Northeastern University

College of Engineering

Master's in Telecommunications Networks

Multi-Tier Application Deployment with Terraform, Ansible, and AWS Auto-Scaling

Capstone Project Report

TELE6420 - Infrastructure Automation and DevOps Tools

Submitted by:
Chetan Pavan Sai
Nannapaneni

Date:
15th December 2025

Contents

1 Executive Summary

This report documents the design, implementation, and testing of a three-tier web application deployed on AWS using Infrastructure as Code (IaC) principles. The project demonstrates practical application of Terraform for infrastructure provisioning, Ansible for configuration management, and AWS Auto Scaling for dynamic resource allocation.

The deployed architecture consists of an Nginx frontend serving static content, a Flask API tier handling business logic, and an RDS MySQL database for persistent storage. Load testing successfully triggered auto-scaling from 2 to 4 instances, validating the scaling policies.

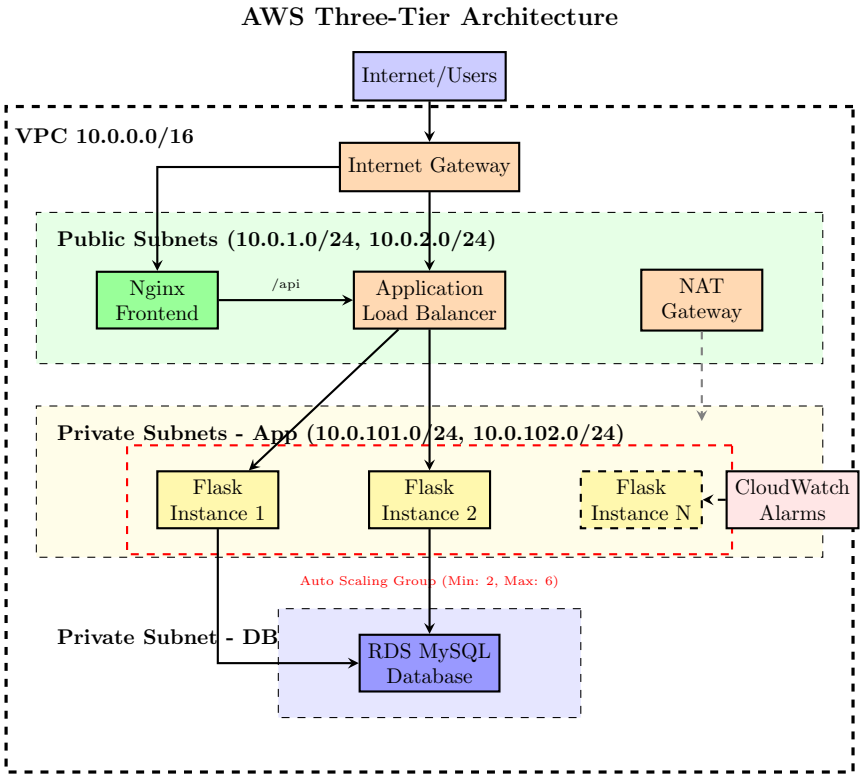
2 Architecture Overview

2.1 High-Level Design

The application follows a classic three-tier architecture pattern:

1. **Presentation Tier (Nginx):** Serves static HTML/CSS/JS content and proxies API requests
2. **Application Tier (Flask):** Handles business logic and database queries
3. **Data Tier (RDS MySQL):** Stores persistent application data

2.2 Architecture Diagram



2.3 Request Flow

1. User accesses Nginx frontend via public IP
2. Nginx serves static HTML and proxies /api requests to ALB
3. ALB distributes requests across Flask instances in private subnets
4. Flask instances query RDS MySQL database
5. Response flows back through the same path

2.4 Network Design

Subnet Type	CIDR Block	AZ	Resources
Public Subnet 1	10.0.1.0/24	us-east-1a	Nginx, ALB, NAT GW
Public Subnet 2	10.0.2.0/24	us-east-1b	ALB Node
Private Subnet 1	10.0.101.0/24	us-east-1a	Flask Instances
Private Subnet 2	10.0.102.0/24	us-east-1b	Flask Instances, RDS

