Table of Contents

[**Application Selection and Justification** 3](#_Toc184237044)

[**Cloud Architecture Design** 4](#_Toc184237045)

[1. Compute 4](#_Toc184237046)

[2. Storage 5](#_Toc184237047)

[3. Database 5](#_Toc184237048)

[4. Networking and Content Delivery 6](#_Toc184237049)

[5. Application Integration 7](#_Toc184237050)

[6. Management and Governanc 8](#_Toc184237051)

[**Adherence to Architecting Principles and Best Practices** 12](#_Toc184237052)

[**Implementation on AWS** 15](#_Toc184237053)

[1. Infrastructure Deployment 15](#_Toc184237054)

[2. Application Deployment 16](#_Toc184237055)

[3. Application Integration 17](#_Toc184237056)

[4. Monitoring and Logging 17](#_Toc184237057)

[5. Deployment Automation 17](#_Toc184237058)

[6. Testing and Validation 18](#_Toc184237059)

[**Analysis of Cost, Performance, Security, and Scalability** 19](#_Toc184237060)

[1. Cost Analysis and Management 19](#_Toc184237061)

[2. Performance Metrics and Optimization 20](#_Toc184237062)

[3. Security Measures and Compliance 20](#_Toc184237063)

[4. Scalability Plan and Future Growth 21](#_Toc184237064)

[**Evidences / Screenshots** 22](#_Toc184237065)

[**Challenges and workarounds** 32](#_Toc184237066)

[**Infrastructure as Code** 34](#_Toc184237067)

[**Conclusion** 34](#_Toc184237068)

[**References** 36](#_Toc184237069)

**Application Selection and Justification**

**Overview**

The architecture is designed to host a blogging platform on AWS, ensuring high availability, scalability, security, and cost-effectiveness. This multi-tier architecture adheres to the six pillars of the AWS Well-Architected Framework, leveraging the following layers:

* **Web Tier**: Public-facing frontend hosted in public subnets across two Availability Zones.
* **Application Tier**: Backend services hosted in private subnets for secure processing.
* **Database Tier**: A relational database hosted in private subnets for persistent storage.
* **Integration Layer**: Serverless components and messaging services for seamless interaction between layers.

Each service was chosen based on its ability to meet application requirements in terms of cost, performance, security, and scalability.

**Purpose and Scope**

* Purpose: To design and implement a cloud architecture that hosts an open-source application on AWS, ensuring high availability, security, and cost-effectiveness.
* Scope: The project encompasses selecting a suitable application, designing the architecture using at least six AWS service categories, implementing the architecture on AWS, and analyzing the solution in terms of cost, performance, security, and scalability.

**Chosen Application**

For this project, I selected my application i.e. blogging platform that allows users to create, edit, and manage blog posts. The application also supports user authentication, notifications, and the ability to interact with other users' posts. This application is built using modern technologies, including React.js for the frontend, Node.js for the backend, and MySQL for the database. The platform also has features like notifications, making it an excellent candidate for showcasing integration with AWS services.

**Reason for Selection**

1. **Learning Opportunity**: This application’s diverse functionalities provide an opportunity to work with a wide range of AWS services. For instance, user authentication and notifications require secure and scalable solutions, which can be addressed with AWS Secrets Manager, SNS, and Lambda.
2. **Complexity**: The application has a modular architecture, requiring integration across multiple layers such as the frontend, backend, database, and notifications. This complexity allows for a robust demonstration of AWS services and their orchestration.
3. **Real-World Use Case**: Blogging platforms are widely used in real-world scenarios. Deploying this application on AWS provides insights into how cloud technologies can support high availability, scalability, and security for similar production-grade applications.

# **Cloud Architecture Design**

**Overview of the Architecture**

The architecture is designed to meet the requirements of the six pillars of the AWS Well-Architected Framework: Operational Excellence, Security, Reliability, Performance Efficiency, Cost Optimization, and Sustainability. The solution utilizes services from at least six AWS service categories, ensuring a comprehensive deployment strategy.

**AWS Service Selection and Justification**

### 1. Compute

**Amazon EC2 (Elastic Compute Cloud)**:

**Usage:** EC2 instances host the frontend and backend components of the blogging platform. Separate Auto Scaling Groups (ASGs) are used for the frontend and backend tiers to ensure optimal resource allocation. The backend instances are hosted in private subnets for secure processing, while the frontend resides in public subnets for user accessibility.

**Justification:**

* + **Cost:** Allows fine-grained control over resource costs with options to right-size instances. ASGs dynamically scale instances based on traffic, avoiding over-provisioning.
  + **Performance:** Delivers consistent and high computational power for serving user requests. ASGs maintain application responsiveness during high traffic periods.
  + **Security:** Backend EC2 instances are hosted in private subnets, isolated from public exposure. Security groups restrict inbound traffic to specific sources like load balancers.
  + **Scalability:** Automatically scales horizontally across multiple Availability Zones, ensuring high availability and fault tolerance.

**AWS Lambda**:

**Usage**: Serverless Lambda functions handle event-driven tasks, such as fetching secrets securely from AWS Secrets Manager (SecretsManagerLambda) and sending notifications for new blog posts (BloggingNotificationLambda).

**Justification:**

* **Cost**: Pay-per-use pricing eliminates costs for idle resources, making it ideal for intermittent workloads like sending notifications.
* **Performance**: Scales automatically to handle multiple concurrent events without latency. Suitable for lightweight operations like querying secrets or sending SNS messages.
* **Security**: Eliminates persistent infrastructure, reducing attack surfaces. Functions operate with restricted IAM roles for secure access to other AWS services.
* **Scalability**: Serverless execution enables infinite scaling based on incoming events.

### 2. Storage

**Amazon S3 (Simple Storage Service)**

**Usage:** S3 stores static assets (e.g., images, CSS, and JavaScript files) for the blogging platform. It also acts as a centralized repository for CloudWatch logs, enabling long-term retention and compliance auditing.

**Justification:**

* **Cost**: Offers tiered pricing with lifecycle policies to transition logs and static files to cheaper storage classes like Glacier.
* **Performance**: Provides high throughput and low-latency access to assets, improving the end-user experience.
* **Security**: Ensures data integrity with server-side encryption (SSE) and enforces strict access controls via bucket policies and IAM roles.
* **Scalability**: Scales seamlessly to handle increasing data volumes, ensuring uninterrupted service even during traffic spikes.

