Cloud Engineering Fundamentals Lab

Well-Architected and Cloud Adoption Frameworks

Prepared by Lennox Owusu Afriyie

Task 1- Reviewing the Existing Architecture

Workload Overview

The organization runs a two-tier web application on-premises. The frontend tier serves web traffic directly to users, and the backend tier handles data storage and retrieval. The plan is to migrate this workload to AWS without redesigning it first, a straightforward lift-and-shift.

Below is the assumed baseline architecture based on a typical on-premises two-tier setup:

Current Architecture Components

Frontend Tier

- A single physical or virtual server running a web server (Apache or Nginx)
- Hosts the application code and serves HTTP/HTTPS requests directly to users
- No load balancing; all traffic hits one machine
- Static assets (images, CSS, JavaScript) served from the same server

Backend Tier

- A single database server running MySQL or PostgreSQL
- Sits on the same local network as the web server
- No read replicas, no clustering, no automated failover
- Database backups are manual and infrequent, stored locally on the same machine

Networking

- Both servers sit on a flat on-premises network with no subnet segmentation
- The web server is directly internet-facing
- The database server is reachable from the web server with no firewall rules restricting which ports or IPs can connect
- No intrusion detection or traffic inspection in place

Security

- No centralized identity management; server access is managed through shared local user accounts and SSH keys stored informally
- No encryption enforced for data in transit between the web server and database

- No encryption at rest for database files
- No audit logging of who accessed what or when

Monitoring and Operations

- No centralized monitoring; the team checks server health manually or reacts to user complaints
- No alerting configured for CPU, memory, disk, or application errors
- No documented runbooks or incident response procedures
- Deployments are done manually by SSHing into the server and pulling code from a repository

Backup and Recovery

- No automated backup schedule
- The only backups that exist are occasional manual snapshots taken before major changes
- No tested recovery procedure; recovery time in the event of failure is unknown
- No offsite or redundant storage for backups

Identified Risks and Weaknesses

1. Single point of failure across both tiers
Both the web server and the database run as single instances with no redundancy. If either goes down, the entire application is unavailable. There is no failover mechanism and no secondary instance to take over.

2. No geographic or zone redundancy
The entire workload runs in one physical location. A power outage, hardware failure, or network disruption at that location takes down the application completely.

3. Overly permissive network access
The database server has no strict firewall rules limiting which machines can connect to it or on which ports. In an AWS context, this would translate to security groups open to 0.0.0.0/0, meaning any IP address on the internet could attempt a connection.
4. No backup or disaster recovery strategy

Manual, infrequent backups stored on the same machine as the data they are backing up provide almost no protection. A disk failure or ransomware attack would destroy both the live data and the backup simultaneously.

5. No encryption in transit or at rest

Traffic between the web server and database travels unencrypted over the internal network. Database files are not encrypted on disk. If an attacker gains network access, they can read data in transit. If they gain physical or OS-level access, they can read data at rest.

6. Shared and unmanaged credentials U

sing shared local accounts and informally stored SSH keys makes it impossible to audit who did what, revoke access for a specific individual without affecting others, or enforce least-privilege access.

7. No monitoring or alerting

The team has no visibility into application performance, error rates, or infrastructure health unless a user reports a problem. There is no way to detect a slow memory leak, a filling disk, or a failed background job before it causes an outage.

8. Manual deployment process

Deploying code by SSHing into a server introduces human error, leaves no deployment audit trail, and makes rollbacks difficult. There is no staging environment to test changes before they reach production.

9. Static assets served from the application server

Serving images, CSS, and JavaScript from the same server that handles application logic wastes compute resources and increases response latency for users who are geographically distant from the server.

10. No scalability mechanism

The architecture cannot handle traffic spikes. If user load doubles, the single web server either degrades or crashes. There is no way to add capacity automatically or quickly.