Table 1: Subnet allocation

3 Terraform Implementation

3.1 Module Structure

The Terraform code is organized into five reusable modules:

```
terraform/
    main.tf              # Root module - orchestrates all modules
    variables.tf         # Input variables
    outputs.tf           # Output values
    modules/
        networking/      # VPC, subnets, IGW, NAT GW, route tables
        alb/             # Application Load Balancer, target group
        asg/             # Launch template, ASG, scaling policies
        rds/             # RDS MySQL instance, subnet group
        frontend/        # Nginx EC2 instance
```

3.2 Module Descriptions

Networking Module: Creates the foundational network infrastructure including VPC with DNS support enabled, public and private subnets across two availability zones, Internet Gateway for public internet access, NAT Gateway allowing private instances to reach the internet for updates, and separate route tables for public (via IGW) and private (via NAT) subnets.

ALB Module: Provisions an internet-facing Application Load Balancer with HTTP listener on port 80, target group configured for Flask instances on port 5000, and health checks against the /health endpoint.

ASG Module: Implements the auto-scaling infrastructure with a launch template specifying Amazon Linux 2 AMI and t3.micro instance type, Auto Scaling Group with min=2, max=6, desired=2 configuration, scale-out policy triggered when CPU exceeds 50% for 2 consecutive minutes, and scale-in policy triggered when CPU drops below 30%.

RDS Module: Deploys a db.t3.micro MySQL instance in private subnets with security group restricting access to application tier only.

Frontend Module: Creates the Nginx EC2 instance in a public subnet with user data script installing Nginx and configuring reverse proxy.

3.3 Key Terraform Resources

Total resources provisioned: **30**

- 1 VPC, 4 Subnets (2 public, 2 private)
- 1 Internet Gateway, 1 NAT Gateway + Elastic IP
- 2 Route Tables with associations
- 1 Application Load Balancer, 1 Target Group, 1 Listener
- 1 Launch Template, 1 Auto Scaling Group
- 2 Scaling Policies, 2 CloudWatch Alarms
- 1 RDS Instance, 1 DB Subnet Group
- 1 Nginx EC2 Instance
- 5 Security Groups

4 Ansible Configuration

4.1 Directory Structure

```
ansible/
    ansible.cfg
    inventory/
        hosts.yml
        group_vars/
            all.yml
    playbooks/
        site.yml
    roles/
        common/
        flask/
        frontend/
```

4.2 Configuration Approach

Due to the Flask instances being in private subnets without public IPs, configuration was performed by SSH tunneling through the Nginx frontend acting as a bastion host. The deployment process involved copying the SSH key to the Nginx instance, connecting to private instances via their private IPs, and deploying the Flask application with database credentials.

4.3 Flask Application Deployment

The Flask application was deployed with the following endpoints:

```
@app.route("/items")      # Returns items from RDS MySQL
@app.route("/cpu")        # CPU-intensive endpoint for load testing
@app.route("/health")     # Health check for ALB target group
```

The application runs under Gunicorn with 4 workers, managed by systemd for automatic restart capability.

5 Auto-Scaling Configuration

5.1 Scaling Policies

Policy	Scale Out	Scale In
Trigger	CPU > 50%	CPU < 30%
Duration	2 consecutive minutes	2 consecutive minutes
Action	Add 1 instance	Remove 1 instance
Cooldown	120 seconds	120 seconds

Table 2: Auto-scaling policy configuration

5.2 CloudWatch Alarms

Two CloudWatch alarms were configured:

- `capstone-high-cpu`: Triggers scale-out when average CPU > 50%
- `capstone-low-cpu`: Triggers scale-in when average CPU < 30%

6 Load Testing and Results

6.1 Load Test Script

A Python-based load testing script was developed using concurrent.futures for parallel request execution:

```
# load_test.py - Key parameters
URL = "http://<ALB-DNS>/cpu"
workers = 20
total_requests = 50000
```

6.2 Test Execution

The load test was executed against the /cpu endpoint, which performs SHA-256 hash computations to generate CPU load. Initial tests against the /items endpoint revealed that database queries are I/O-bound rather than CPU-bound, necessitating the CPU-intensive endpoint.