### 3. Database

**Amazon RDS (Relational Database Service) with MySQL**

**Usage**: RDS hosts the relational database for storing user profiles, blog posts, comments, and notifications. It uses Multi-AZ deployment for fault tolerance.

**Justification**:

* **Cost**: Automated backups and managed updates reduce operational costs. Pay-as-you-go pricing for instance size and storage ensures cost efficiency.
* **Performance**: Optimized for high throughput with automated performance tuning. Multi-AZ deployments ensure low-latency failover in case of outages.
* **Security**: Data is encrypted both at rest and in transit. Security groups allow access only from backend EC2 instances.
* **Scalability**: Vertical scaling supports increased workloads, and read replicas can be added for read-heavy operations.

**AWS Secrets Manager**:

**Usage**: Stores sensitive credentials, such as database connection strings and API secrets, securely. Backend services fetch these secrets dynamically using Lambda functions.

**Justification**:

* **Cost**: Eliminates the need for manual credential management, reducing operational overhead.
* **Performance**: Secrets are accessed with minimal latency, supporting high-speed backend operations.
* **Security**: Secrets are encrypted and rotated automatically, ensuring they remain secure.
* **Scalability**: Seamlessly integrates with multiple AWS services and scales with the application.

### 4. Networking and Content Delivery

**Amazon VPC (Virtual Private Cloud)**

**Usage**: Provides an isolated network for hosting the blogging platform. Public subnets host the frontend and load balancers, while private subnets host backend EC2 instances and RDS.

**Justification**:

* **Cost**: VPC traffic is cost-effective for communication between application components.
* **Performance**: Custom route tables and NAT Gateways ensure optimized internal traffic routing.
* **Security**: Public and private subnets isolate resources for secure processing. VPC flow logs provide insights into network traffic.
* **Scalability**: Multi-AZ subnets enable redundancy and support high availability.

**Subnet Configuration**:

* **Public Subnets**: Two subnets (one in each AZ) host the Internet-facing ALB and NAT Gateways.
* **Private Subnets (Application Tier)**: Two subnets (one in each AZ) host backend EC2 instances.
* **Private Subnets (Database Tier)**: Two subnets (one in each AZ) host RDS instances.

**NAT Gateways**

**Usage**: Two NAT Gateways (one in each public subnet) allow backend EC2 instances in private subnets to securely access the internet.s

**Justification**:

* **Cost**: Provides cost-effective outbound internet access, avoiding the need for public IPs on private instances.
* **Performance**: Redundancy across AZs ensures uninterrupted internet connectivity.
* **Security**: Keeps private instances secure by restricting direct internet exposure.
* **Scalability**: Handles increasing traffic volumes from private instances efficiently**.**

**Elastic Load Balancers (ALB)**

**Usage**: An **Internet-facing** ALB manages HTTP/HTTPS traffic to frontend EC2 instances, while an **internal** ALB distributes requests to backend EC2 instances.

**Justification**:

* **Cost**: Pay-per-use pricing ensures cost efficiency for traffic distribution.
* **Performance**: Balances load evenly across instances, ensuring reliability during high traffic.
* **Security**: Supports SSL termination for secure traffic encryption and enforces access control via security groups.
* **Scalability**: Automatically adjusts to handle varying levels of incoming traffic.

**Security Groups**

* **Frontend ALB Security Group**: Allows HTTP/HTTPS traffic from the internet.
* **Backend ALB Security Group**: Allows traffic only from the frontend ALB.
* **EC2 Security Groups**: Restrict access to EC2 instances based on tier-specific rules.
* **RDS Security Group**: Restricts access to the database from the backend EC2 instances only.

### 5. Application Integration

**Amazon SNS (Simple Notification Service)**

**Usage:** Sends notifications to users for events like new blog posts or comments.

**Justification:**

* **Cost**: Pay-per-use pricing makes it cost-effective for high-volume message delivery.
* **Performance**: Delivers messages reliably with minimal latency.
* **Security**: IAM roles restrict publishing and subscription to authorized entities.
* **Scalability**: Handles high message throughput across multiple protocols

**Amazon API Gateway**

**Usage**: Exposes RESTful APIs to securely invoke Lambda functions for secrets management and notifications.

**Justification**:

* **Cost**: Pay-per-call pricing minimizes expenses for API exposure.
* **Performance**: Optimized for handling thousands of requests per second.
* **Security**: Provides authentication and authorization, ensuring only valid requests are processed.
* **Scalability**: Automatically scales to support varying API traffic volumes.

6. Management and Governanc**e**

**AWS CloudFormation**

**Usage**: Automates the deployment of the blogging platform infrastructure using YAML templates.

**Justification**:

* **Cost**: Reduces manual errors and operational overhead.
* **Performance**: Speeds up provisioning and updates to infrastructure.
* **Security**: Ensures consistent resource configuration across environments.
* **Scalability**: Supports deployment of complex, multi-region architectures.

**Architecture Diagram:**

The architecture diagram showcases the multi-tier design of the blogging platform hosted on AWS, built to ensure high availability, scalability, and security. It integrates key AWS services like EC2 for frontend and backend hosting, Amazon RDS for the database, and S3 for static assets and logs. The design leverages Application Load Balancers (ALBs) to distribute traffic, Auto Scaling Groups for dynamic resource management, and NAT Gateways for secure internet access from private subnets. AWS Lambda and SNS enable serverless operations and notifications, while Secrets Manager ensures secure credentials management. This modular and fault-tolerant architecture adheres to the AWS Well-Architected Framework, supporting efficient performance and cost optimization.

A diagram of a computer

Description automatically generated with medium confidence

Diagram : Architecture Diagram

**Data Interaction Sequence Diagram:**The sequence diagram provides a detailed flow of interactions between various components of the blogging platform, demonstrating how user requests are processed. It begins with the user sending a request (e.g., logging in or creating a blog post) through the frontend, which is routed via the Application Load Balancer (ALB) to the backend EC2 instances. The backend processes the request, retrieving necessary data securely from Amazon RDS using credentials managed by AWS Secrets Manager. For operations like sending notifications, the backend invokes AWS Lambda functions, which interact with Amazon SNS to deliver messages to subscribed users. This diagram showcases the seamless integration and communication between services, emphasizing security, scalability, and modular design.

