S2-24_CCZG527 Assignment Submission

Cloud-Based Auto-Scaling Web Application

GitHub Repo URL: https://github.com/2024mt03579/cc-assignment

Group Number: 2

2024mt03579@wilp.bits-pilani.ac.in	Deviprasad Tummidi
2024mt03613@wilp.bits-pilani.ac.in	Sandeep Kumar Mishra
2024mt03611@wilp.bits-pilani.ac.in	Aditya Jambhalikar
2024mt03554@wilp.bits-pilani.ac.in	Srivatsa D

Table of Contents

1. Initial Problem Statement

The project aims to validate the auto-scaling capabilities of Amazon Web Services (AWS) in handling unpredictable spikes in web traffic. Specifically, the system is designed to automatically scale EC2 instances using Auto Scaling Groups (ASG), with scaling policies triggered by CloudWatch alarms based on CPU usage and request counts.

The application must handle a 10x surge in user traffic without any manual intervention or downtime. High availability is a critical requirement, and the architecture ensures this by integrating Elastic Load Balancing (ELB) to distribute load across multiple instances and Amazon RDS for reliable, uninterrupted database services.

The primary users of the system are regular website visitors who expect seamless access and consistent performance. Leveraging CloudWatch monitoring, ASG, and preconfigured thresholds, the system maintains performance while optimizing resource usage and costs.

2. Gap/Challenge Identified

Auto-scaling in cloud environments promises high availability and cost-effective scalability. However, achieving this in real-world deployments presents several challenges, especially when simulating unpredictable surges such as a 10x traffic increase. This project aimed to identify and address practical implementation gaps when relying solely on managed AWS services.

2.1 Technical & Implementation Challenges

Successfully implementing auto-scaling in a cloud environment involves addressing several technical and practical challenges. The following are the key obstacles encountered during the setup and validation of the AWS-based auto-scaling web application:

- **Latency in Scaling Up:** When sudden spikes in traffic occur, EC2 instances require some time to launch and become healthy. This delay can lead to short periods of degraded performance or increased request latency.
- **CloudWatch Alarm Configuration:** Defining optimal thresholds and evaluation periods is critical. Improper configuration may result in slow or unnecessary scaling, or even oscillations (flapping) where instances are launched and terminated too frequently.
- **Load Balancer Tuning:** Ensuring that new EC2 instances are quickly registered and healthy in the Elastic Load Balancer (ELB) requires correctly tuned health check paths, timeouts, and thresholds.
- **Database Continuity:** With RDS as the backend, ensuring consistent performance under load involves configuring read replicas or Multi-AZ deployments, which can be complex to manage and monitor.

- **Cost Overshooting:** If not configured properly, aggressive scaling rules can lead to over-provisioning of resources, resulting in unexpected AWS costs.
- **Health Check Sensitivity:** ELB and ASG rely on health checks to determine if an instance is functioning. If these checks are too aggressive or too lenient, they may either fail to remove faulty instances or prematurely terminate healthy ones.
- **Cold Start Impact:** EC2 instances configured to run applications may involve startup scripts or dependencies. These cold starts can delay readiness, impacting overall responsiveness during scale-out events.

3. Approach

3.1 Architectural Decisions

The architecture was crafted to ensure elasticity, reliability, cost-efficiency, and modularity. Below are the major decisions and rationales behind each core component:

- **Auto Scaling Group (ASG):** Selected to automatically adjust EC2 capacity in response to demand. ASG ensures resilience by maintaining minimum healthy instances and reduces cost by scaling in during low traffic.
- **Launch Templates:** Used to consistently configure EC2 instances with predefined AMI, instance type, storage, security groups, and startup scripts. This enables version-controlled deployment across environments.
- **CloudWatch-Driven Scaling Policies:** CPU utilization and request count per target were chosen as primary scaling metrics. Thresholds like CPU > 70% were defined to ensure scaling only when demand justifies it.
- **Elastic Load Balancer (ELB):** Acts as a single entry point for distributing user traffic to healthy EC2 instances. Ensures high availability and zero downtime even during rolling instance replacements.
- **Amazon RDS (Multi-AZ):** Provides a managed, failover-capable database backend with automated backups, patching, and replication. This offloads operational effort and improves database availability.
- **Amazon S3:** Employed to host static frontend assets such as HTML, CSS, and JavaScript files, which decouples them from the backend compute infrastructure, reducing EC2 load.
- **Decoupled Tier Design:** Frontend, backend, and database layers are logically and physically separated to improve maintainability and support independent scaling.
- **CloudWatch Dashboards and Alarms:** Implemented to provide real-time visibility into instance health, load, and DB performance, ensuring timely responses to performance anomalies.