6.3 Results

Metric	Value
Total Requests	50,000
Workers	20
Requests/sec	19-24
Success Rate	>95%
Scaling Result	
Initial Capacity	2 instances
Final Capacity	4 instances
Scale Events	2 (2 to 3, then 3 to 4)
Alarm State	ALARM

Table 3: Load test results

6.4 Scaling Timeline

1. **07:43:22 UTC**: CloudWatch alarm transitioned to ALARM state
2. **07:43:28 UTC**: ASG launched instance 3 (capacity 2 to 3)
3. **07:46:22 UTC**: Second scale-out triggered
4. **07:46:34 UTC**: ASG launched instance 4 (capacity 3 to 4)

7 Screenshots and Evidence

7.1 Frontend Application

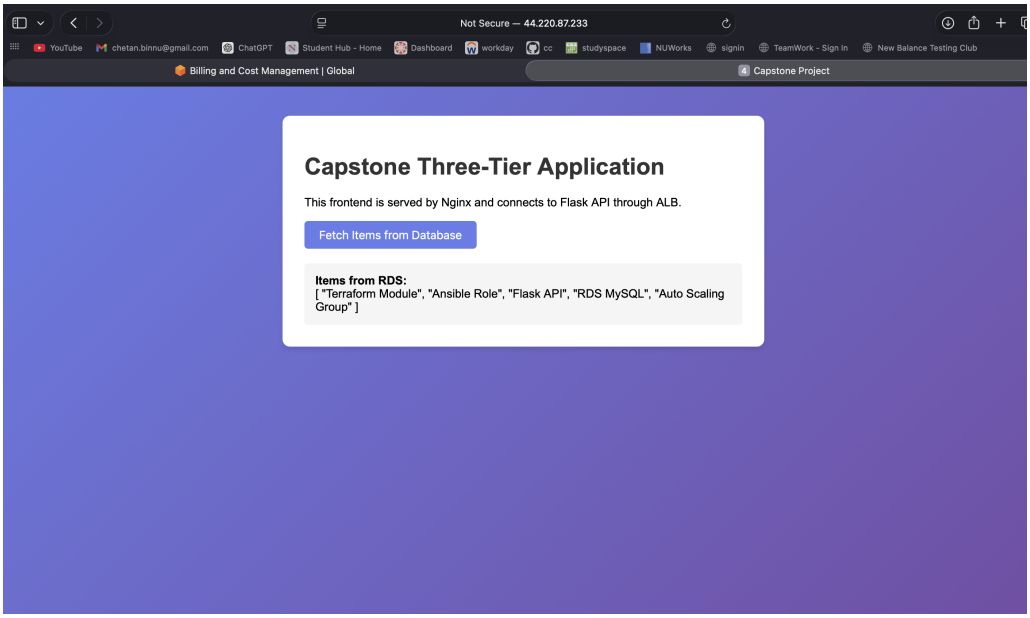


Figure 1: Nginx Frontend displaying items fetched from RDS database

7.2 Auto-Scaling Group Console

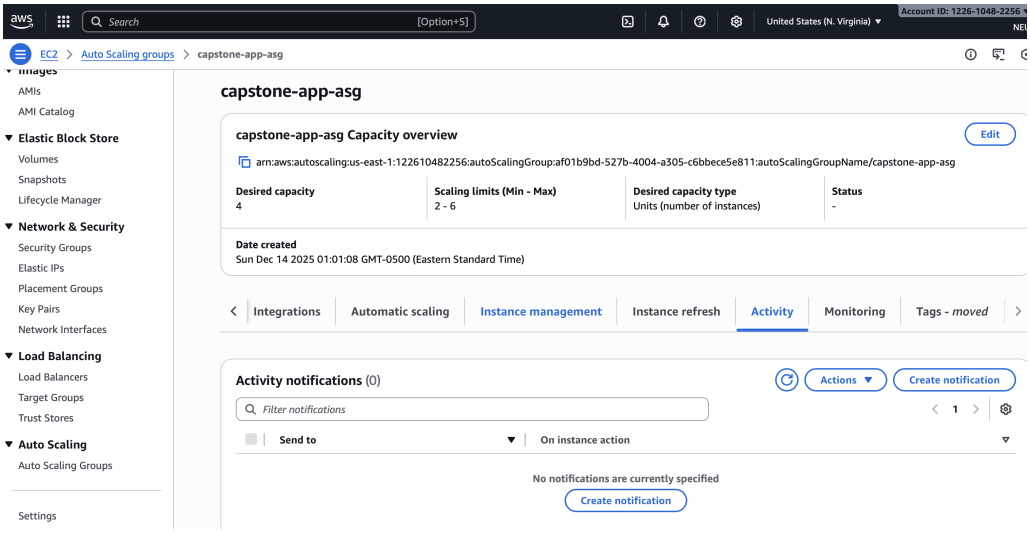


Figure 2: Auto Scaling Group showing scaled capacity of 4 instances

7.3 EC2 Instances

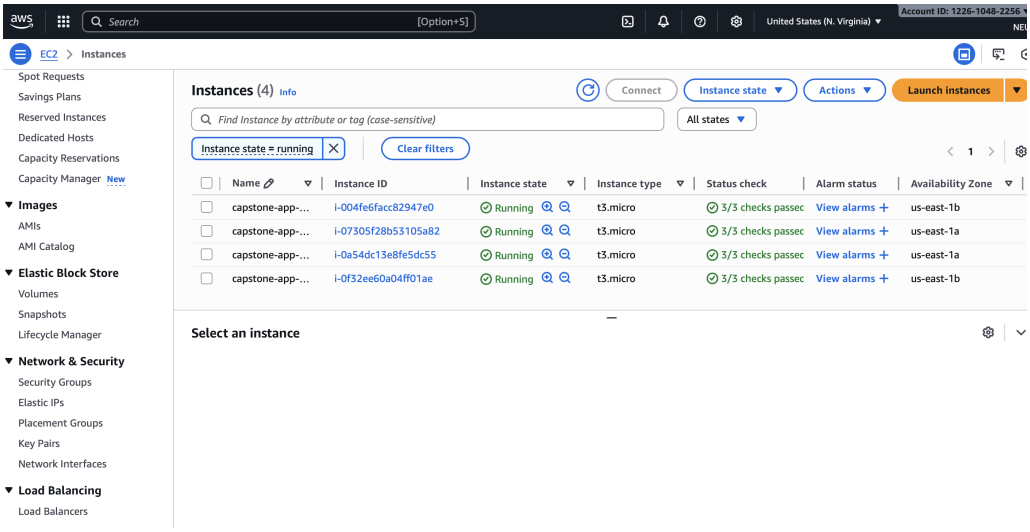


Figure 3: EC2 Console showing 4 running Flask application instances

7.4 Target Group Health

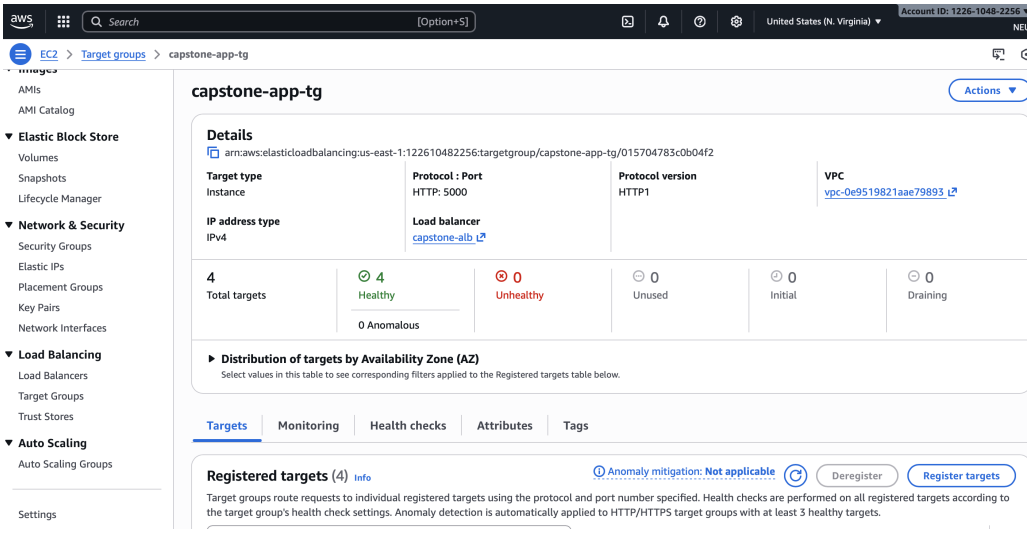


Figure 4: ALB Target Group showing 4 healthy registered targets

7.5 Load Test Execution

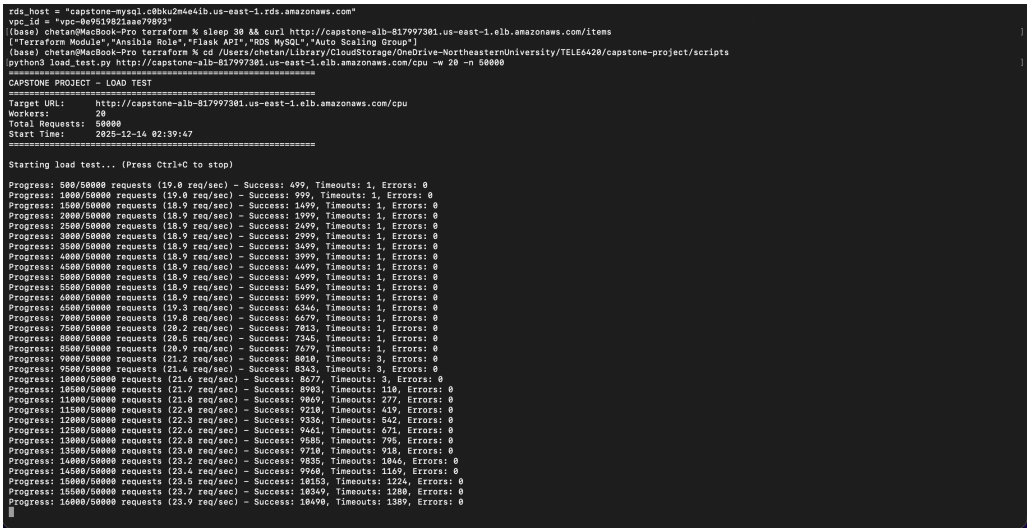