A screenshot of a computer

Description automatically generated

Diagram : Data Interaction Sequence Diagram

# **Adherence to Architecting Principles and Best Practices**

The architecture for the blogging platform adheres to the six pillars of the AWS Well-Architected Framework, which ensures a scalable, secure, cost-efficient, and reliable deployment. Below is a detailed explanation:

**1. Operational Excellence**

* **Best Practices:**
  + Automating resource provisioning using AWS CloudFormation to reduce manual intervention.
  + Deploying Auto Scaling Groups for EC2 instances ensures seamless scaling during demand fluctuations without human oversight.
  + Centralized logging and monitoring with Amazon CloudWatch provides actionable insights into application performance and resource utilization.
* **Implementation:**
  + Logs from EC2, ALBs, and Lambda are aggregated in CloudWatch and stored in Amazon S3 for long-term retention.
  + CloudWatch alarms notify administrators of any critical issues, enabling proactive incident management.
  + Consistent resource tagging helps identify and manage resources effectively.

**2. Security**

* **Best Practices:**
  + Implementing the principle of least privilege by restricting access through IAM roles and policies.
  + Using VPC isolation to separate public-facing resources (frontend) from private resources (backend and database).
  + Encrypting data at rest and in transit to prevent unauthorized access.
* **Implementation:**
  + AWS Secrets Manager securely stores sensitive credentials like database connection strings, eliminating hard-coded secrets.
  + Security Groups:
    - Frontend ALB allows only HTTP/HTTPS traffic from the internet.
    - Backend EC2 instances and RDS are isolated within private subnets, with restricted access controlled by security groups.
  + Logs and metrics are secured with access controls, ensuring only authorized users can access operational data.

**3. Reliability**

* **Best Practices:**
  + Designing for high availability and fault tolerance using Multi-AZ deployment for critical resources.
  + Using health checks to ensure only healthy resources serve traffic.
* **Implementation:**
  + Elastic Load Balancers (ALBs):
    - Internet-facing ALB ensures frontend availability.
    - Internal ALB balances backend traffic, providing failover support.
  + Amazon RDS with Multi-AZ deployment ensures database availability during infrastructure failures.
  + Auto Scaling Groups maintain resource availability during demand spikes and hardware failures.

**4. Performance Efficiency**

* **Best Practices:**
  + Caching frequently accessed data and static content to reduce latency.
  + Monitoring performance metrics to optimize application components.
* **Implementation:**
  + Backend communication with AWS Lambda for fetching secrets and processing notifications ensures lightweight, low-latency operations.
  + Amazon CloudWatch Dashboards provide real-time insights into performance metrics such as request latency, throughput, and CPU utilization.
  + Database performance is optimized with query indexing and connection pooling.

**5. Cost Optimization**

* **Best Practices**:
  + Using right-sized resources and dynamic scaling to reduce costs.
  + Transitioning infrequently accessed data to cost-efficient storage classes.
* **Implementation:**
  + Amazon S3 Lifecycle Policies transition older logs to Glacier for cost savings.
  + Auto Scaling Groups ensure compute resources scale down during low-traffic periods.
  + AWS Lambda enables a pay-per-use model for event-driven tasks, avoiding idle server costs.
  + Amazon RDS automatically adjusts instance size and storage as needed, reducing unnecessary expenditures.

**6. Sustainability**

* **Best Practices:**
  + Leveraging managed services to minimize energy consumption and carbon footprint.
  + Scaling resources efficiently to avoid waste.
* **Implementation:**
  + Using serverless technologies like AWS Lambda eliminates the need for over-provisioning.
  + Auto Scaling ensures resources are dynamically adjusted based on demand, minimizing idle capacity.
  + Managed services like Amazon RDS and S3 reduce the overhead of managing on-premise infrastructure, lowering energy consumption.

**How the Architecture Follows AWS Best Practices**

**Modular Design:**

* The architecture is decoupled into layers (frontend, backend, database, integration) to ensure scalability and maintainability.

**Automation:**

* Infrastructure is provisioned using **CloudFormation templates**, ensuring consistent deployments across environments.

**Monitoring and Logging:**

* Centralized logging with **CloudWatch Logs** enables auditing and real-time monitoring of application behavior.

**Disaster Recovery:**

* Multi-AZ deployment for RDS and Auto Scaling across Availability Zones ensures fault tolerance and disaster recovery readiness.

# **Implementation on AWS**

The implementation of the blogging platform on AWS involves multiple stages, including infrastructure deployment, application setup, and configuration of services to ensure seamless functionality. Below are the detailed steps taken for the implementation:

### 1. Infrastructure Deployment

* **Network Configuration**:
  + **Amazon VPC**:
    - A custom VPC was created with a CIDR block of 10.0.0.0/16 to host the application resources securely.
  + **Subnets**:
    - **Public Subnets**: Two public subnets (10.0.1.0/24 in AZ-1 and 10.0.2.0/24 in AZ-2) were created for hosting the frontend EC2 instances, NAT Gateways, and the Internet-facing Application Load Balancer (ALB).
    - **Private Subnets (Application Tier)**: Two private subnets (10.0.3.0/24 in AZ-1 and 10.0.4.0/24 in AZ-2) were created for backend EC2 instances.
    - **Private Subnets (Database Tier)**: Two private subnets (10.0.5.0/24 in AZ-1 and 10.0.6.0/24 in AZ-2) were created for hosting RDS instances.
  + **Routing**:
    - A public route table was created for the public subnets, with a route to the Internet Gateway (IGW).
    - Private subnets used NAT Gateways for secure outbound internet access.
* **NAT Gateways**:
  + Two NAT Gateways (one in each public subnet) were deployed to ensure fault-tolerant internet access for private subnets.
* **Security Groups**:
  + **Frontend ALB Security Group**: Allows inbound HTTP (port 80) and HTTPS (port 443) traffic from the internet.
  + **Backend ALB Security Group**: Allows inbound traffic only from the frontend ALB.
  + **Frontend EC2 Security Group**: Allows inbound traffic from the ALB and outbound internet access via NAT Gateways.
  + **Backend EC2 Security Group**: Allows inbound traffic from the backend ALB and outbound traffic to the database.
  + **RDS Security Group**: Allows inbound traffic on port 3306 from the backend EC2 instances.