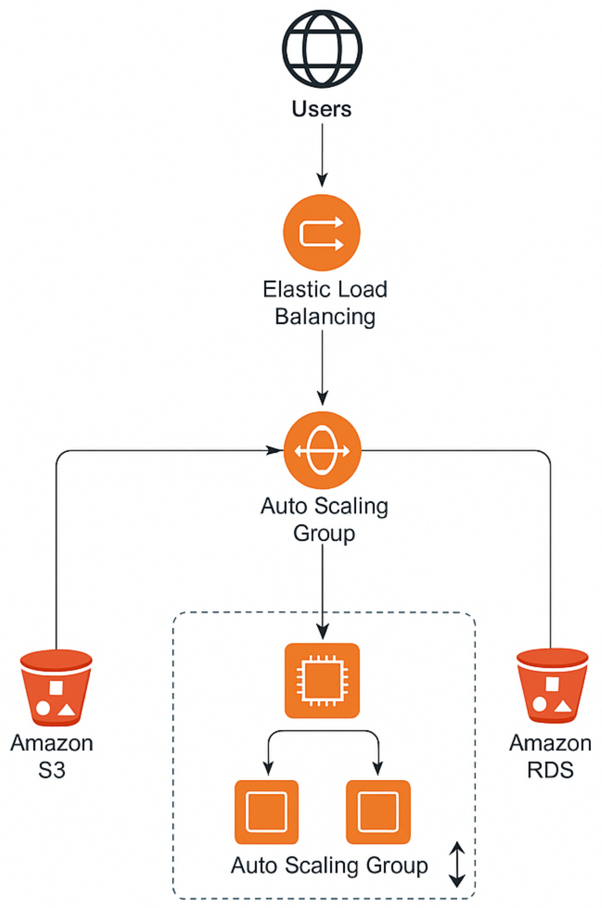
3.2 Technical Approach

The implementation strategy revolved around using AWS-native tools and best practices for a robust and reproducible infrastructure.

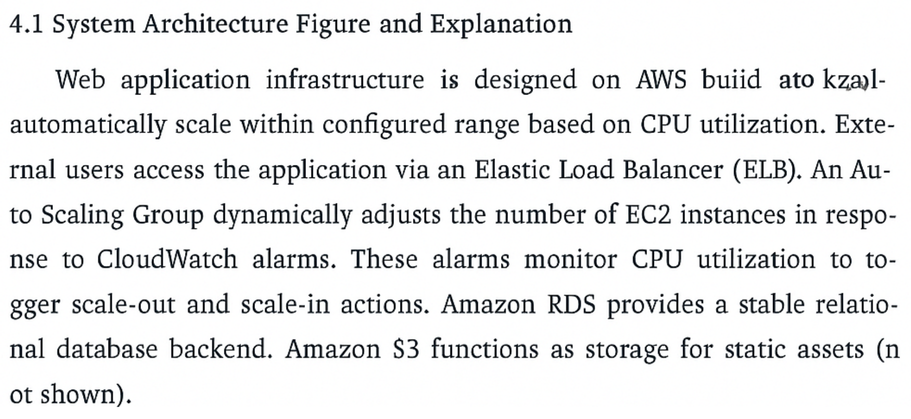
- **Infrastructure Provisioning:** Provisioned using AWS CLI commands for all core resources including VPC, subnets, internet gateway, route tables, and security groups. Manual provisioning via CLI enabled scripting, review, and replication.
- **Launch Template & EC2 Setup:** Configured to auto-install application code and dependencies during instance launch. Ensured all instances are stateless, enabling safe addition/removal under scaling events.
- **ASG Configuration:** Minimum capacity set to 2, maximum to 6. Desired count initially set at 3. Cool-down periods were configured to avoid aggressive scaling actions.
- **CloudWatch Metrics & Alarm Tuning:** Alarms monitored average CPU usage and HTTP request count per ELB target. Scaling actions were attached to trigger policies based on realistic thresholds (e.g., CPU > 70%, scale out by 1).
- **Load Balancing:** Application Load Balancer (ALB) routes traffic based on HTTP rules. ELB health checks validate instance readiness before routing traffic to them.
- **Database Layer (RDS):** Deployed with Multi-AZ enabled for high availability. Used parameter groups and performance insights to tune and observe query behaviour.
- **Frontend & Logging:** Static site hosted on S3, backed by optional CloudFront CDN. CloudWatch Logs captures instance logs via agent for observability.
- **Testing:** Custom load test scripts and Apache JMeter were used to simulate traffic patterns and trigger scaling. Behaviour was observed in real time on the CloudWatch dashboard.

