DALHOUSIE UNIVERITY

Cloud Term Project Report

Pratik Sakaria
pratik.sakaria@dal.ca
B00954261

What You Built and Its Purpose

The 'Journal' web application is a Flask-based platform designed to enable users to add, edit, and delete journal entries seamlessly. Additionally, the application offers robust login and signup functionalities, ensuring secure access for users. A notable feature of the 'Journal' app is the ability for users to upload images along with their entries, enhancing the visual aspect of journaling. This application caters to individuals seeking a digital platform to organize their thoughts, memories, and experiences in a user-friendly and visually appealing manner. The project aligns with the cloud computing course requirements, showcasing a practical application of various AWS services in developing a functional web application.

How You Met Your Menu Item Requirements

Selected Services and Comparison:

1. **AWS RDS Aurora**:

   - **Why it was Chosen**: Provides a scalable, high-performance database service with MySQL and PostgreSQL compatibility, which is ideal for transactional data storage like user information and journal entries.
   - **Alternatives**: Amazon DynamoDB, a NoSQL database service, could have been used for flexible schema and scalability. However, RDS Aurora was chosen for its relational database features and ease of querying complex data, which suits the structured data model of user and journal entries better. [1]

2. **AWS Secrets Manager**:

   - **Why it was Chosen**: Securely manages and rotates credentials for connected services, enhancing security by automating the management of database credentials.
   - **Alternatives**: AWS Parameter Store also offers secrets management with integration into AWS IAM for access control but lacks the advanced rotation capabilities and direct integration with RDS which is why the Secrets Manager was chosen.[2]

3. **Lambda Functions**:

   - **Why it was Chosen**: Allows for serverless image processing, which is cost-effective and scales automatically with the application load.
   - **Alternatives**: AWS EC2 instances could perform image processing but would require manual scaling and incur higher costs due to continuous running.

4. **AWS SNS**:

   - **Why it was Chosen**: Provides a simple way to send notifications or emails, perfect for sending welcome emails upon user sign-up.

- **Alternatives**: AWS SES is a direct email sending service that offers more control over email sending but is more complex to set up for simple notification use cases. In order to keep the configurations to a minimal I chose to implement SNS instead of SES.

5. **AWS VPC**:

   - **Why it was Chosen**: Ensures that your AWS resources run in a network isolated from other networks, enhancing security by segregating the database in private subnets while allowing the web application public access.
   - **Alternatives**: Using default AWS networking without a custom VPC might simplify setup but would not provide the same level of security and network control.

6. **AWS Elastic Beanstalk and Docker**:

   - **Why it was Chosen**: Provides an easy-to-use service for deploying and scaling web applications, with Docker offering containerization for consistent environments across development and production.
   - **Alternatives**: Amazon ECS or Kubernetes (EKS) offer more control over container orchestration but require more complex setup and management.

Deployment Model

For the "Journal" Flask web application, the chosen deployment model is the **Public Cloud**, specifically utilizing a range of services provided by Amazon Web Services (AWS). This model was selected due to its scalability, reliability, cost-effectiveness, and the breadth of services it offers, which align well with the requirements of the application. Below is a detailed description and the rationale behind this choice.[4]

Description of the Deployment Model

The Public Cloud model involves leveraging cloud computing services offered by third-party providers over the public internet. In this scenario, AWS serves as the cloud provider, hosting the application and its associated components, including databases, serverless functions, email notification services, and more. The infrastructure is owned and managed by AWS, while the application owner pays for the services used on a pay-as-you-go basis.

Reasoning for Choosing the Public Cloud Model

- **Managed Services**: AWS offers a wide array of managed services like RDS Aurora for databases, Lambda for serverless computing, and Elastic Beanstalk for application deployment. These services abstract away much of the underlying infrastructure management, allowing the development team to focus on building and enhancing application features rather than managing servers and databases.

- **Reliability**: AWS's global infrastructure offers high availability and fault tolerance. By deploying the "Journal" application on AWS, it inherits these characteristics, ensuring the application is always accessible to its users.