### 2. Application Deployment

* **Frontend**:
  + **EC2 Instances**:
    - Deployed in Auto Scaling Groups (ASGs) across two public subnets.
    - Instances run Docker containers hosting the React.js frontend.
    - Configured the environment variables for backend communication (ALB DNS name).
  + **Internet-facing ALB**:
    - Distributes traffic to frontend EC2 instances.
    - Configured health checks on the /health endpoint to ensure instance health.
* **Backend**:
  + **EC2 Instances**:
    - Deployed in ASGs across two private subnets.
    - Instances run Docker containers hosting the Node.js backend.
    - Communicates securely with RDS using credentials fetched from Secrets Manager.
  + **Internal ALB**:
    - Distributes traffic to backend EC2 instances.
    - Configured health checks to monitor instance availability.
* **Database**:
  + **Amazon RDS (MySQL)**:
    - Hosted in private subnets with Multi-AZ deployment for high availability.
    - Automated backups enabled for data protection.
  + **AWS Secrets Manager**:
    - Stored RDS credentials securely.
    - Configured Lambda functions to fetch credentials dynamically.
* **Storage**:
  + **Amazon S3**:
    - Used for storing static assets and CloudWatch logs.
    - Configured bucket policies to restrict access and enable server-side encryption.
  + **CloudWatch Log Exports**:
    - Set up CloudWatch to export logs (e.g., application logs, API Gateway logs) to S3 for long-term storage.

### 3. Application Integration

* **AWS Lambda**:
  + Deployed two Lambda functions:
    - SecretsManagerLambda: Fetches secrets from Secrets Manager for backend EC2 instances.
    - BloggingNotificationLambda: Sends notifications for new blog posts via Amazon SNS.
  + Configured IAM roles to grant Lambda functions access to required AWS services.
* **Amazon API Gateway**:
  + Exposed RESTful APIs to invoke Lambda functions securely.
  + Configured API Gateway to route /secrets and /notify requests to their respective Lambda functions.
* **Amazon SNS**:
  + Created an SNS topic (BloggingPlatformNotifications) for sending notifications about new blog posts.
  + Subscribed email endpoints for testing and configured Lambda to publish messages to the topic.

### 4. Monitoring and Logging

* **Amazon CloudWatch**:
  + Configured CloudWatch to monitor metrics such as CPU utilization, memory usage, and error rates.
  + Set up alarms for critical metrics to notify via SNS.
  + Aggregated logs from EC2 instances, ALBs, and Lambda into CloudWatch Logs.

### 5. Deployment Automation

* **AWS CloudFormation**:
  + Created templates for provisioning the entire infrastructure as code.
  + Deployed multiple stacks:
    - **Networking Stack**: Configured VPC, subnets, NAT Gateways, and route tables.
    - **Application Stack**: Provisioned EC2 instances, ALBs, ASGs, and RDS.
    - **Integration Stack**: Configured API Gateway, Lambda functions, and SNS.
* **Deployment Process**:
  + Deployed the backend before the frontend to ensure that API endpoints were accessible during frontend setup.
  + Configured health checks in ALBs to validate application readiness during deployment.

### 6. Testing and Validation

Comprehensive validation and testing were conducted to ensure the system's functionality and security.

For network validation, subnet routing and NAT Gateway functionality were verified to confirm internet access for private subnets, and security group rules were reviewed to ensure that only intended traffic was allowed.

For application functionality, the frontend and backend endpoints were tested to confirm proper communication, and database connectivity was validated using secrets securely retrieved from AWS Secrets Manager.

In logging and monitoring, logs were confirmed to flow into CloudWatch and be exported to S3, with performance metrics being actively monitored via CloudWatch dashboards. Lastly, notifications were tested by triggering events through Lambda functions, and email delivery to subscribed endpoints was verified for SNS notifications, ensuring reliable and timely alerts.

**Implementation Highlights**

* **Multi-AZ Redundancy**: Ensured high availability for all critical components (EC2, RDS, ALBs).
* **Centralized Logging**: Used S3 as a centralized log repository with lifecycle management for cost savings.
* **Dynamic Scaling**: Auto Scaling Groups handle varying traffic levels efficiently.
* **Serverless Integration**: Lambda functions and API Gateway simplify application integration while reducing costs.

# **Analysis of Cost, Performance, Security, and Scalability**

### 1. Cost Analysis and Management

**Cost Drivers**:

The cost of the infrastructure is influenced by several factors across different service categories. Compute resources contribute through the use of EC2 instances for the frontend and backend, as well as AWS Lambda invocations for serverless tasks. Storage costs arise from Amazon S3, which stores static assets and logs, and EBS volumes used by EC2 instances. Database expenses include the instance hours and storage for Amazon RDS. Networking costs are incurred from data transfers between services, such as EC2 and RDS, as well as from Elastic Load Balancer usage, including hours of operation and data processed. Application integration adds to the costs with API Gateway calls and SNS messages for notifications, ensuring seamless communication between components.

**Cost Optimization Strategies**:

Several strategies were implemented to optimize costs while maintaining performance and functionality. Right-sizing resources involved selecting **t3.micro** instances for EC2 where appropriate to balance cost and performance, with regular reviews of instance utilization to adjust sizes as needed. **Auto Scaling Groups** were employed to dynamically adjust the number of instances based on demand, avoiding unnecessary expenses during low-traffic periods.

Serverless computing with AWS Lambda was utilized for event-driven tasks, eliminating the need for always-on servers and reducing costs. In a production environment, Reserved Instances or Savings Plans would be considered for consistent workloads to further cut compute expenses. Monitoring tools like AWS Cost Explorer and billing alerts were used to track expenses and avoid unexpected charges. Additionally, S3 lifecycle policies were configured to transition infrequently accessed data to cheaper storage classes, ensuring efficient use of storage resources.

**Cost Management Tools**:

To effectively manage costs, several tools were utilized. AWS Cost Explorer was employed to analyze spending patterns and identify areas for optimization, enabling proactive cost management. AWS Budgets were set up to define cost and usage limits with alerts, ensuring expenses stayed within planned boundaries. Additionally, cost allocation tags were applied to resources, providing detailed reporting and insights into resource-specific expenditures, which facilitated more granular tracking and optimization.