Figure 5: Load test script execution showing requests per second

7.6 Scaling Activities

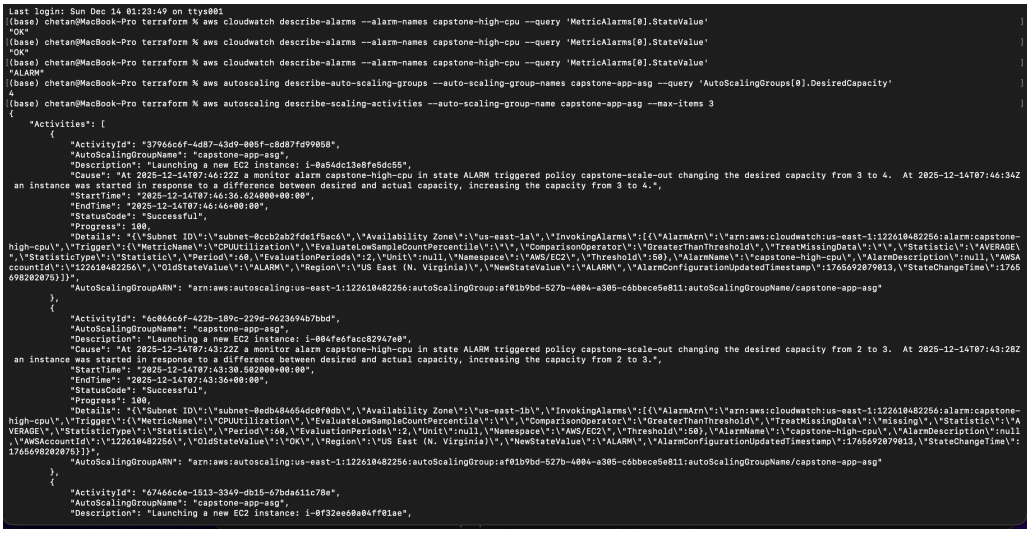


Figure 6: Terminal showing ALARM state and ASG scaling from 2 to 4 instances

7.7 API Response

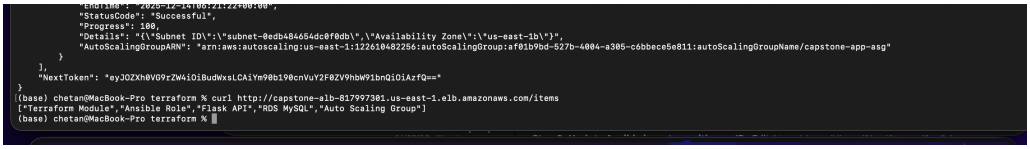


Figure 7: API response showing items retrieved from database via ALB

7.8 Terraform Deployment

```
(base) chetan@MacBook-Pro terraform % cd /Users/chetan/Library/CloudStorage/OneDrive-NortheasternUniversity/TELE6420/capstone-project/terraform
terraform init
terraform plan

Initializing the backend...
Initializing modules...
- frontend in modules/frontend

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v5.100.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
module.networking.data.aws_availability_zones.available: Reading...
module.asg.data.aws_ami.amazon_linux_2: Reading...
module.frontend.data.aws_ami.amazon_linux_2: Reading...
module.networking.data.aws_availability_zones.available: Read complete after 1s [id=us-east-1]
module.asg.data.aws_ami.amazon_linux_2: Read complete after 1s [id=ami-03f9680ef0c07a3d1]