Task 2 – Evaluating the Workload Using the AWS Well-Architected Framework

Assessment Table

Pillar	Observation	Improvement Recommendation	Supporting AWS Service
Operational Excellence	The team deploys manually via SSH with no pipeline, no infrastructure-as-code, and no centralized logging, making operations reactive and error-prone	Automate infrastructure provisioning with code and establish a CI/CD pipeline so deployments are repeatable, auditable, and reversible	AWS CloudFormation, AWS CodePipeline, Amazon CloudWatch
Security	Security groups are open to 0.0.0.0/0, credentials are shared and unmanaged, and no encryption is enforced in transit or at rest	Restrict security group rules to minimum required ports and sources, enforce IAM roles with least-privilege policies, and enable encryption on all data paths	AWS IAM, AWS WAF, AWS KMS, AWS Certificate Manager
Reliability	The entire workload runs as single instances in one location with no automated failover, no backups, and no tested recovery procedure	Deploy across multiple Availability Zones with automated failover for both the web and database tiers, and implement automated backups with tested restore procedures	Amazon RDS Multi-AZ, AWS Auto Scaling, AWS Backup
Performance Efficiency	All read traffic hits the database directly, static assets are served from the application server, and there is no mechanism to scale compute in response to load	Add a caching layer to absorb repetitive read queries, offload static assets to object storage with a CDN, and use auto scaling to match compute capacity to actual demand	Amazon ElastiCache, Amazon S3, Amazon CloudFront, AWS Auto Scaling
Cost Optimization	EC2-equivalent compute runs at fixed capacity around the clock regardless of actual traffic patterns, and there is no visibility into what the workload costs to run	Right-size instances based on observed utilization, use auto scaling to avoid paying for idle capacity, and enable cost monitoring with budget alerts	AWS Cost Explorer, AWS Budgets, AWS Auto Scaling, AWS Compute Optimizer

Detailed Pillar Analysis

Pillar 1 – Operational Excellence

Current State

The team manages the application entirely through manual intervention. Code deployments happen over SSH, infrastructure changes are applied by hand, and there is no version-controlled record of what configuration the servers are running. When something breaks, the team has no centralized logs to consult and no runbook to follow. This means every incident response starts from scratch.

Strength

The application is functional and the team has direct familiarity with the codebase and server configuration. That institutional knowledge exists, even if it is not documented.

Weakness

There is no infrastructure-as-code, no deployment pipeline, and no centralized observability. Manual processes cannot scale, introduce inconsistency between environments, and leave no audit trail. A misconfigured deployment has no automated rollback path.

Improvement

Define all infrastructure in AWS CloudFormation templates so the environment can be reproduced exactly, version-controlled, and peer-reviewed before any change is applied. Build a CI/CD pipeline using AWS CodePipeline and AWS CodeDeploy so every code change goes through automated testing and a controlled deployment process. Send all application and infrastructure logs to Amazon CloudWatch Logs with metric filters and alarms configured for error rates, latency thresholds, and resource utilization.

Supporting AWS Services

- **AWS CloudFormation** – defines and provisions infrastructure as code
- **AWS CodePipeline** – orchestrates the build, test, and deploy stages
- **AWS CodeDeploy** – automates application deployments to EC2 instances
- **Amazon CloudWatch** – collects logs, metrics, and triggers alarms

Pillar 2 – Security

Current State

The network has no segmentation. The database is reachable from anywhere the web server can reach, and in an AWS migration done without redesign, the security groups would likely be left open to 0.0.0.0/0 for convenience. Credentials are shared local accounts with SSH keys stored informally. No encryption is applied to data moving between the web server and database, and database files are not encrypted on disk. There is no audit trail of access or changes.

Strength

The web server and database are on separate machines, so there is at least a logical separation between the presentation layer and the data layer. This separation can be formalized properly in AWS.

Weakness

Open security groups, unmanaged credentials, no encryption in transit or at rest, and no audit logging collectively mean that a single point of network access gives an attacker broad reach across the entire workload.

