*A*

*Report Submitted*

*for*

**Project Review 1**

*on the project entitled*

# Java Service on AWS EC2 via Infrastructure as Code

*submitted as part of the requirements for the course*

**BCSE408L - Cloud Computing (Theory Only)**

**Bachelor of Technology**

*in*

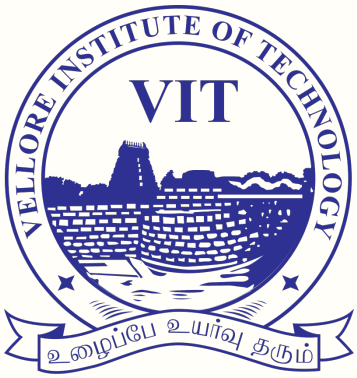
**Computer Science & Engineering**

*by*

**Akshat Sinha (22BCE2218), Rakshika (22BCE3675)**

*Under the guidance of*

**SIVA SHANMUGAM G**




Department of Computer Science & Engineering
Vellore Institute of Technology Vellore - 632014, INDIA

1 September 2025

# Abstract

Using Amazon EC2, AWS CloudFormation, AWS Systems Manager, Amazon S3, Amazon CloudWatch, AWS CodeDeploy, and AWS Budgets, this work designs and operates a small Java Spring Boot REST service on AWS. The infrastructure is defined declaratively with CloudFormation, covering compute, identity and access, logging, and cost controls. The service runs on a t3.micro instance as a systemd unit; artifacts are stored in S3; IAM enforces least-privilege access; Session Manager enables administrative access without public SSH; CloudWatch aggregates logs and alarms; Budgets provides spend guardrails; and CodeDeploy performs controlled in-place releases with automated health verification and rollback. The application exposes /health and /version endpoints and a minimal notes API used for functional and operational validation. We define three scenarios for evaluation: initial provisioning, version upgrade, and failure injection that triggers rollback and alerting. The result is a reproducible EC2-based service that demonstrates core cloud practices—infrastructure as code, immutable artifacts, automated deployment, and basic observability—while keeping cost and complexity low.



**Keywords:** Amazon EC2; AWS CloudFormation; AWS Systems Manager; AWS CodeDeploy; Amazon S3; Amazon CloudWatch; Infrastructure as Code; DevOps; cost management.

# 1 Introduction

Cloud platforms make it possible to provision repeatable infrastructure and deliver updates safely. This report studies a minimal yet realistic pattern: a Java Spring Boot REST service on Amazon EC2 defined with AWS CloudFormation as Infrastructure as Code (IaC), operated with AWS Systems Manager, observed with Amazon CloudWatch, released with AWS CodeDeploy, and cost-guarded with AWS Budgets.

## 1.1 Contributions

- A minimal EC2-based architecture with declarative stacks for compute, identity, logging, and cost control.
- Operational model using Session Manager (no public SSH), systemd-managed service, and CloudWatch alarms.
- Controlled in-place releases with CodeDeploy and a failure scenario that triggers rollback and alerting.

## 1.2 Scope and Assumptions

- Single t3.micro instance in the default VPC; security group allows HTTP only.
- Build artifacts in Amazon S3; logs and metrics in Amazon CloudWatch.
- No NAT gateway or load balancer by default; a short demo workload with health checks.

## 1.3 Paper Organization

Section 2 reviews background. Section 3 details the system architecture. Section 4 explains implementation. Section 5 presents the deployment workflow and scenarios. Section 6 defines the evaluation plan and success criteria. Section 7 outlines the project plan and milestones. Section 8 covers security considerations. Section 9 outlines limitations and future work. Section 10 concludes.

# 2 Background and Related Work

Cloud computing offers elastic, metered resources delivered as services. In Infrastructure-as-a-Service (IaaS), Amazon EC2 provides virtual machines (instances) that boot from Amazon Machine Images and attach persistent EBS volumes. Network access is controlled by security groups. This project uses a small general-purpose instance class to keep cost low while showing standard operations for a production-style service.