Delivery Model

the chosen delivery model combines aspects of **Platform as a Service (PaaS)** and **Function as a Service (FaaS)**. This hybrid approach was selected for its flexibility, scalability, and ability to streamline development and deployment processes. Below is a detailed explanation and the rationale behind this choice.

Platform as a Service (PaaS)

AWS Elastic Beanstalk, a PaaS offering, was utilized for deploying and managing the web application infrastructure. This service abstracts away much of the underlying server and network infrastructure management, providing a platform on which the application runs.

- **Why it was Chosen**:
  - **Simplified Deployment**: Elastic Beanstalk automates the deployment process, from capacity provisioning and load balancing to auto-scaling. This allows the development team to focus on application development rather than infrastructure setup.
  - **Language and Framework Support**: Elastic Beanstalk offers first-class support for a wide range of programming languages and frameworks, including Flask for Python, making it an ideal platform for the "Journal" application.
  - **Scalability and Manageability**: Automatically handles scaling of the application according to the defined policies and metrics, ensuring the application remains responsive under varying loads without manual intervention.

Function as a Service (FaaS)

AWS Lambda, a FaaS solution, was chosen for image processing tasks, such as scaling and resizing images uploaded by the users.

- **Why it was Chosen**:
  - **Event-driven Execution**: Lambda functions are triggered automatically by AWS services (e.g., S3 for image uploads), making it an efficient way to process data on-demand without provisioning or managing servers.

Rationale for the Chosen Delivery Model

The combination of PaaS and FaaS as the delivery model for the "Journal" application was driven by several key considerations:

- **Development Efficiency**: Leveraging PaaS and FaaS minimizes the amount of boilerplate code and infrastructure management tasks, allowing me to concentrate on creating and improving application features.
- **Scalability**: Both Elastic Beanstalk and Lambda provide automatic scaling capabilities. This ensures that both the application and its associated tasks (like image processing) can handle varying loads efficiently.

System Architecture

The architecture of the "Journal" Flask web application predominantly aligns with the **Dynamic Scalability Architecture**, **Load Balanced Virtual Server Instances Architecture**, and **Rapid Provisioning Architecture**. These architectural choices reflect the application's requirements for scalability, performance, and agility in deployment and operations.

Dynamic Scalability Architecture

This architecture is pivotal for web applications like "Journal" that experience variable user traffic. By leveraging AWS Elastic Beanstalk and AWS Lambda, the application can automatically scale resources in response to demand. This ensures that the system remains responsive during peak usage periods without incurring unnecessary costs during times of low demand.

- **Analysis**: This choice is wise for maintaining performance and cost-effectiveness. The potential flaw could be over-reliance on AWS's scaling algorithms, which may not scale down as quickly as possible, potentially incurring slightly higher costs during periods of decreasing load.

Load Balanced Virtual Server Instances Architecture

The application utilizes AWS Elastic Beanstalk, which inherently provides load balancing across multiple instances. This ensures high availability and uniform distribution of incoming traffic, crucial for user experience and system reliability.

- **Analysis**: Opting for load balancing is a sound decision for enhancing application reliability and managing traffic spikes efficiently. However, it requires careful configuration of health checks and thresholds to ensure that traffic is distributed optimally across healthy instances.

Rapid Provisioning Architecture

Rapid provisioning is facilitated through the use of AWS services like Elastic Beanstalk for the application and RDS Aurora for the database, enabling quick deployment and updates. This agility supports continuous integration and delivery practices, allowing for swift iterations based on user feedback or requirements changes.

- **Analysis**: Embracing rapid provisioning fosters a development culture that can quickly adapt to changes, a critical advantage in web application development. The potential downside might be in scenarios where rapid changes lead to insufficient testing or unanticipated costs due to frequent deployments.

