Quickbook — AWS Scalable Web App Infrastructure

Technical Report

---

1. Introduction

This project demonstrates the design and implementation of a **scalable, production-style web application architecture on AWS** using Infrastructure-as-Code (IaC), containerization, managed database services, content delivery networks, monitoring, and automated deployment pipelines.

The application,**Quickbook**, is a full-stack appointment-booking system consisting of:

- A Vue.js single-page application frontend
- A Node.js + Express REST API backend
- A MySQL relational database
- AWS-managed infrastructure deployed using Terraform
- CI/CD automation via GitHub Actions

The objectives of the project were to:

- Apply AWS best-practice architectural patterns
- Implement secure networking and isolation
- Deploy a highly available backend using Auto Scaling
- Serve static content globally using a CDN
- Enable monitoring, alarms, and cost controls
- Automate deployments

This report explains the architectural decisions, service selection rationale, security and scalability design, monitoring approach, limitations, and future improvements.

---

2. Architecture Overview

2.1 High-Level Design

The system follows a **three-tier architecture**:

1. **Presentation Layer**
    - Vue.js SPA hosted in Amazon S3
    - Distributed globally via CloudFront
2. **Application Layer**
    - Node.js API running in Docker containers
    - EC2 Auto Scaling Group behind an Application Load Balancer
    - API fronted by a CloudFront distribution for HTTPS
3. **Data Layer**
    - Amazon RDS MySQL in private subnets
    - Accessible only from backend security group

Traffic flow:

1. User accesses the frontend CloudFront URL.
2. Frontend calls the API CloudFront distribution over HTTPS.
3. CloudFront forwards requests to the ALB.
4. ALB routes to healthy EC2 instances.
5. Backend queries RDS.
6. Responses propagate back through CloudFront to the user.

Terraform manages all AWS resources, ensuring the entire environment can be recreated consistently.

2.2 AWS Services Used

Layer	Services
Networking	VPC, Subnets, Route Tables, IGW
Compute	EC2, Auto Scaling Group
Load Balancing	Application Load Balancer
Containers	Docker, Amazon ECR
Database	Amazon RDS MySQL
CDN	CloudFront
Static Hosting	Amazon S3
Secrets	SSM Parameter Store
Monitoring	CloudWatch

Cost Control	AWS Budgets
IaC	Terraform
CI/CD	GitHub Actions

---

3. Infrastructure and IaC Design

3.1 VPC and Networking

A custom VPC was created with:

- **Public subnets** for the Application Load Balancer
- **Private subnets** for EC2 instances
- **Private RDS subnet group**

This separation ensures:

- The database is never exposed to the public internet
- Backend servers cannot be accessed directly
- Only the ALB is internet-facing

3.2 Security Groups

Security groups enforce least-privilege connectivity:

- **ALB SG** → inbound port 80 from anywhere
- **Backend SG** → inbound port 4000 only from ALB SG
- **RDS SG** → inbound port 3306 only from backend SG

This layered approach prevents lateral movement and reduces attack surface.

3.3 Compute and Auto Scaling

Backend services run in Docker containers on EC2 instances managed by an Auto Scaling Group.

Key features:

- Desired capacity of one instance (Free Tier-friendly)
- Health checks from ALB target groups
- Instance refresh used for deployments

- Launch template installs Docker, logs into ECR, pulls images, and runs the container

Auto Scaling and health checks provide **self-healing**—if an instance becomes unhealthy, AWS replaces it automatically.

3.4 Database Layer

Amazon RDS MySQL was selected for:

- Automated backups
- Managed patching
- Monitoring integration
- High availability options

The database resides in private subnets and is accessed only by backend EC2 instances.

Schema initialization and data imports were performed directly against the RDS endpoint after connectivity was secured.

---

4. Application Deployment

4.1 Backend

The backend is a Node.js Express API exposing:

- `/health`
- `/appointments`
- `/db-check`

It uses the `mysql2` library with a connection pool to RDS.

Secrets such as the database password are retrieved at boot time from **SSM Parameter Store**, avoiding hard-coding credentials in Terraform or GitHub.

Docker images are stored in Amazon ECR and pulled automatically by EC2 instances during startup.

CORS restrictions allow requests only from the frontend CloudFront domain.

4.2 Frontend

The Vue.js application is built using Vite into static assets and uploaded to S3.

CloudFront provides:

- HTTPS
- Global distribution
- Caching
- Reduced latency

The API base URL is injected during build time so the frontend communicates with the secure CloudFront API endpoint.

---

5. Monitoring and Cost Management

5.1 CloudWatch Metrics and Alarms

CloudWatch dashboards and alarms were configured to monitor:

- ALB request counts and unhealthy hosts
- EC2 CPU utilization
- RDS CPU and storage
- Target group health

Alarms trigger when unhealthy hosts appear or storage drops below thresholds.

5.2 Cost Controls

AWS Budgets were configured with:

- Monthly spending limit
- Alert thresholds

This demonstrates cost-awareness and proactive cloud financial governance.

---

6. CI/CD Automation

GitHub Actions pipelines were added to automate deployments:

Backend Pipeline

- Builds Docker image
- Pushes to ECR
- Triggers ASG instance refresh

Frontend Pipeline

- Installs dependencies
- Builds SPA
- Syncs to S3
- Invalidates CloudFront cache

This supports:

- Versioning
- Repeatable deployments
- Rollback capability
- Reduced manual error

---

7. Security and Scalability Considerations

Security

- Private subnets for backend and DB
- SG-to-SG connectivity
- Secrets stored in SSM
- HTTPS enforced via CloudFront
- IAM roles attached to EC2
- No direct public DB access

Scalability

- ALB + ASG enable horizontal scaling
- Stateless backend containers
- CDN offloads frontend and API traffic
- RDS can be vertically scaled or upgraded to Multi-AZ

---

8. Limitations and Future Improvements

Current Limitations

- Single EC2 instance for cost reasons
- No Redis/ElastiCache layer
- Logs stored locally on EC2 rather than centralized
- Static IAM keys still used for CI/CD authentication
- No Web Application Firewall

Future Improvements

- GitHub OIDC for short-lived credentials
- ECS/Fargate migration
- Centralized logging via CloudWatch Agent
- ElastiCache for session and query caching
- WAF integration
- Blue/green deployments
- Custom domain names with ACM
- Multi-AZ RDS configuration

---

9. Conclusion

This project successfully delivered a **cloud-native, scalable AWS architecture** using best-practice patterns:

- Infrastructure-as-Code
- Secure network segmentation
- Auto-scaling compute
- Managed database services
- CDN-based delivery
- Monitoring and cost governance
- CI/CD automation

It demonstrates real-world cloud engineering skills applicable to production environments and provides a strong foundation for further expansion into enterprise-grade workloads.