Improvement

Place the database in a private subnet with no route to the internet. Restrict the database security group to accept connections only from the web tier security group on the specific database port. Replace shared SSH keys with AWS Systems Manager Session Manager so no inbound SSH port needs to be open at all. Assign IAM roles to EC2 instances so the application authenticates to AWS services without storing credentials in code or config files. Enable AWS KMS encryption for the RDS database and any S3 buckets. Use AWS Certificate Manager to enforce TLS on all traffic entering the load balancer. Enable AWS CloudTrail across all regions to log every API call made in the account.

Supporting AWS Services

- **AWS IAM** – manages roles and least-privilege access policies
- **AWS KMS** – handles encryption key management for data at rest
- **AWS Certificate Manager** – provisions and manages TLS certificates
- **AWS WAF** – filters malicious HTTP/HTTPS traffic at the load balancer
- **AWS CloudTrail** – logs all API activity across the AWS account

- **AWS Systems Manager Session Manager** – provides shell access without opening SSH ports

Pillar 3 – Reliability

Current State

Both the web server and the database are single instances with no redundancy. There is no automated failover, no health check mechanism, and no backup schedule. If the web server crashes, the application goes down. If the database crashes or its disk fills up, data may be lost and recovery time is undefined because no restore procedure has ever been tested.

Strength

The two-tier separation means a frontend issue does not necessarily corrupt the database, and vice versa. The tiers can be made redundant independently.

Weakness

Single-AZ, single-instance deployment with no backups and no tested recovery procedure means the workload has no defined recovery time objective (RTO) or recovery point objective (RPO). Any failure is a full outage of unknown duration.

Improvement

Deploy the web tier as an Auto Scaling Group across at least two Availability Zones, sitting behind an Application Load Balancer. The ALB performs health checks and stops routing traffic to any instance that fails. Replace the self-managed database with Amazon RDS in Multi-AZ configuration so AWS automatically maintains a synchronous standby replica in a second AZ and fails over to it within minutes if the primary instance becomes unavailable. Enable automated RDS snapshots with a retention period that matches the organization's RPO requirement. Use AWS Backup to manage backup policies centrally across RDS and any other resources. Test the restore procedure on a schedule, not just before a crisis.

Supporting AWS Services

- **Amazon RDS Multi-AZ** – maintains a synchronous standby and handles automatic failover
- **AWS Auto Scaling** – replaces unhealthy EC2 instances and adjusts capacity
- **Application Load Balancer** – distributes traffic and performs instance health checks
- **AWS Backup** – centralizes and automates backup policies with restore testing

Pillar 4 – Performance Efficiency

Current State

Every user request that requires data goes directly to the single database instance. There is no caching layer to serve repeated queries from memory. Static files like images, CSS, and JavaScript are served from the same EC2 instance handling application logic, consuming compute and network resources that could be used for dynamic requests. The instance size was chosen at setup and has never been reviewed against actual utilization data.

Strength

A two-tier architecture is straightforward to reason about and profile. The separation between web and database tiers means each can be optimized independently without restructuring the whole application.

Weakness

No caching, no CDN, and no utilization-based right-sizing mean the application is slower and more expensive than it needs to be. The database becomes a bottleneck as read traffic grows.

Improvement

Deploy Amazon ElastiCache (Redis or Memcached) between the application tier and the database to cache the results of frequent read queries. Move all static assets to Amazon S3 and serve them through Amazon CloudFront so users receive static content from an edge location geographically close to them rather than from a single origin server. Use AWS Compute Optimizer to analyze EC2 instance utilization and recommend the right instance type and size for the actual workload. Configure Auto Scaling policies based on CPU utilization or request count so the web tier scales out during traffic spikes and scales in during quiet periods.