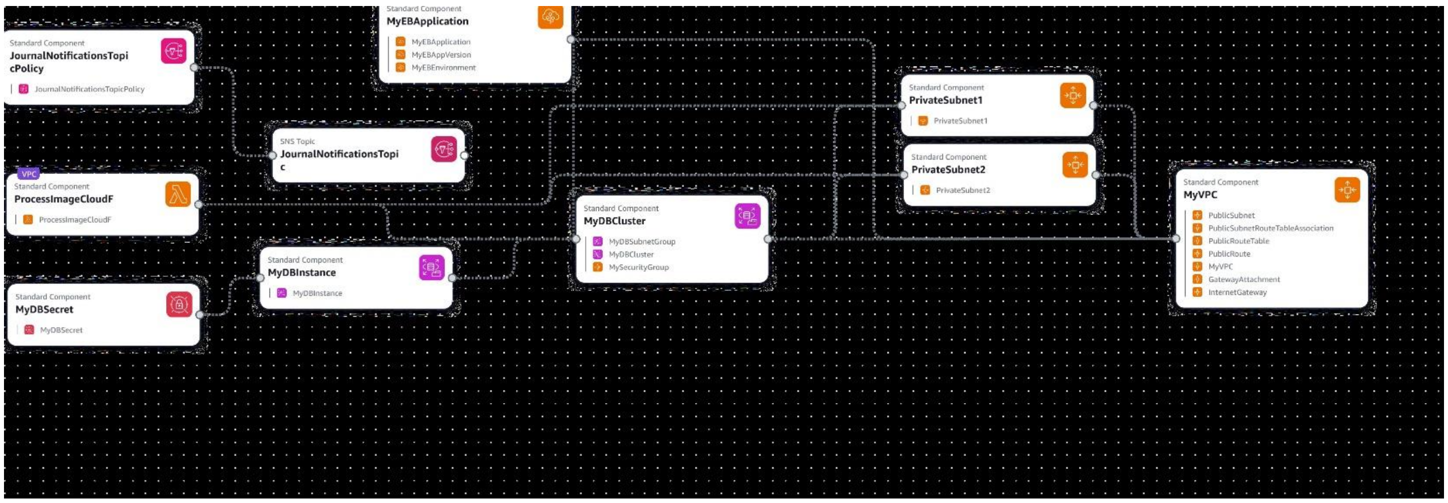

Figure 1: Service Architecture captured in Application Composer

Final Architecture Overview

The architecture is designed to be scalable, secure, and resilient, leveraging various AWS services to build a robust backend for a Flask-based web application. Here's how the components integrate:

- **Frontend/Client Side**: Users interact with the "Journal" application through a web interface, which communicates with the backend over HTTPS, ensuring secure data transmission.
- **AWS Elastic Beanstalk with Docker**: Manages the deployment, scaling, and health monitoring of the Flask application, which is containerized with Docker for consistency across development and production environments.
- **AWS RDS Aurora**: Hosts the application's relational database in a scalable, managed environment, storing user accounts and journal entries.
- **AWS Lambda**: Processes image uploads by resizing and optimizing them for web use. Triggered by upload events, it runs code without provisioning or managing servers.
- **AWS SNS**: Handles sending email notifications, such as welcome emails to new users.
- **AWS VPC**: Encapsulates the entire infrastructure, providing network isolation. RDS Aurora resides in a private subnet, accessible only by the application in the public subnet, enhancing security.
- **AWS Secrets Manager**: Securely manages the database credentials, which the Flask application uses to connect to RDS Aurora.

Data Storage

Data is stored in:

- **AWS RDS Aurora**: User data and journal entries.
- **AWS S3 (implicitly through Lambda processing)**: Images uploaded by users, after being processed by Lambda for optimization and scaling.
- **AWS Secrets Manager**: Database credentials and possibly other sensitive configurations.

Programming Languages Used

- **Python (Flask)**: Chosen for its simplicity, readability, and the rich ecosystem of libraries, including Flask for web application development. Flask is used for routing, request handling, and rendering the web interface. Python is also used for AWS Lambda functions due to its support for serverless architectures and ease of integration with AWS SDKs.
- **JavaScript (optionally)**: For dynamic client-side behaviors in the web interface.