### 2. Performance Metrics and Optimization

**Key Performance Indicators (KPIs)**:

Performance was measured using latency (time for backend request processing), throughput (requests per second), error rates (HTTP 4xx and 5xx), and resource utilization (CPU, memory, and network usage) of EC2 instances. These metrics ensured responsiveness, capacity, and efficiency.

**Performance Optimization Techniques**:

* **Auto Scaling**: Ensured that the application scales out during peak demand and scales in during low traffic to maintain performance levels.
* **Database Optimization**: Used Read Replicas in RDS for read-heavy workloads/ Optimized database queries and indexing.
* **Monitoring with CloudWatch**: Set up dashboards to visualize performance metrics in real-time.Configured alarms to alert on performance degradation.
* **Load Balancing**: Used ALBs to distribute traffic evenly across instances, preventing any single instance from becoming a bottleneck.

### 3. Security Measures and Compliance

**Network Security**:

The network infrastructure leverages Amazon VPC to segregate resources into public and private subnets, ensuring a structured and secure environment. Network ACLs (Access Control Lists) are implemented as an additional layer of security, providing fine-grained control over inbound and outbound traffic at the subnet level. Security Groups are meticulously configured with inbound and outbound rules to allow only the necessary traffic, adhering to the principle of least privilege. This approach ensures that access is restricted to what is explicitly required, minimizing the risk of unauthorized access and enhancing the overall security posture.

**Data Protection**:

To protect sensitive data, encryption at rest has been enabled for RDS databases, ensuring that stored data is securely encrypted. Additionally, S3 buckets are encrypted using SSE-S3 or SSE-KMS to safeguard object data. For encryption in transit, SSL/TLS protocols have been configured to secure data transfers between services. HTTPS listeners are implemented on Application Load Balancers (ALBs) to ensure secure communication over the network, providing robust protection for data both at rest and in transit.

**Identity and Access Management**:

AWS Secrets Manager is utilized to securely store sensitive information, eliminating the need to hardcode credentials in the codebase and reducing the risk of accidental exposure. While IAM configurations were limited within the lab environment, in a production setting, IAM roles and policies would be meticulously crafted to enforce the principle of least privilege. This ensures that users and services have access only to the resources and actions necessary for their tasks, enhancing security and reducing potential vulnerabilities.

**Compliance and Auditing**:

To enhance visibility and analysis, logs are centralized in CloudWatch Logs, allowing for efficient monitoring of system activities. Additionally, alerts are implemented for critical security events, ensuring timely detection and response to potential threats, thereby bolstering the overall security posture of the environment.

### 4. Scalability Plan and Future Growth

**Current Scalability Features**:

* **Auto Scaling Groups**: Automatically adjust the number of EC2 instances based on predefined scaling policies.
* **Decoupled Components**: Used services like SNS and Lambda to decouple components, enhancing scalability.

**Future Scalability Enhancements**:

* **Microservices Architecture**: Refactor the monolithic backend into microservices to improve scalability and maintainability.
* **Containerization and Orchestration**: Migrate to container orchestration platforms like Amazon ECS or EKS for better resource utilization and scalability.
* **Database Scaling**: Implement Read Replicas and use Amazon Aurora for automatic scaling.
* **Global Expansion**: Utilize AWS Global Accelerator and deploy resources in multiple regions to reduce latency for international users.
* **Serverless Technologies**: Transition more components to serverless architectures using AWS Lambda and AWS Fargate.
* **Advanced Caching**: Implement application-level caching using Amazon ElastiCache (Redis or Memcached).

**Scalability Planning**:

* **Load Testing**: Regularly perform load testing to identify bottlenecks and plan capacity.
* **Monitoring Trends**: Analyze usage patterns to anticipate scaling needs.
* **Continuous Integration/Continuous Deployment (CI/CD)**: Implement CI/CD pipelines using AWS CodePipeline and CodeDeploy for rapid and reliable deployments.

# **Evidences / Screenshots**

**Public URL -** [**http://frontendalb-1580561673.us-east-1.elb.amazonaws.com/**](http://frontendalb-1580561673.us-east-1.elb.amazonaws.com/)

This figure displays the secure login interface for users to access the blogging platform, accessed through the frontend load balancer DNS after the successful deployment of the application using CloudFormation.

**A screenshot of a computer

Description automatically generated**

Figure 1: Login Page

The user registration page showcases the secure signup process, accessible via the frontend DNS configured through the Application Load Balancer.

**A screenshot of a computer

Description automatically generated**

Figure 2: User Registration Backend API from Frontend EC2 Instance

Figure 3 demonstrates the successful interaction between the frontend and backend services, routed through the internal load balancer deployed via CloudFormation.

**A screenshot of a computer

Description automatically generated**

Figure 3: Backend login API successful from Frontend instance

**Instances**

EC2 instances running the frontend, backend of the blogging platform, deployed in public and private subnets for security and scalability.

**A screenshot of a computer

Description automatically generated**

Figure 4: List of Instances

**CloudFormation Stacks**

CloudFormation stacks are used to provision and automate the deployment of all infrastructure components for the blogging platform.

**A screenshot of a computer

Description automatically generated**

Figure 5: Cloud Formation Stacks Created Successfully

**VPC**

Virtual Private Cloud (VPC) is configured to isolate and securely host the application’s resources, including public and private subnets.