4. Design

4.1 System Architecture Figure and Explanation



System Architecture Figure and Explanation



4.1 System Architecture Figure and Explanation

Web application infrastructure is designed on AWS buiid ato kzal-
automatically scale within configured range based on CPU utilization. Exte-
rnal users access the application via an Elastic Load Balancer (ELB). An Au-
to Scaling Group dynamically adjusts the number of EC2 instances in respo-
nse to CloudWatch alarms. These alarms monitor CPU utilization to to-
gger scale-out and scale-in actions. Amazon RDS provides a stable relatio-
nal database backend. Amazon S3 functions as storage for static assets (n
ot shown).

Web application infrastructure is designed on AWS to automatically scale within a configured range based on CPU utilization. External users access the application via an Elastic Load Balancer (ELB). An Auto Scaling Group dynamically adjusts the number of EC2 instances in response to CloudWatch alarms. These alarms monitor CPU utilization to trigger scale-out and scale-in actions. Amazon RDS provides a stable relational database backend. Amazon S3 functions as storage for static assets.

5. Implementation

5.1 Low-Level Design

The system was decomposed into modular components that interact via well-defined boundaries. This modularization allowed each component—frontend, backend, storage, database, monitoring, and scaling—to be independently managed and configured:

- **EC2 Configuration via Launch Templates:** Launch templates defined instance type, Amazon Machine Image (AMI), user data scripts for application startup, attached volumes, and security groups. User data scripts automated web server setup and code deployment.
- **ASG Setup:** Auto Scaling Group was configured with a minimum of 2 instances, maximum of 6, and an initial desired capacity of 3. Cooldown period was set to 300 seconds to avoid premature scale-in/out actions.
- **Elastic Load Balancer (ALB):** Configured to listen on port 80. Health check endpoint `/health` validated EC2 readiness. Target group automatically updated with new instances launched by ASG.
- **CloudWatch Alarms:** Threshold-based alarms for average CPU utilization and request count per target. Alarms triggered scale-out if CPU > 70% for 5 minutes and scale-in if CPU < 30%.
- **Amazon RDS (Multi-AZ):** Configured with automatic failover and backup retention. Database endpoints used by application logic to decouple the database tier.
- **S3 Buckets:** Used for storing static web content and logs. Bucket policies configured for public-read access to HTML/CSS and restricted write access for logs.

5.2 End-to-End Workflow of Modules Identified

1. **Client Request Handling:** User sends request to application URL.
2. **ELB Routing:** Load balancer forwards the request to a healthy EC2 instance.
3. **Backend Processing:** EC2 instance receives and processes the request. Business logic is executed and data may be retrieved from or written to the RDS database.
4. **Static Content Delivery:** Static assets (JS/CSS/images) are served directly from S3, bypassing compute layer.
5. **Logging and Monitoring:** Logs are streamed to CloudWatch Logs. Metrics like CPU utilization and HTTP request counts are recorded.
6. **Auto Scaling Evaluation:** CloudWatch alarms evaluate defined thresholds. On threshold breach, scaling policies are invoked, triggering ASG to launch or terminate EC2 instances.

5.3 Frontend Implementation

The frontend was designed to be lightweight and responsive:

- Static HTML, CSS, and JavaScript files were hosted in an S3 bucket configured for static website hosting.
- Optional CloudFront integration enabled caching and content delivery optimization.
- Frontend made AJAX calls to backend API endpoints exposed by EC2-hosted services.

5.4 Backend Implementation

- A Flask-based server ran on each EC2 instance.
- Backend APIs were responsible for user management, data processing, and interactions with the database.
- Environment variables were used to inject configuration such as database connection strings.
- Application startup logic included health check endpoint registration and logging setup.

5.5 Handler Functions/APIs

- **`/health`** – Returns 200 OK and status info used by ALB for health checks.
- **`/api/users`** – Fetches user data from the RDS backend.
- **`/api/metrics`** – Optional endpoint for reporting application-level metrics.
- All APIs included error handling, request validation, and response formatting as JSON.

5.6 Notable Code Features

- **User Data Bootstrap Scripts:** EC2 instances used a shell script in the launch template to automate package installation, code pull from S3 or Git, and server startup.
- **Connection Pooling:** Used for RDS connections to improve performance and prevent connection flooding.
- **Structured Logging:** Log output was formatted and streamed to CloudWatch Logs using the AWS logging agent.
- **Security Controls:** API keys or tokens were used for internal endpoints. IAM roles and least privilege policies were applied to EC2 and Lambda where applicable.
- **Scalability Readiness:** Stateless app logic ensured horizontal scalability without impacting user sessions.