Supporting AWS Services

- **Amazon ElastiCache** – in-memory caching for database read offloading
- **Amazon S3** – object storage for static assets
- **Amazon CloudFront** – CDN that serves content from edge locations
- **AWS Compute Optimizer** – analyzes utilization and recommends right-sized instances
- **AWS Auto Scaling** – adjusts EC2 capacity based on real-time demand metrics

Pillar 5 – Cost Optimization

Current State

On-premises, the organization pays for hardware regardless of whether it is being used. In a direct lift-and-shift to EC2 without any changes, the same pattern continues: fixed-size instances running at full cost around the clock even during off-peak hours when utilization may be very low. There is no cost monitoring, no budget alerting, and no process for reviewing whether the resources provisioned still match what the workload actually needs.

Strength

Moving to AWS at all shifts the cost model from large upfront capital expenditure on hardware to operational expenditure that can be adjusted. That flexibility exists even if it is not yet being used.

Weakness

Without auto scaling, right-sizing, reserved capacity planning, or cost visibility tooling, the organization will likely overprovision to feel safe and have no mechanism to detect or correct that overprovisioning over time.

Improvement

Use AWS Cost Explorer to get a clear picture of where money is being spent from day one of the migration. Set up AWS Budgets with alerts so the team is notified before spending exceeds a defined threshold. Run AWS Compute Optimizer after the workload has been live for at least two weeks to get utilization-based instance size recommendations. For workloads with predictable baseline traffic, purchase Reserved Instances or Savings Plans to reduce the hourly cost of that baseline capacity by up to 72% compared to On-Demand pricing. Use Auto Scaling so the fleet only runs additional instances when traffic actually requires them.

Supporting AWS Services

- **AWS Cost Explorer** – visualizes spending patterns and forecasts costs
- **AWS Budgets** – sets spending thresholds and sends alerts when they are approached or exceeded
- **AWS Compute Optimizer** – recommends right-sized instance types based on utilization data
- **AWS Savings Plans / Reserved Instances** – reduces cost for predictable baseline workloads
- **AWS Auto Scaling** – eliminates idle capacity during low-traffic periods

Task 3 – AWS Cloud Adoption Framework (CAF) Readiness Assessment

Overview

The six CAF perspectives cover the full range of organizational capabilities needed for a successful cloud migration. The assessment below reflects the state of an organization running the two-tier application described in Task 1, preparing to migrate to AWS for the first time.

1. Business Perspective

The organization has not yet formally documented a business case for the migration. While management has approved the move to AWS, there are no defined metrics for measuring whether the migration delivers value, no cost-benefit analysis comparing on-premises running costs against projected AWS spend, and no alignment between the migration timeline and broader business objectives such as reducing infrastructure maintenance overhead or improving application availability for customers.

To move forward effectively, the organization needs to produce a migration business case that quantifies current on-premises costs (hardware, maintenance, staffing, downtime costs) against projected AWS costs. Leadership should agree on measurable outcomes, for example a target uptime percentage or a cost reduction figure, so the migration has clear success criteria. Stakeholder buy-in beyond the IT department is also needed, particularly from finance and operations, since cloud spending operates on a consumption model that requires different budget planning than capital hardware purchases.

2. People Perspective

The current IT team has strong familiarity with on-premises server administration but limited AWS experience. Deployments are handled manually, infrastructure is managed without code, and the team has no established practice around cloud-native tooling such as IAM, CloudFormation, or managed services like RDS and ElastiCache.

Before and during the migration, the organization should identify skill gaps across the team and map them to specific AWS training paths. AWS offers structured learning through AWS Skill Builder, and certifications such as AWS Cloud Practitioner and AWS Solutions Architect Associate give the team a common technical baseline. Beyond training, the organization should define new roles or responsibilities that reflect cloud operations, since managing

EC2 instances and RDS differs meaningfully from managing physical servers. A cloud center of excellence, even if it starts as one or two people, helps establish standards and share knowledge as the team's AWS experience grows.

3. Governance Perspective

The organization currently has no cloud governance policies in place. There are no tagging standards for AWS resources, no defined process for requesting or approving new cloud infrastructure, no cost allocation framework, and no compliance controls mapped to the AWS environment. Access to the AWS account is not yet governed by a formal policy.