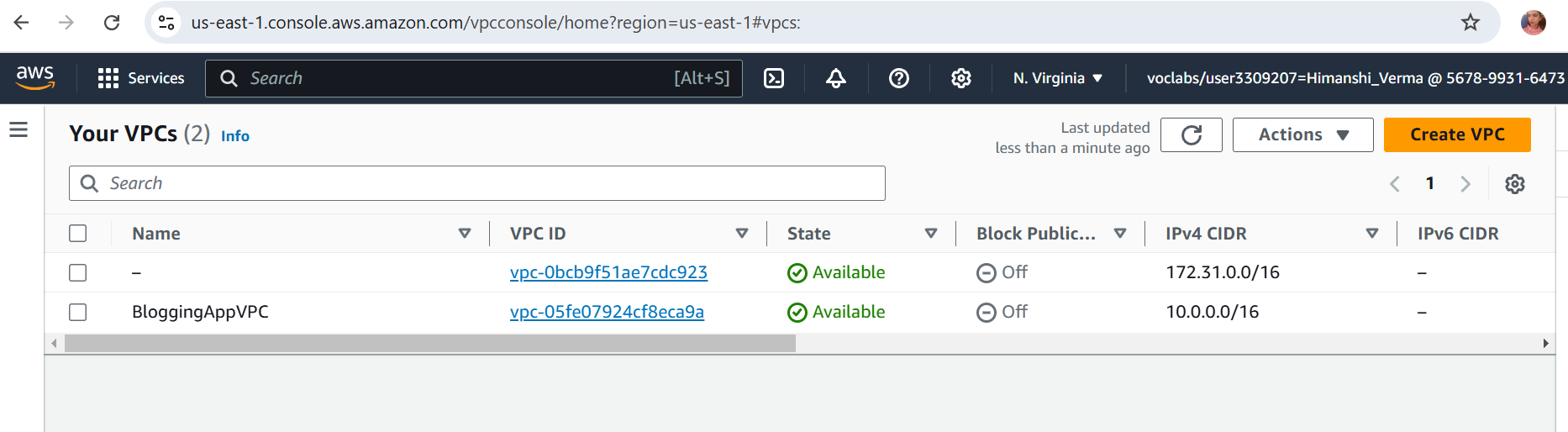
****

Figure 6: BloggingAppVPC Created Successfully

**Elastic IPs**

Elastic IPs assigned to frontend-facing resources, ensuring consistent access to the blogging platform through a stable public interface. **A screenshot of a computer

Description automatically generated**

Figure 7: Elastic IPs

**VPC Route map**

These figures detail the routing setup within the VPC, enabling secure and efficient traffic flow between subnets, NAT gateways, and the internet.

**A screenshot of a computer

Description automatically generated**

Figure 8: VPC Route Map \_1

**A screenshot of a computer

Description automatically generated**

Figure 9: VPC Route Map \_2

**Autoscaling Groups**

Auto Scaling Groups configured for the blogging platform to ensure automatic scaling of EC2 instances based on traffic demands. **A screenshot of a computer

Description automatically generated**

Figure 10: Frontend and Backend Autoscaling Groups

**Internet Gateway**

The internet gateway that facilitates internet access for public-facing resources like the frontend load balancer was created successfully through cloud formation.

**A screenshot of a computer

Description automatically generated**

Figure 11: BloggingAPPIGW Internet Gateway

**NAT Gateways**

NAT gateways providing secure outbound internet access for backend resources in private subnets.

**A screenshot of a computer

Description automatically generated**

Figure 12: NAT Gateways Deployed in 2 Availability Zones

**Route Tables**

Route tables are configured to manage traffic between subnets, NAT gateways, and external networks for the blogging platform

**A screenshot of a computer

Description automatically generated**

Figure 13: Route Tables for Database, Public and Private Routes

**Load Balancers**

Application Load Balancers are configured to distribute HTTP/HTTPS traffic across EC2 instances for frontend and backend services.

**A screenshot of a computer

Description automatically generated**

Figure 14: Load Balancers for Frontend and Backend Instances

**Target groups**

The figure below lists the target groups associated with load balancers, ensuring requests are routed to the appropriate backend services of the blogging platform.

**A screenshot of a computer

Description automatically generated**

Figure 15: Target Groups for Frontend and Backend

**Security groups**

Security groups restricting and controlling inbound and outbound traffic for the application’s EC2 instances, load balancers, and database.

**A screenshot of a computer

Description automatically generated**

Figure 16: Security Groups

**Lambdas**

AWS Lambda functions used for serverless tasks like fetching secrets and sending notifications for new blog posts.

**A screenshot of a computer

Description automatically generated**

Figure 17: Lambdas for Sending Notification and Fetching Secrets

**Secrets Manager**

These figures(18 and 19) illustrate the use of AWS Secrets Manager to securely store and manage sensitive credentials used by backend components.