System Deployment

Deployment is managed through AWS Elastic Beanstalk, which automates:

- Launching EC2 instances to run Docker containers.
- Configuring load balancers and auto-scaling groups.
- Monitoring health and adjusting resources as needed. The Docker container, containing the Flask application, is built from a Dockerfile, which defines the environment, dependencies, and entry points. This container image is then pushed to a Docker Hub repository, from where Elastic Beanstalk pulls it during deployments.

Security Analysis

The security strategy for the "Journal" project is meticulously crafted to safeguard user data against unauthorized access, disclosure, and alterations at every stage within the system. By integrating encryption, access management, and network security measures, the application ensures a comprehensive protection framework for data both in transit and at rest. Here's an enhanced analysis of the project's security approach:

Data in Transit Security

- **HTTPS and TLS**: All data exchanged between the client's browser and the server is encrypted using HTTPS, underpinned by Transport Layer Security (TLS). This encryption protects data from interception, tampering, and eavesdropping during transmission.
- **Secure AWS Service Integration**: AWS Lambda, utilized for serverless image processing, employs AWS Identity and Access Management (IAM) roles to securely interact with other AWS services. IAM roles provide fine-grained access controls, ensuring that each component operates with the least privilege necessary to perform its function, thereby minimizing the potential impact of a security breach.

Data at Rest Security

- **Encrypted Database Storage**: AWS RDS Aurora is configured to use encryption at rest, safeguarding journal entries and user information stored in the database. This encryption is managed by AWS, utilizing strong encryption algorithms to prevent unauthorized data access.
- **Credential Management**: AWS Secrets Manager is employed to handle database credentials securely. It not only encrypts the credentials but also facilitates regular rotation, reducing the risk associated with credential leakage or misuse.

Additional Security Measures

- **Virtual Private Cloud (VPC)**: The deployment of services within an AWS VPC provides an additional layer of security. This isolated network environment allows for precise control over inbound and outbound traffic to the application and database, effectively mitigating the risk of direct attacks.
- **Security Groups and Network ACLs**: These are utilized within the VPC to enforce stricter control over traffic at both the instance and subnet levels. By defining allowable traffic patterns, these tools help in preventing unauthorized network access.
- **IAM Policies for Principle of Least Privilege**: The project adheres to the principle of least privilege by assigning IAM policies that grant only the permissions necessary for each service or user to perform their designated tasks. This minimizes potential damage from compromised credentials or insider threats.
- **Regular Security Audits and Compliance**: The project incorporates regular security assessments to identify and rectify vulnerabilities, ensuring compliance with best practices and industry standards for cloud security.

Cost Metrics Analysis

Up-front Costs

Minimal, as AWS services are pay-as-you-go. Initial costs involve provisioning resources and storage.

Ongoing Costs: Ongoing costs are largely variable, depending on the application's usage:

- **RDS Aurora**: Charges are based on instance size and hours run. Choosing an instance size that balances performance with cost is crucial.
- **AWS Lambda**: Priced per request and compute time. Since Lambda supports scaling, costs will align with the number of image processing operations.
- **AWS SNS**: Costs are incurred per email sent. However, for applications with a small number of users, this may remain minimal.
- **AWS Elastic Beanstalk**: While Elastic Beanstalk itself has no additional charge, underlying resources (EC2 instances, load balancer) incur costs based on their size and running time.
- **Data Transfer**: Networking costs can add up, especially if the application serves a large amount of data or operates across multiple regions.

Assuming that all the services are kept running 24/7 and that traffic to the website is not a lot initially. A very rough ball park figure estimate of $185.64 + Lambda compute costs is calculated. [3]

Additional Costs

- **Monitoring and Logging**: Services like AWS CloudWatch incur additional costs based on the amount of data ingested and stored for logs, events, and metrics.
- **Backup and Data Retention**: RDS snapshots and backups, along with any S3 storage used for logs or image storage, will contribute to the cost based on the storage used.

Cost-Saving Alternatives

- **Serverless Architecture**: Further leveraging serverless services for parts of the application not already using Lambda could reduce costs by eliminating the need to run and manage EC2 instances.
- **Database Choices**: For smaller applications, Amazon RDS with a MySQL or PostgreSQL engine may be more cost-effective than Aurora, especially if high availability across multiple zones is not required.