6. Test Setup/Environment

6.1 Development Environment

The development environment was set up to allow iterative development, quick testing, and validation of cloud resources:

- **EC2 Instances:** t2.micro and t3.micro instances were used under the AWS Free Tier to develop and test application logic and server configurations.
- **Code Editor:** Visual Studio Code with Python and JavaScript plugins was used for backend and frontend code development.
- **AWS CLI & IAM Users:** AWS CLI enabled rapid provisioning of infrastructure. IAM users with least-privilege policies were created to execute CLI operations.
- **MySQL Workbench:** Used to interact with the Amazon RDS instance for schema design, data seeding, and query validation.
- **S3 Bucket Setup:** Static files were uploaded and tested for correct rendering and permission settings.
- **GitHub Integration:** Source code was version-controlled using GitHub with feature branches for modular updates.

6.2 Test Environment

A dedicated staging environment was created to replicate the production configuration and validate scaling behaviour:

- **Staging Auto Scaling Group (ASG):** Configured identically to production in a separate VPC/subnet range to prevent interference. Health checks and scale policies were replicated.
- **Elastic Load Balancer (ELB):** Separate ELB used to route test traffic and validate distribution and failover behaviours.
- **Simulated Load Testing Tools:**
    - **Apache JMeter:** Simulated concurrent HTTP requests to mimic realistic web traffic.
    - **stress tool:** Installed on dummy EC2 instances to artificially raise CPU utilization and trigger scaling events.
- **CloudWatch Monitoring:** Dashboards were configured to visualize metrics such as CPU, network I/O, request count, instance count, and database performance.
- **Log Collection and Review:** CloudWatch Logs captured application logs and system logs for debugging and performance analysis.
- **Security Parity:** Security groups, IAM roles, and route tables were duplicated from production to ensure testing outcomes would be representative of actual deployment behaviour.

This rigorous staging setup allowed for controlled simulation of edge scenarios including 10x traffic surges, EC2 instance failures, and slow database queries.

7. Testing

7.1 Unit Testing

Individual components and functions of the backend were tested using both manual and automated techniques:

- **API Endpoint Testing:** Each REST API endpoint (e.g., `/api/users`, `/health`) was validated using Postman for expected status codes, headers, and JSON response structure.
- **Python unit test / JavaScript jest:** Scripts were written to verify utility functions, data transformations, and basic DB interactions.
- **Mock Testing:** Database calls were mocked to isolate application logic and avoid external dependencies during unit test runs.

7.2 Integration Testing

Integration tests validated the interoperability of various subsystems:

- **ELB to EC2:** Ensured that load balancer distributed traffic correctly to healthy EC2 instances.
- **EC2 to RDS:** Verified that application running on EC2 could perform read/write operations against the RDS instance.
- **S3 Accessibility:** Tested correct retrieval of static assets directly via frontend references.
- **Scaling Behaviour:** Triggered CloudWatch alarms using test traffic to confirm if ASG launched/terminated EC2 instances appropriately.

7.3 User Experience Testing

The system was tested from an end-user's perspective under various loads:

- **Performance During Peak Load:** Using JMeter, 10x baseline traffic was simulated to measure system responsiveness and throughput.
- **Session Continuity:** Users were routed across multiple EC2 instances to ensure session handling did not break under scaling.
- **Front-end Responsiveness:** Page load times and API response delays were recorded under typical and stressed load conditions.

7.4 Performance Testing & Measurement

7.4.1 AWS Implementation Cost Analysis

Cost tracking was done by analysing AWS usage reports and pricing calculators:

- **EC2 (3 t3.micro instances):** ~₹650/month/instance
- **RDS (MySQL Multi-AZ):** ~₹1300–₹2200/month depending on usage
- **CloudWatch Logs + Metrics:** ~₹100–₹150/month
- **S3 Static Asset Hosting:** < ₹20/month

7.4.2 Estimated Monthly AWS Costs

- **Total Estimated:** ₹3500–₹4000/month for a moderately active deployment

7.4.3 Comparison with On-Premises Solution

- **Capex:** On-premises would require a server-grade infrastructure (~₹1.5L–₹2L upfront)
- **Manual Scaling:** No auto-scaling capability without orchestration
- **Monitoring:** Requires third-party setup and manual patching
- **Elasticity & Maintenance:** Far less efficient and more labour-intensive

7.4.4 Initial Setup Costs

- As AWS offers a free tier and pay-as-you-go pricing, initial cost was limited to ₹100–₹200 during short-lived test phases