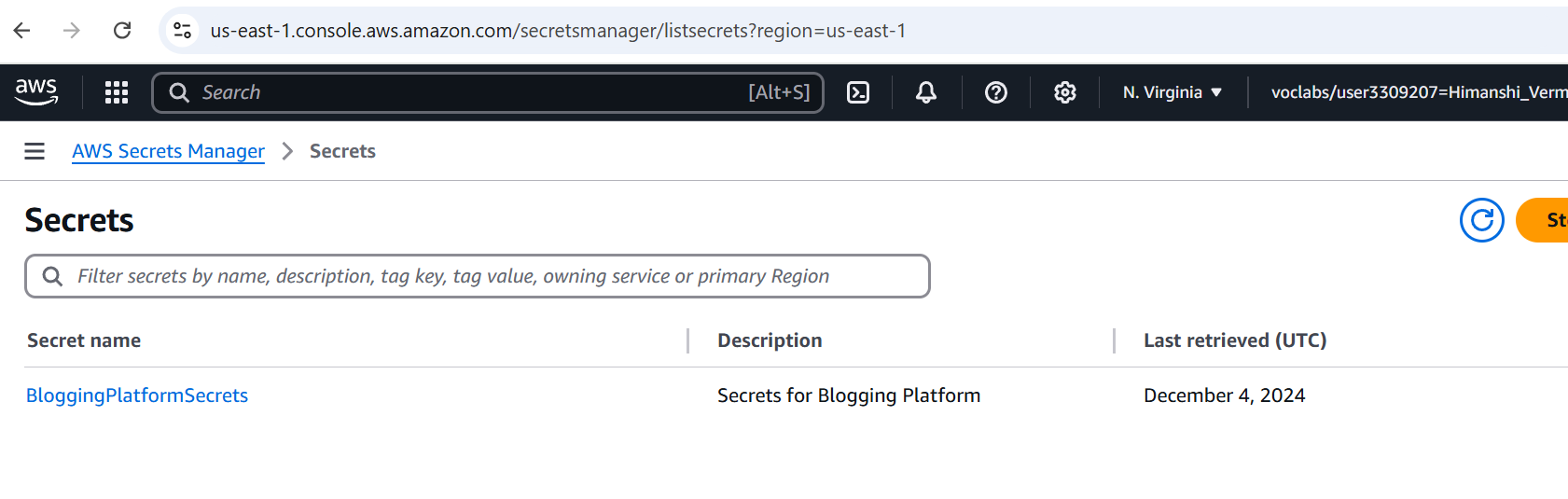
****

Figure 18: Secrets Manager

**Secrets**

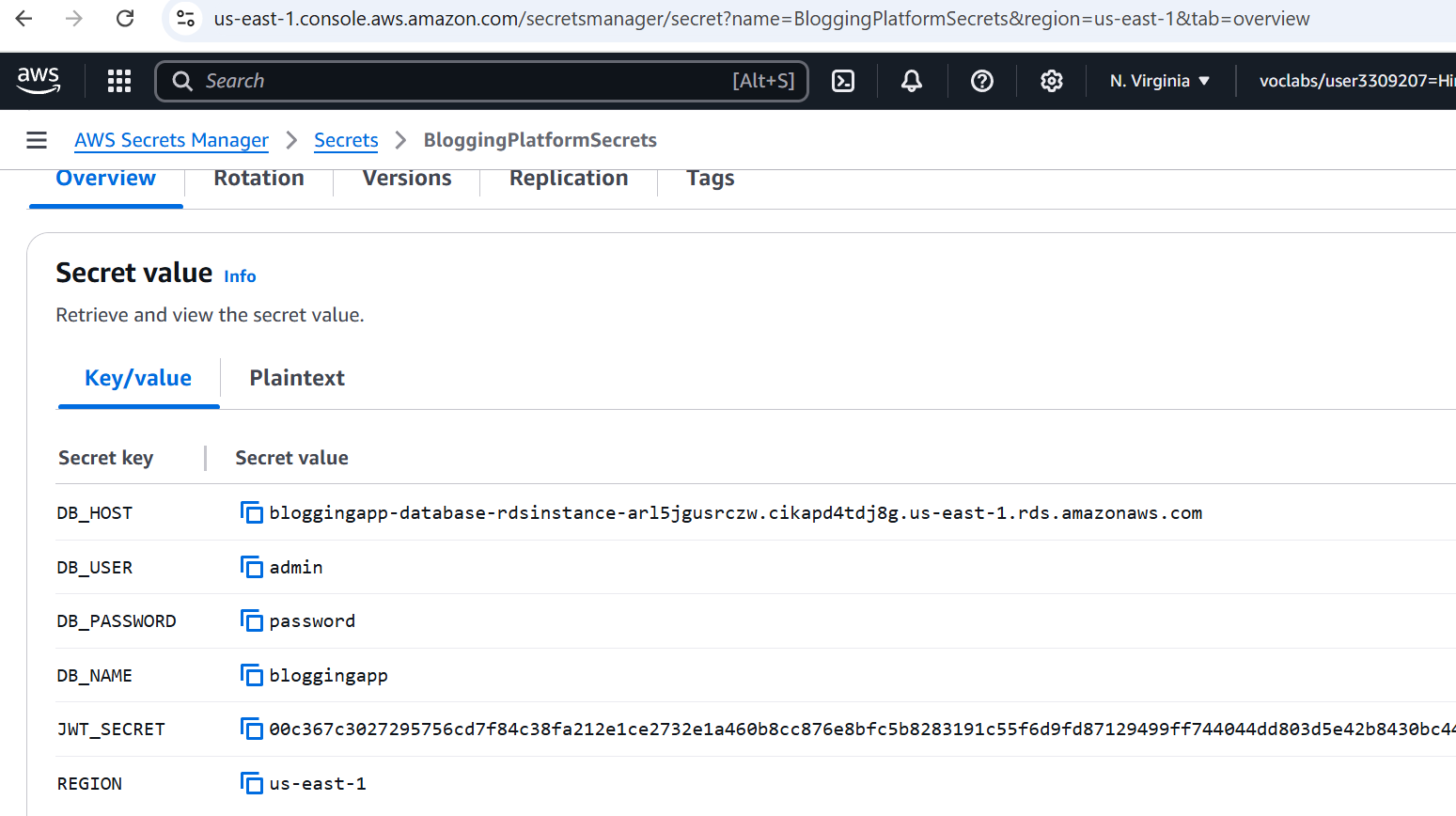
****

Figure 19: Secrets

# **Challenges and workarounds**

One of the key challenges encountered during the project was enabling communication between the frontend UI and the backend services, as the backend was hosted entirely in private subnets. This setup, while enhancing security, prevented direct requests from the frontend UI (technically the client’s browser or internet) to the backend ALB. Several options were considered to address this issue, each evaluated for feasibility, impact on security, performance, and overall architecture.

**Explored Options**

1. **Reverse Proxy with NGINX:**

**Reasoning:** Acts as a middle layer between the frontend and the backend, routing requests securely without exposing backend services directly.

**Why Chosen:** Maintains the backend's private nature while enabling controlled access.It is Lightweight and easily configurable, fitting well into the existing architecture and is also Cost-effective compared to alternative options like switching to EBS.

1. **Using a Firewall with Specific Rules:**

**Reasoning:** Configure firewall rules to allow specific IPs or services to access the backend ALB directly.

**Why Not Chosen:** Introducing specific access rules to private subnets increases the risk of misconfigurations, potentially compromising security. Also it is complex to maintain as the application scales.

1. **Switching to Elastic Beanstalk (EBS):**

**Reasoning:** Elastic Beanstalk simplifies deployment and manages both frontend and backend services while handling traffic routing internally.

**Why Not Chosen:** Significant architectural change, requiring redeployment and potential reconfiguration of services. The overhead of transitioning to EBS outweighed the benefits, as the focus was on maintaining the current VPC-based setup.

1. **Making the Load Balancer Public:**

**Reasoning:** Allow frontend UI to directly access the backend ALB over the internet.

**Why Not Chosen:** Contradicts the principle of isolating backend services in private subnets and increases the attack surface and undermines security best practices.

**Final Solution: NGINX Reverse Proxy**

The NGINX reverse proxy was selected as the most suitable workaround. It allowed frontend requests to be securely routed to the backend ALB while keeping the backend private. The NGINX server was configured within the VPC to ensure that it adhered to the security model, maintaining robust access controls and a seamless user experience.

This solution effectively resolved the challenge without compromising the core principles of the architecture, such as security and reliability, and ensured compatibility with the AWS Well-Architected Framework.  
  