Infrastructure as Code (IaC) captures the desired state of cloud resources in versioned templates. AWS CloudFormation builds and updates stacks from these templates, tracks dependencies, and supports change sets and drift detection. IaC improves repeatability, review, and rollback. Alternative tools such as Terraform and AWS CDK provide similar goals; we select CloudFormation for native integration and a smaller toolchain.

Application release strategies aim to reduce risk. In-place deployment replaces the application on existing hosts and is simple but has brief service impact if not orchestrated. Blue/green keeps two environments and shifts traffic between them, which reduces risk at the cost of extra capacity. Canary is a controlled form of blue/green that shifts a small percentage first, then the remainder after health verification. AWS CodeDeploy supports these patterns with lifecycle hooks and automatic rollback on failed health checks or alarms.

Operations require visibility and safe access. Amazon CloudWatch aggregates logs and metrics and can raise alarms on resource health and application errors. AWS Systems Manager Session Manager provides shell access without opening public SSH, and Run Command supports controlled changes. IAM enables least-privilege roles for instances and automation, reducing the blast radius of any fault.

Cost governance is essential even for small projects. AWS Budgets warns when actual or forecasted spend crosses a threshold. Tagging resources by project and environment helps with cost allocation and cleanup. Avoiding fixed-cost components (for example NAT gateways and always-on load balancers) keeps idle cost minimal while preserving the learning goals.

Related work includes serverless designs that replace EC2 with AWS Lambda and API Gateway, and alternative IaC and deployment stacks (Terraform, CDK, GitHub Actions). We retain EC2 to demonstrate virtualization concepts, system initialization with systemd, and host-level observability while still applying modern automation practices.

Table 1: Release strategies used in practice

| | |
|---|---|
| In-place | Replace the application on current hosts; lowest cost; brief risk window if not drained. |
| Blue/green | Maintain two environments and switch traffic; safer rollbacks; requires extra capacity. |
| Canary | Shift a small percentage first, then the rest after health checks; balances safety and speed. |

# 3 System Architecture

## 3.1 Overview

The system hosts a small Java Spring Boot REST service on a single Amazon EC2 instance and augments it with declarative infrastructure, safe access, logging, and cost control. Core services are: Amazon EC2 (compute), AWS CloudFormation (IaC), AWS Identity and Access Management (IAM), AWS Systems Manager Session Manager (administrative access), Amazon S3 (artifact storage), Amazon CloudWatch (logs and alarms), AWS CodeDeploy (in-place releases with rollback), and AWS Budgets (spend guardrail). The instance runs in the default VPC with a security group that permits inbound HTTP and denies public SSH.

## 3.2 Component responsibilities

- EC2 instance: Amazon Linux 2023 t3.micro; EBS root volume; systemd unit launches the Spring Boot jar; Instance Metadata Service v2 enabled.
- IAM instance profile: least-privilege permissions for S3 object read, CloudWatch logs, and SSM core.
- CloudFormation stack: declares S3 bucket, IAM roles and policies, security group, EC2 instance, CloudWatch log group and alarms, and a budget.
- Systems Manager: Session Manager for shell access without open SSH; Run Command for controlled operational steps.
- S3 artifact bucket: versioned object storage for application zips and configuration assets.
- CloudWatch: application and system logs, CPU/StatusCheck alarms, optional custom metrics from the app.
- CodeDeploy (optional in minimal run): coordinates in-place deployments using lifecycle hooks and health verification.
- Budgets: notifies on actual or forecasted spend crossing a threshold.

## 3.3 Networking and access

The instance resides in a public subnet of the default VPC and receives a public IPv4 address. Inbound access is restricted to TCP/80; SSH is closed. Administrative sessions use Session Manager (encrypted channels, audit logs in CloudWatch or S3 if enabled). Outbound egress is open to AWS endpoints needed by S3, CloudWatch, and Systems Manager.

## 3.4 Deployment model

Build artifacts are produced as a fat jar, packaged with auxiliary scripts, and stored in S3. Releases are executed either via CodeDeploy (recommended for clean versioning and rollback) or via Systems Manager Run Command (lightweight). Health is checked via the /health endpoint; failures during installation or validation trigger rollback when using CodeDeploy.