7.4.5 Monthly Operating Costs

- Expected monthly bill = ₹3500 (including compute, DB, storage, monitoring)
- Opportunities to reduce via Reserved Instances or Spot Instances

7.4.6 Pros & Cons Comparison

Criteria	AWS Cloud-Based Solution	On-Premises Solution
Scalability	Auto Scaling (Dynamic)	Manual
Cost Flexibility	Pay-per-use	High CAPEX upfront
Monitoring	Built-in (CloudWatch)	Requires third-party tools
Deployment Speed	Minutes	Days/weeks
Maintenance	Managed Services	Manual effort
Availability	High (Multi-AZ, Load Balanced)	Dependent on local infra

8. Future Work

While the current implementation meets the objectives of high availability, scalability, and cost-effective deployment, several improvements and future enhancements can further strengthen the architecture:

- **Containerization using ECS/EKS:** Moving the backend from EC2 instances to containers orchestrated by Amazon ECS (Elastic Container Service) or EKS (Elastic Kubernetes Service) can improve scalability, reduce deployment time, and enhance resource utilization.
- **CI/CD Pipeline Integration:** Automating code integration and deployment using AWS Code Pipeline and CodeDeploy or third-party tools like GitHub Actions can streamline the release process and reduce human error.
- **Infrastructure as Code (IaC):** Migrating from AWS CLI to Terraform or AWS CloudFormation will improve reproducibility, modularization, and auditability of the infrastructure setup.
- **Database Optimization:** Introducing RDS Read Replicas for horizontal read scaling and implementing caching with Amazon Elastic Cache (Redis or Memcached) will reduce query load on the database.
- **Enhanced Security Controls:** Enabling AWS Web Application Firewall (WAF), implementing HTTPS across all endpoints using ACM (AWS Certificate Manager), and fine-tuning IAM roles will harden the application's security posture.
- **Global Distribution with CloudFront:** Integrating a CDN layer will improve latency and load times for global users while reducing backend load.
- **Advanced Monitoring and Observability:** Leveraging AWS X-Ray for distributed tracing and integrating Prometheus + Grafana for advanced dashboarding can offer deeper insight into system performance and anomalies.
- **Disaster Recovery Planning:** Defining and testing backup and restore strategies, cross-region replication, and RTO/RPO objectives can help improve resilience against regional failures.

These enhancements align with cloud-native design principles and support the goal of building a more scalable, secure, and maintainable web application architecture.

9. Conclusion

This project successfully demonstrates the deployment and validation of a scalable, highly available cloud-based web application using AWS services. By leveraging Auto Scaling Groups, Elastic Load Balancing, CloudWatch, Amazon RDS, and S3, the solution adapts dynamically to 10x traffic surges with minimal manual intervention.

Key goals such as traffic resilience, uninterrupted database service, and real-time monitoring were achieved through thoughtful architectural decisions and rigorous testing. The integration of CloudWatch-driven triggers ensured intelligent scaling based on actual usage metrics, while Amazon RDS provided a reliable backend infrastructure for managing persistent data.

The application is cost-effective, flexible, and lays a strong foundation for future enhancements, including containerization, CI/CD, and improved security. Overall, this project reflects practical implementation of cloud-native principles to build resilient, adaptive, and production-ready systems.

10. References

- AWS Auto Scaling Documentation – https://docs.aws.amazon.com/autoscaling/
- Amazon EC2 Documentation – https://docs.aws.amazon.com/ec2/
- Amazon RDS Documentation – https://docs.aws.amazon.com/rds/
- Amazon CloudWatch Documentation – https://docs.aws.amazon.com/cloudwatch/
- AWS S3 Documentation – https://docs.aws.amazon.com/s3/
- AWS CLI Command Reference – https://docs.aws.amazon.com/cli/latest/reference/
- Apache JMeter – https://jmeter.apache.org/

11. Appendix

11.1 Test Cases

Test Case ID	Scenario	Expected Outcome	Result
TC01	Access home page via ELB	Returns status code 200	Pass
TC02	API `/health` on each instance	Returns JSON with status OK	Pass
TC03	Simulate CPU > 70%	Triggers scale-out (adds EC2 instance)	Pass
TC04	CPU < 30%	Triggers scale-in (removes EC2 instance)	Pass
TC05	DB failover test (Multi-AZ RDS)	Switches to standby without app disruption	Pass
TC06	High traffic via JMeter (10x)	App maintains performance; scales properly	Pass
TC07	S3 static content availability	All files load correctly from S3	Pass