**Another Challenge: Restricted Use of IAM in AWS Academy Learner Lab**

Due to the limitations imposed by the AWS Academy Learner Lab environment, the use of Identity and Access Management (IAM) was restricted. This posed challenges in implementing the **Security** pillar of the AWS Well-Architected Framework, as IAM plays a critical role in defining granular access controls, enforcing the principle of least privilege, and managing roles and policies for secure interactions between AWS services.

**Proposed Approach to Handle IAM in Real-World Architecture**

If unrestricted IAM access were available, the architecture would include the following IAM implementations:

1. **Role-Based Access Control (RBAC):** Define specific IAM roles for EC2 instances, Lambda functions, and other AWS services. Example: The backend EC2 instances would be assigned a role granting access to Amazon RDS and Secrets Manager only.
2. **Least Privilege Principle:** Configure IAM policies to allow only the minimum permissions required for each service or user. Example: Lambda functions fetching secrets would have policies restricted to read-only access for Secrets Manager.
3. **Secure Credential Management:** Use IAM roles instead of hardcoded credentials to allow services like EC2 or Lambda to securely interact with other AWS resources.
4. **Fine-Grained Access Control:** Apply resource-level policies to secure specific buckets in S3 or restrict database access to certain VPCs. Example: Attach an IAM policy to S3 buckets allowing access only from designated roles or services.
5. **IAM Policy Validation:** Use AWS IAM Access Analyzer to ensure policies are not overly permissive, minimizing potential security risks.
6. **Auditing and Monitoring:** Enable AWS CloudTrail to log IAM changes and monitor access patterns for compliance and threat detection.

**Workarounds Used Without IAM**

To adapt to the lab environment restrictions:

* **Preconfigured Security Groups and Networking Controls** were used to isolate and protect resources.
* Secrets and configurations were embedded securely within application code or environment variables.
* **NAT Gateways** and private subnets ensured that backend services could securely access external resources without exposing them to the internet.

# **Infrastructure as Code**

The **Infrastructure as Code (IaC)** for the blogging platform was implemented using **AWS CloudFormation**, enabling automated and consistent deployment of resources. The deployment process involved several YAML templates, organized into logical stacks:

1. **Networking Stack**: Configured the VPC, public and private subnets, NAT gateways, route tables, and security groups.
2. **Application Stack**: Provisioned EC2 instances, Auto Scaling Groups (ASGs), and Application Load Balancers (ALBs) for the frontend and backend tiers.
3. **Database Stack**: Deployed Amazon RDS (MySQL) along with private subnets and security groups for secure database access.
4. **Integration Stack**: Configured serverless components, including AWS Lambda, API Gateway, and Amazon SNS for seamless application integration.

To deploy these stacks in the correct order, a **custom script file** was used. The script ensured that the outputs of one stack were passed to subsequent stacks using **ImportValues**. For example, subnet and security group IDs from the Networking Stack were used to deploy compute resources in the Application Stack, and the RDS endpoint from the Database Stack was used by backend services.

Additionally, the YAML code included comprehensive **CloudWatch** configurations. CloudWatch was configured to track key metrics such as EC2 CPU utilization, ALB latency, and RDS performance. Alarms were defined in the YAML templates to send SNS notifications for threshold breaches, ensuring proactive issue resolution.

# **Conclusion**

The project successfully designed and implemented a robust cloud architecture on AWS to host a complex open-source blogging application. Leveraging a diverse range of AWS services across six categories, the architecture aligns with the AWS Well-Architected Framework principles, ensuring high availability, security, and scalability. Thoughtful resource selection and architecture design were key to achieving these objectives while implementing cost optimization strategies to manage and reduce operational expenses. The architecture establishes a strong foundation for future enhancements and scalability, accommodating the platform's growth and evolving requirements.

**Future Improvements and Recommendations**

* **Enhanced Security:** Introduce AWS WAF for protection against web exploits and AWS Shield for DDoS mitigation.
* **CI/CD Automation:** Streamline build, test, and deployment processes with AWS CodePipeline and CodeBuild.
* **Advanced Monitoring:** Leverage AWS X-Ray for request tracing and Amazon QuickSight for data visualization.
* **Cost and Compliance Optimization:** Conduct AWS Trusted Advisor reviews for cost-saving recommendations. Use AWS Config to track resource compliance and AWS Organizations for account management.
* **User Experience:** Enhance personalization with Amazon Personalize for tailored content suggestions.

# **References**

1. J. Dahjohnson, "Deploy 3-Tier Architecture YAML," GitHub, [Online]. Available: <https://github.com/dahjohnson/IaC/blob/master/aws-cloudformation/deploy-3tier-architecture.yml>. [Accessed: Dec. 4, 2024].
2. A. Bach, "Creating a Highly Available 3-Tier Architecture for Web Applications in AWS," Medium, [Online]. Available: <https://medium.com/@abach06/creating-a-highly-available-3-tier-architecture-for-web-applications-in-aws-23f37d49bd25>. [Accessed: Dec. 4, 2024].
3. AWS Managed Services, "AWS CloudFormation 3-Tier Architecture Example," AWS Documentation, [Online]. Available: <https://docs.aws.amazon.com/managedservices/latest/appguide/cfn-ingest-ex-3-tier.html>. [Accessed: Dec. 4, 2024].
4. K. Matt, "Three-Tier Architecture CloudFormation Template," GitHub, [Online]. Available: <https://github.com/KMatt11/ThreeTierArchitecture-CloudFormationTemplate>. [Accessed: Dec. 4, 2024].
5. AWS, "Building Three-Tier Architectures with Security Groups," AWS Blog, [Online]. Available: <https://aws.amazon.com/blogs/aws/building-three-tier-architectures-with-security-groups/>. [Accessed: Dec. 4, 2024].
6. W. Bolton, "Creating a Three-Tier Architecture in AWS," Medium, [Online]. Available: <https://medium.com/@boltonwill/creating-a-three-tier-architecture-in-aws-777c927c4beb>. [Accessed: Dec. 4, 2024].
7. PlantText, "Create Diagrams with PlantUML," [Online]. Available: [https://planttext.com/#](https://planttext.com/). [Accessed: Dec. 4, 2024].
8. Diagrams.net, "Diagramming Tool for Cloud Architecture," [Online]. Available: <https://app.diagrams.net/>. [Accessed: Dec. 4, 2024].