The organization needs to establish a governance baseline before the migration goes live. This includes defining a resource tagging policy so every AWS resource is labeled by environment, owner, and cost center, enabling accurate cost tracking and accountability. AWS Organizations should be used to separate production, staging, and development environments into distinct accounts with appropriate permission boundaries. AWS Config can enforce compliance rules automatically, flagging resources that drift from defined standards. Budget controls through AWS Budgets should be set from day one so spending is visible and bounded. A change management process for cloud infrastructure, even a lightweight one, prevents unreviewed changes from reaching production.

4. Platform Perspective

The existing application was built to run on physical servers and has not been assessed for cloud readiness. The web server and database are tightly coupled to specific OS configurations, and there is no infrastructure-as-code describing the environment. Static assets are bundled with the application, and the database connection strings are likely hardcoded in configuration files rather than managed through a secrets service.

The platform readiness work involves several concrete steps. The application configuration should be externalized using AWS Systems Manager Parameter Store or AWS Secrets Manager so database credentials and environment-specific settings are not hardcoded. Infrastructure should be defined in AWS CloudFormation so the environment is reproducible. The database should be migrated from a self-managed EC2 instance to Amazon RDS, which handles patching, backups, and failover automatically. Static assets should be separated from the application and moved to S3. These changes do not require rewriting the application but do require careful planning and testing before cutover.

5. Security Perspective

The organization's current security posture is weak. There are no formal access control policies, no encryption standards, no audit logging, and no process for reviewing or responding to security events. The on-premises network relies on physical security and informal practices rather than documented controls.

Migrating to AWS creates an opportunity to establish a proper security baseline. The AWS shared responsibility model means AWS secures the underlying infrastructure, but the organization is responsible for everything above it, including IAM policies, network configuration, data encryption, and application-level security. The immediate actions needed are: enable AWS CloudTrail in all regions to log every API call, enable Amazon GuardDuty for threat detection, enforce MFA on all IAM users especially the root account, apply least-privilege IAM policies, encrypt all data at rest using AWS KMS, and enforce TLS for all data in transit using AWS Certificate Manager. A security review should be conducted before the application goes live in AWS, not after.

6. Operations Perspective

The organization currently has no formal operations practice for the application. There are no runbooks, no monitoring dashboards, no alerting thresholds, no on-call procedures, and no defined SLAs for availability or response time. Incidents are handled reactively when users report problems.

Cloud operations require a more structured approach. Before go-live, the team should define what normal looks like for the application by establishing baseline metrics for CPU utilization, request latency, error rates, and database connections. Amazon CloudWatch should be configured with alarms that notify the team when any metric crosses a defined threshold. AWS Systems Manager can automate routine operational tasks such as patch management and compliance checks. Runbooks for common failure scenarios, such as a failed deployment, a database failover, or a spike in error rates, should be written and tested before they are needed. The organization should also define an RTO and RPO for the application so recovery procedures are designed to meet a specific target rather than improvised during an outage.

Task 4 – Improved Architecture Design

Design Principles Applied

The revised architecture addresses every weakness identified in Task 1 and maps directly to all five WAF pillars. The design assumes the organization is migrating to AWS for the first time and needs a production-ready environment from day one.

Architecture Overview

The improved architecture separates the workload into distinct layers, each placed in the appropriate network zone, with redundancy built into every tier. The environment runs inside a custom VPC spanning two Availability Zones.

Layer-by-Layer Description

1. DNS and Content Delivery Layer

Amazon Route 53 handles DNS resolution for the application domain. It routes user requests to the Application Load Balancer and can be configured with health checks so traffic is only directed to a healthy endpoint.

Amazon CloudFront sits in front of the Application Load Balancer as a CDN. It caches static assets at AWS edge locations globally, reducing latency for users regardless of their geographic location and offloading static file requests from the origin servers entirely.

