EC2 instances, I have more control over the virtual machines, allowing me to customize the server settings and install specific software or services required for my application. This additional control might be beneficial if I require more flexibility in managing the backend environment or if there are specific configurations that are not supported by Elastic Beanstalk.

Considering the benefits of managed services suggestion is, it may be beneficial to change the **React frontend deployment from Heroku (unmanaged) to AWS Amplify**

Console (managed). AWS Amplify Console is a continuous deployment and hosting service that simplifies the process of deploying frontend applications. By using Amplify Console, I can leverage its built-in integrations with AWS services and the ability to automate deployment based on changes to the code repository. This shift to a managed service can reduce the operational overhead of managing the frontend deployment and ensure seamless integration with other AWS services in my application.

Bibliography

[1] Amazon, "AWS Elastic Beantalk," Amazon, [Online]. Available: https://aws.amazon.com/elasticbeanstalk/. [Accessed 05 July 2023].

[2] Amazon, "AWS Lambda," Amazon, [Online]. Available: https://aws.amazon.com/lambda/. [Accessed 10 July 2023].

[3] Amazon, "Amazon EC2," Amazon, [Online]. Available: Amazon EC2. [Accessed 07 July 2023].

[4] Amazon, "AWS Elastic Beanstalk security," Amazon, [Online]. Available: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/security.html. [Accessed 07 July 2023].

[5] Amazon, "Security in AWS Lambda," Amazon, [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-security.html. [Accessed 07 July 2023].

[6] Amazon, "What is Amazon CloudWatch?," Amazon, [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html. [Accessed 08 July 2023].

[7] Amazon, "AWS Cost Explorer," Amazon, [Online]. Available: https://aws.amazon.com/aws-cost-management/aws-cost-explorer/. [Accessed 09 July 2023].

[8] Amazon, "AWS CloudTrail," Amazon, [Online]. Available: https://aws.amazon.com/cloudtrail/. [Accessed 08 July 2023].

[9] Amazon, "Amazon Virtual Private Cloud," Amazon, [Online]. Available: https://aws.amazon.com/vpc/. [Accessed 09 July 2023].

[10] Amazon, "Control traffic to subnets using network ACLs," Amazon, [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html. [Accessed 10 July 2023].

[11] Amazon, "Infrastructure security in Amazon VPC," Amazon, [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/infrastructure-security.html. [Accessed 10 July 2023].

[12] Amazon, "Security in Amazon Virtual Private Cloud," Amazon, [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/security.html. [Accessed 11 July 2023].

[13] Amazon, "Connect to the internet using an internet gateway," Amazon. [Online]. [Accessed 02 July 2023].

[14] Amazon, "Associate Elastic IP addresses with resources in your VPC," Amazon, [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/vpc-eips.html. [Accessed 04 July 2023].

[15] Amazon, "Security groups," Amazon, [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/security-groups.html. [Accessed 14 July 2023].

[16] Amazon, "Connect your VPC to services using AWS PrivateLink," Amazon, [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/endpoint-services-overview.html. [Accessed 05 July 2023].

[17] Amazon, "Logging IP traffic using VPC Flow Logs," Amazon, [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/flow-logs.html. [Accessed 04 July 2023].

[18] Amazon, "What is Amazon DynamoDB?," Amazon, [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html. [Accessed 04 July 2023].

[19] Amazon, "Security and compliance in Amazon DynamoDB," Amazon, [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/security.html. [Accessed 10 July 2023].

[20] Amazon, "Amazon," Security and compliance in Amazon DynamoDB, [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ddb-security-iam.awsmanpol.html. [Accessed 10 July 2023].

[21] Amazon, "Resilience and disaster recovery in Amazon DynamoDB," Amazon, [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/disaster-recovery-resiliency.html. [Accessed 10 July 2023].

[22] Amazon, "Best practices for DynamoDB global table design," Amazon, [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-global-table-design.html. [Accessed 09 July 2023].

[23] Amazon, "Amazon ElastiCache," Amazon , [Online]. Available: https://aws.amazon.com/elasticache/. [Accessed 08 July 2023].

[24] Amazon, "What is Amazon S3?," Amazon, [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html. [Accessed 04 July 2023].

[25] Amazon, "What is Amazon CloudFront?," Amazon, [Online]. Available: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html. [Accessed 11 July 2023].

[26] Amazon, "Amazon Cognito," Amazon , [Online]. Available: https://aws.amazon.com/cognito/. [Accessed 07 July 2023].

[27] DinoCloud, "Benefits of Using AWS Managed Services Providers," DinoCloud, [Online]. Available: https://dinocloud.co/aws-managed-services-provider/. [Accessed 02 July 2023].

[28] M. Perry, "Should Your Startup Use AWS Managed Services?," Qovery, 15 March 2022. [Online]. Available: https://www.qovery.com/blog/should-your-startup-use-aws-managed-services. [Accessed 2 July 2023].

[29] N. Foster, "Managed vs. Unmanaged Cloud Services – What are the differences?," Ace Cloud, 07 December 2020. [Online]. Available: https://www.acecloudhosting.com/blog/managed-vs-unmanaged-cloud-services/. [Accessed 05 July 2023].

[30] M. Osman, "Managed vs. unmanaged cloud hosting: the differences explained," Nexcess, 13 April 2023. [Online]. Available: https://www.nexcess.net/blog/managed-vs-unmanaged-cloud/. [Accessed 06 July 2023].

[31] Amazon, "Infrastructure security in Amazon EC2," Amazon, [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/infrastructure-security.html. [Accessed 07 July 2023].