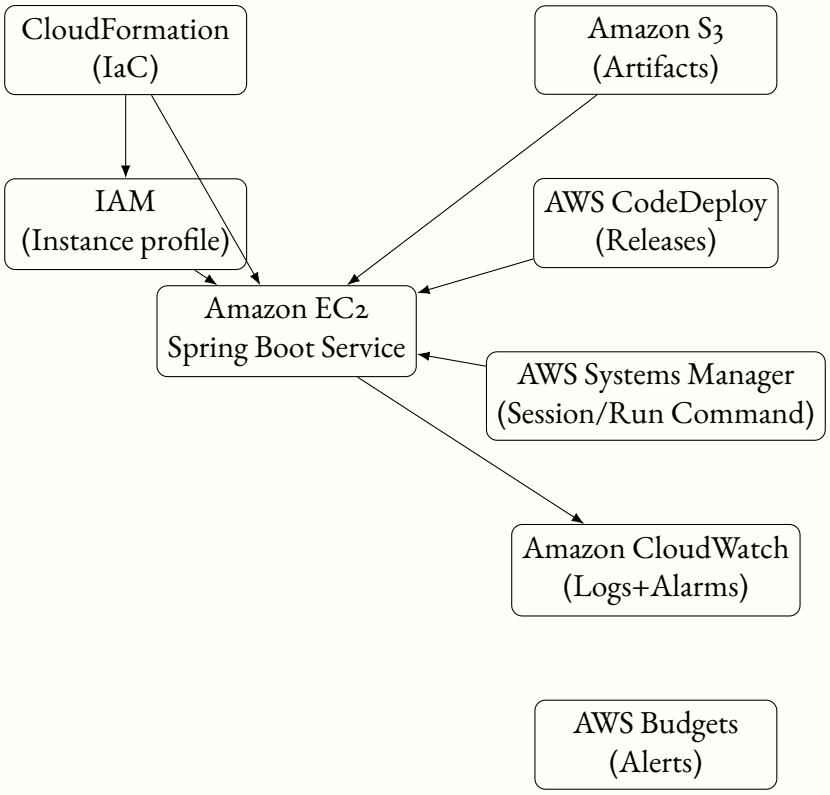



Figure 1: Logical architecture and service responsibilities.

## 3.5 Configuration and secrets

Application parameters (for example, port, log level, feature flags) are externalized. Non-sensitive values are set as environment variables; sensitive values should use SSM Parameter Store with KMS encryption. The instance role grants read access only to the required parameter paths.

## 3.6 Reliability, observability, and cost

Systemd restarts the service on failure. CloudWatch captures instance and application logs and raises alarms on StatusCheckFailed and sustained CPU pressure. The budget is set low (e.g., $5/month) and can be tuned per environment. All resources carry project and owner tags to simplify accounting and teardown.

# 4 Implementation and Methods

## 4.1 Infrastructure-as-Code templates

A single CloudFormation stack declares all required resources: an Amazon S3 bucket for artifacts, an IAM role and instance profile with least-privilege policies (S3 object read, CloudWatch logs, SSM core), a security group that allows inbound TCP/80 only, a t3.micro Amazon EC2 instance with an EBS root volume, a CloudWatch log group and two alarms (StatusCheckFailed and sustained CPU), and an AWS Budget. Parameters expose instance type, artifact bucket name, budget amount, and application port; tags record project, owner, and environment. Change sets are used for safe updates and to detect drift.

## 4.2 Instance bootstrapping and runtime

The instance runs Amazon Linux 2023 and Amazon Corretto 17. Cloud-init user data performs: package updates; creation of a dedicated `app` user; installation and configuration of the CloudWatch agent; creation of `/opt/app` and `/var/log/app`; and installation of a systemd unit that runs the Spring Boot fat jar with restart-on-failure policy. Instance Metadata Service v2 is enforced. On first boot, if an artifact is present in S3 it is downloaded and started; otherwise a placeholder service responds at `/health`.