Amazon S3 stores all static assets including images, CSS files, and JavaScript bundles. CloudFront pulls from this S3 bucket as its origin for static content, completely removing the responsibility of serving those files from the EC2 instances.

2. Network Layer – VPC Design

A custom Amazon VPC is created with the following subnet structure across two Availability Zones:

```
VPC: 10.0.0.0/16

│ ├── Public Subnet AZ-1 (10.0.1.0/24)  ← Load Balancer, NAT Gateway

 ├── Public Subnet AZ-2 (10.0.2.0/24)  ← Load Balancer, NAT Gateway

 ├── Private Subnet AZ-1 (10.0.3.0/24) ← EC2 Web/App Tier

 ├── Private Subnet AZ-2 (10.0.4.0/24) ← EC2 Web/App Tier
```

```
│ ├── Private Subnet AZ-1 (10.0.5.0/24) ← RDS Primary
└── Private Subnet AZ-2 (10.0.6.0/24) ← RDS Standby (Multi-AZ)
```

Internet Gateway is attached to the VPC and associated only with the public subnets, giving the load balancer a path to the internet.

NAT Gateways are deployed in each public subnet so EC2 instances in the private subnets can initiate outbound connections (for software updates and AWS API calls) without being directly reachable from the internet. Two NAT Gateways, one per AZ, prevent a single NAT Gateway from becoming a point of failure.

Security Groups are configured with strict rules:

- ALB security group: accepts inbound HTTPS (443) from 0.0.0.0/0 onlyEC2 security group: accepts inbound traffic only from the ALB security group on the application port
- RDS security group: accepts inbound traffic only from the EC2 security group on port 3306 (MySQL) or 5432 (PostgreSQL)
- No security group allows inbound SSH from the internet

3. Web and Application Tier

Application Load Balancer (ALB) is deployed across both public subnets. It terminates HTTPS using a certificate provisioned by **AWS Certificate Manager**, so TLS is enforced for all traffic entering the application. The ALB performs health checks against each EC2 instance and stops routing traffic to any instance that fails the check.

AWS WAF is attached to the ALB to filter common web exploits including SQL injection, cross-site scripting, and requests from known malicious IP ranges.

Auto Scaling Group manages a fleet of EC2 instances spread across both private subnets in AZ-1 and AZ-2. The group maintains a minimum of two instances (one per AZ) so the application stays available even if one AZ experiences a disruption. Scaling policies add instances when average CPU utilization exceeds 70% and remove them when it drops below 30%, keeping capacity matched to actual demand.

EC2 instances run with **IAM Instance Roles** rather than any hardcoded credentials. The role grants only the permissions the application needs, such as reading from a specific S3 bucket or retrieving a secret from Secrets Manager.

AWS Systems Manager Session Manager is enabled on all EC2 instances. This removes the need to open port 22 for SSH entirely. All shell access goes through the Systems Manager console or CLI, and every session is logged to CloudWatch Logs.

AWS Secrets Manager stores the database credentials. The application retrieves them at runtime via the Secrets Manager API rather than reading them from a config file. Secrets Manager rotates the credentials automatically on a defined schedule.

4. Caching Layer

Amazon ElastiCache (Redis) is deployed in the private subnets between the application tier and the database. The application checks ElastiCache before querying RDS. Frequently read data such as session information, product listings, or configuration values is served from memory, reducing the number of queries that reach the database and lowering response latency.

ElastiCache is configured in cluster mode across both AZs so a single node failure does not take down the cache.

5. Database Tier

Amazon RDS (MySQL or PostgreSQL) replaces the self-managed database EC2 instance. RDS is deployed in **Multi-AZ** configuration, meaning AWS maintains a synchronous standby replica in the second AZ. If the primary instance fails or becomes unreachable, RDS automatically promotes the standby to primary within one to two minutes without any manual intervention.

The **RDS instance** sits in the isolated private database subnets with no route to the internet. The only inbound access allowed is from the EC2 security group on the database port.