Private cloud or on-premise environment while aiming to maintain the same level of availability

Hardware Requirements

1. **Servers**: For running application instances, databases, and supporting services (e.g., load balancing, image processing). Given the need for high availability, you'd need multiple servers for redundancy.
   - **Estimate**: High-performance servers can range from $5,000 to $15,000 each, depending on specifications. Assuming a minimum of 10 servers for various roles and redundancy, the cost could range from $50,000 to $150,000.
2. **Storage**: For database storage and images uploaded by users. High-availability setups often use SAN (Storage Area Network) or NAS (Network Attached Storage).
   - **Estimate**: Prices can vary widely, but a decent setup might start around $20,000 to $50,000.
3. **Networking Equipment**: Including switches, routers, and hardware firewalls for secure, efficient network traffic management.
   - **Estimate**: $10,000 to $30,000 for a medium setup.
4. **Load Balancers**: Hardware-based solutions for distributing traffic across servers.
   - **Estimate**: $5,000 to $10,000 each, with at least 2 for redundancy, totaling $10,000 to $20,000.

Software and Licensing

1. **Operating System Licenses**: Assuming a mix of Linux (potentially free) and Windows Server (for specific workloads or management tools).
   - **Estimate**: Windows Server licenses can cost around $1,000 per server; assuming a few servers need this, allocate around $5,000 to $10,000.
2. **Database Software**: Replicating AWS RDS Aurora might involve a commercial database like Oracle or MS SQL Server, or open-source solutions with enterprise support.
   - **Estimate**: Oracle and SQL Server licenses can be very costly, easily reaching tens of thousands; an open-source solution with support might be less, so budget $10,000 to $40,000.
3. **Virtualization/Containerization Software**: For running and managing containers, you'd need a Docker Enterprise subscription or similar.
   - **Estimate**: Around $3,500 per server, totaling approximately $35,000 for 10 servers.
4. **Monitoring and Management Tools**: Similar to AWS CloudWatch or third-party solutions for infrastructure monitoring and alerting.
   - **Estimate**: $5,000 to $15,000 for a comprehensive suite.

Total Rough Estimate

Adding up the hardware and software costs, the total estimate for reproducing the "Journal" application in a private cloud/on-premise environment could range from **$133,500 to $330,000**, not including ongoing maintenance, power, cooling, and staffing costs.

Cloud Mechanism for Monitoring to Control Costs

Given the usage pattern of the "Journal" application, **AWS Lambda** (for image processing) and **AWS RDS Aurora** have significant potential to escalate costs, especially if traffic spikes or inefficient queries/load patterns are not managed. Monitoring these services closely through AWS CloudWatch to track executions, duration, and database load would be crucial in controlling costs.

Application Evolution

If development were to continue, potential features might include:

- **Enhanced Media Management**: Including video support using AWS Elastic Transcoder for video processing.
- **Machine Learning-based Insights**: Offering users insights into their journaling habits or sentiment analysis of entries, utilizing AWS SageMaker.
- **Collaboration Features**: Allowing users to collaborate on entries, which could leverage AWS AppSync for real-time data synchronization.

References:

[1] "What Is Amazon Aurora? - Amazon Aurora." *Docs.aws.amazon.com*, Available Online: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_AuroraOverview.html [Accessed April 9th 2024]

[2] "What is AWS Secrets Manager? - AWS Secrets Manager." *Docs.aws.amazon.com*, Available Online: https://docs.aws.amazon.com/secretsmanager/latest/userguide/intro.html [Accessed April 9th 2024]

[3] "AWS Pricing Calculator". *Calculator.aws,* Available Online: https://calculator.aws/#/ [Accessed April 9th 2024]

[4] "Cloud Deployment Models – GeeksForGeeks". Geeksforgeeks, Available Online: https://www.geeksforgeeks.org/cloud-deployment-models/ [Accessed April 9th 2024]