## 4.3 Application

The service is a minimal REST endpoint set used for operational validation: `/health` (readiness probe), `/version` (build metadata), and a small notes API with create/read/update/delete to exercise logs and updates. Configuration (port, log level, feature flags) is externalized via environment variables; sensitive values, if any, are placed in SSM Parameter Store under a project-specific path with KMS encryption and read-only access from the instance role.

## 4.4 Release and rollback

Releases are executed with AWS CodeDeploy in in-place mode. The artifact is a zip containing the fat jar, an `appspec.yml`, and lightweight lifecycle scripts. The hooks perform: *BeforeInstall* (stop service), *AfterInstall* (unpack files, set ownership), *ApplicationStart* (enable and start the systemd unit), and *ValidateService* (HTTP check on `/health`). Auto-rollback is enabled on deployment failure or alarm breach; the previous artifact is restored and the service is restarted. For very small demos, Systems Manager Run Command can replace CodeDeploy with a shorter path at the cost of automated rollback.

## 4.5 Observability and cost controls

System logs and application logs are routed to an application-specific CloudWatch log group. Alarms fire on instance status failure and on sustained CPU utilization, notifying the default SNS topic or email (course policy dependent). The budget is set low (e.g., $5/month) with forecasted and actual alerts. All resources are tagged to support allocation and cleanup.

### 4.6 Verification

Provisioning is successful when the stack completes without error and the instance serves `/health` with HTTP 200. A release is considered successful when CodeDeploy reports *Succeeded* and the `/version` endpoint reflects the new build. A negative test intentionally fails *ValidateService* to confirm automatic rollback and alarms.

## 5 Deployment Workflow and Scenarios

### 5.1 Workflow

1. Build: package the Spring Boot service as a fat jar; embed build time and git commit for `/version`.
2. Package: zip the jar with `appspec.yml` and small lifecycle scripts.
3. Publish: upload the artifact to Amazon S3 under a versioned key (e.g., `java-service/{commit}.zip`); retain at least one prior version.
4. Provision/Update: apply the CloudFormation stack; confirm stack events show *UPDATE_COMPLETE*.
5. Deploy: create a CodeDeploy in-place deployment targeting the EC2 instance (tagged `App=JavaService`).
6. Validate: lifecycle hook requests `GET /health`; deployment succeeds only on HTTP 200; logs stream to CloudWatch.
7. Rollback policy: enable automatic rollback on deployment failure or alarm breach; CodeDeploy restores the last good artifact.
8. Teardown (optional): delete the stack and empty the artifact bucket after the review to control cost.

### 5.2 Scenarios

**S1: Initial provisioning and v1 release** Create the stack, publish `v1`, and deploy. Success criteria: stack completes; instance reachable on HTTP 80; `/health` returns 200; `/version` shows the expected commit.

**S2: Version upgrade (v1 → v2)** Change a visible response (e.g., banner text or minor endpoint behavior), publish `v2`, and redeploy. Success criteria: CodeDeploy status *Succeeded*; `/version` reflects the new commit; no alarms fire.

**S3: Failure injection and rollback** Introduce a deliberate failure (e.g., make `/health` return 500 during *ValidateService*) and deploy. Expected outcome: CodeDeploy marks *Failed*, performs automatic rollback to the previous artifact, and service recovers without manual intervention.

### 5.3 Measurements and evidence

- Deploy duration: CodeDeploy timeline from *Created* to *Succeeded/Failed*.
- Health outcomes: HTTP status for `/health` before and after each deployment.
- Resource signals: CPU and StatusCheck alarms in CloudWatch (should remain normal for S1–S2; S3 triggers failure then recovery).
- Cost: monthly budget threshold and a short snapshot of estimated spend during the review window.

## 6 Evaluation Plan and Success Criteria

### 6.1 Objectives

Validate that (i) infrastructure provisions repeatably, (ii) the service updates safely with rollback, (iii) logging and alarms provide actionable signals, and (iv) monthly cost stays low.