Encryption at rest is enabled on the RDS instance using AWS KMS. All data written to disk, including automated snapshots, is encrypted.

Automated backups are enabled with a retention period of 7 days, giving the team the ability to restore to any point within the last week. **AWS Backup** manages the backup policy centrally and sends backup job reports to the operations team.

6. Monitoring and Observability

Amazon CloudWatch collects metrics from every layer of the architecture:

- EC2 instance metrics: CPU, memory (via CloudWatch Agent), disk utilization
- ALB metrics: request count, target response time, HTTP 5xx error rate
- RDS metrics: database connections, read/write latency, freeable memory
- ElastiCache metrics: cache hit rate, evictions, CPU utilization

CloudWatch Alarms are configured for each critical metric with thresholds that trigger **Amazon SNS** notifications to the operations team. A CloudWatch Dashboard gives the team a single view of application health.

AWS CloudTrail is enabled across all regions and logs every API call made in the account to an S3 bucket. CloudTrail logs are also sent to CloudWatch Logs so the team can query them and set alerts for specific events such as security group rule changes or IAM policy modifications.

Amazon GuardDuty runs continuously in the background, analyzing CloudTrail logs, VPC Flow Logs, and DNS logs for signs of malicious activity such as unusual API calls, communication with known malicious IPs, or credential compromise patterns.

7. Deployment Pipeline

AWS CodePipeline orchestrates the deployment workflow. When a developer pushes code to the main branch of the repository, CodePipeline triggers automatically:

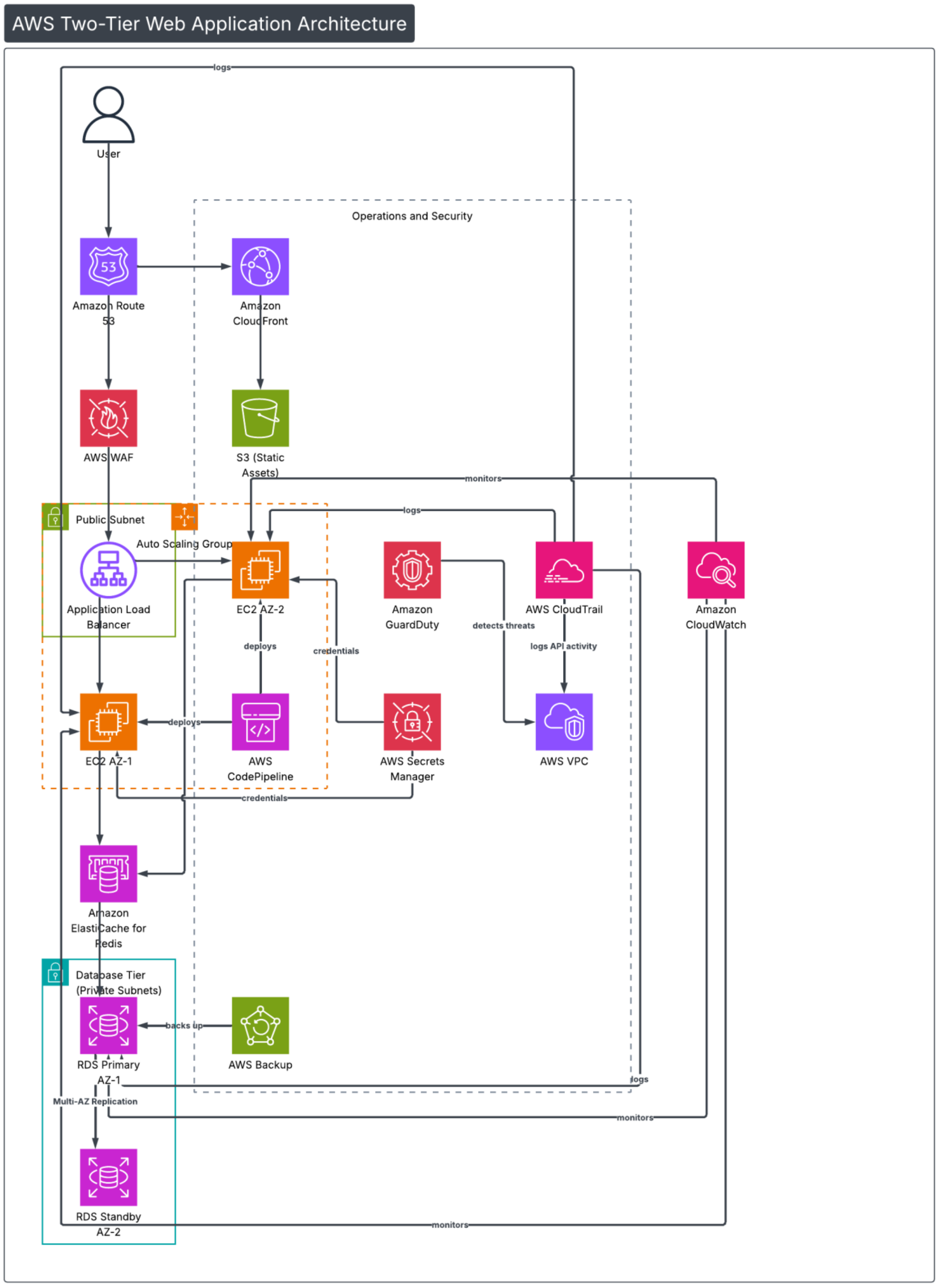
AWS CodeBuild runs automated tests and builds the application artifact

If tests pass, **AWS CodeDeploy** deploys the artifact to the Auto Scaling Group using a rolling deployment strategy so at least one instance stays in service throughout the deployment

If the deployment fails health checks, CodeDeploy rolls back automatically to the previous version

AWS CloudFormation defines the entire infrastructure as code. Every resource described above, the VPC, subnets, security groups, ALB, Auto Scaling Group, RDS instance, ElastiCache cluster, and IAM roles, is declared in CloudFormation templates stored in the same version-controlled repository as the application code.

Revised Architecture Diagram



How the Design Addresses Each WAF Pillar

WAF Pillar	How the Design Addresses It
Operational Excellence	CloudFormation manages all infrastructure as code; CodePipeline automates deployments with rollback; CloudWatch provides centralized observability
Security	Private subnets isolate the database; WAF and strict security groups control traffic; KMS encrypts data at rest; ACM enforces TLS; CloudTrail logs all API activity; GuardDuty detects threats
Reliability	Multi-AZ RDS with automatic failover; Auto Scaling Group across two AZs; ALB health checks; AWS Backup with 7-day retention
Performance Efficiency	ElastiCache reduces database read load; CloudFront serves static assets from edge locations; Auto Scaling matches compute to demand; Compute Optimizer guides right-sizing
Cost Optimization	Auto Scaling eliminates idle capacity; S3 and CloudFront reduce EC2 bandwidth costs; AWS Budgets and Cost Explorer provide spending visibility; Reserved Instances or Savings Plans reduce baseline compute cost

Reflection

Before this exercise, the gap between "moving to AWS" and "moving to AWS correctly" was easy to underestimate. Working through the Well-Architected Framework pillar by pillar made that gap concrete. A direct lift-and-shift of the two-tier application would have reproduced every on-premises weakness inside AWS, single points of failure, open network access, no backups, no visibility, just hosted on someone else's hardware.

The CAF assessment added a dimension that purely technical frameworks miss. An organization can design a sound architecture and still fail the migration because staff lack the skills to operate it, governance policies are absent, or leadership has no way to measure whether the migration delivered value.

The most practical takeaway is that security, reliability, and operational visibility are not features added after the architecture is built. Designing them out from the start, through private subnets, IAM roles, Multi-AZ deployments, and automated pipelines, costs less effort than retrofitting them later.