### 6.2 Metrics and targets

Table 2: Planned evaluation metrics

| Metric | Target |
|---|---|
| Provision success | 100% stack completion without manual fixes. |
| Deployment success | 100% for normal updates (S2). |
| Rollback effectiveness | Service restored to last good version after induced failure (S3). |
| Deployment time | $\leq$5 minutes per release (build excluded). |
| Health during update | $\geq$99% /health success over the update window. |
| Cost guardrail | Forecasted monthly spend $\leq$\$5 with alerting enabled. |

### 6.3 Planned scenarios

S1: initial provisioning and v1 release; S2: v1$\rightarrow$v2 update; S3: induced failure triggering rollback. Evidence will be collected from CloudFormation events, CodeDeploy deployment details, and CloudWatch logs/alarms; results will be summarized at the final review.

## 7 Project Plan and Milestones

- M0: Design and Project Review 1 submission
- M1: CloudFormation stack (EC2, IAM, SG, S3, logs, budget)
- M2: Instance bootstrap (systemd, SSM, logging)
- M3: Artifact packaging and first deploy (v1)

- M4: Alarms and budget verification
- M5: Update (v2) and rollback scenario (S3)
- M6: Final write-up and demo

### 7.1 Risks and mitigations

Quota or permission errors (pre-check IAM and quotas); cost creep (budget alerts, avoid fixed-cost components); time constraints (small milestones and teardown after tests).

## 8 Security Considerations

- Least privilege: the instance profile grants only S3 object read, CloudWatch logs, and SSM core; resource paths are scoped by prefix and tag conditions.
- Network: security group allows inbound TCP/80 only; SSH closed; egress limited to required AWS endpoints where feasible.
- Access: administrative sessions use Systems Manager Session Manager; actions are audited (CloudWatch/CloudTrail if enabled).
- Host hardening: IMDSv2 enforced; systemd restarts on failure; regular OS updates (via SSM Patch Manager in later milestone).
- Secrets/config: non-sensitive via environment variables; sensitive values planned in SSM Parameter Store with KMS encryption and read-only access.
- Artifact integrity: S3 versioning retained; optional checksum verification or code-signing planned for later milestone.
- S3 artifact bucket hardening: block all public access, enable versioning, enforce server-side encryption (SSE-S3 or KMS), and deny non-TLS and unencrypted uploads via a bucket policy.

## 9 Limitations and Future Work

- Single instance in default VPC; no multi-AZ or auto scaling.
- No load balancer by default; blue/green or canary may be demonstrated later with a short-lived ALB.
- No database layer; current API is for operational validation only.
- Manual approvals and richer CI/CD, code-signing, and parameterized environments (dev/stage) are planned for the final review.

# 10 Conclusion

This review defined a minimal, cloud-oriented architecture for a Java Spring Boot service on Amazon EC2 using Infrastructure as Code. The design emphasizes repeatable provisioning, safe operational access without public SSH, and basic observability and cost control. The next milestone implements the stack, performs the first release, and prepares the update and rollback scenarios for evaluation.

# References

# References


[1] Amazon Web Services, "Amazon EC2 User Guide for Linux Instances," Available at: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/. Accessed: 1 Sept. 2025.

[2] Amazon Web Services, "AWS CloudFormation User Guide," Available at: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/. Accessed: 1 Sept. 2025.

[3] Amazon Web Services, "AWS CodeDeploy User Guide," Available at: https://docs.aws.amazon.com/codedeploy/latest/userguide/. Accessed: 1 Sept. 2025.

[4] Amazon Web Services, "AWS Systems Manager User Guide," Available at: https://docs.aws.amazon.com/systems-manager/latest/userguide/. Accessed: 1 Sept. 2025.

[5] Amazon Web Services, "Amazon CloudWatch User Guide," Available at: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/. Accessed: 1 Sept. 2025.

[6] Amazon Web Services, "Amazon Simple Storage Service User Guide," Available at: https://docs.aws.amazon.com/AmazonS3/latest/userguide/. Accessed: 1 Sept. 2025.

[7] Amazon Web Services, "AWS Budgets User Guide," Available at: https://docs.aws.amazon.com/cost-management/latest/userguide/budgets-what-is.html. Accessed: 1 Sept. 2025.

[8] Amazon Web Services, "AWS Well-Architected Framework," Available at: https://docs.aws.amazon.com/wellarchitected/latest/framework/. Accessed: 1 Sept. 2025.