module.frontend.data.aws_ami.amazon_linux_2: Read complete after 1s [id=ami-03f9680ef0c07a3d1]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # module.alb.aws_lb.main will be created
  + resource "aws_lb" "main" {
      + arn                                                          = (known after apply)
      + arn_suffix                                                   = (known after apply)
      + client_keep_alive                                            = 3600
      + desync_mitigation_mode                                       = "defensive"
      + dns_name                                                     = (known after apply)
      + drop_invalid_header_fields                                   = false
      + enable_deletion_protection                                   = false
      + enable_http2                                                 = true
      + enable_tls_version_and_cipher_suite_headers                  = false
      + enable_waf_fail_open                                         = false
      + enable_xff_client_port                                       = false
      + enable_zonal_shift                                           = false
      + enforce_security_group_inbound_rules_on_private_link_traffic = (known after apply)
      + id                                                           = (known after apply)
      + idle_timeout                                                 = 60
```

Figure 8: Terraform plan showing resources to be created

```
Apply complete! Resources: 30 added, 0 changed, 0 destroyed.

Outputs:

alb_dns_name = "capstone-alb-755726009.us-east-1.elb.amazonaws.com"
ansible_vars = {
  "alb_dns_name" = "capstone-alb-755726009.us-east-1.elb.amazonaws.com"
  "db_host" = "capstone-mysql.c0bku2m4e4ib.us-east-1.rds.amazonaws.com"
  "db_name" = "capstone"
  "db_port" = 3306
  "frontend_public_ip" = "44.220.87.233"
}
asg_name = "capstone-app-asg"
frontend_public_dns = "ec2-44-220-87-233.compute-1.amazonaws.com"
frontend_public_ip = "44.220.87.233"
private_subnet_ids = [
  "subnet-09fdfaed5f8fdb541",
  "subnet-0dae0d45a756f2aef",
]
public_subnet_ids = [
  "subnet-06a9ade1d2aa0fc7d",
  "subnet-09bdf39c5fa8c82ab",
]
rds_endpoint = "capstone-mysql.c0bku2m4e4ib.us-east-1.rds.amazonaws.com:3306"
rds_host = "capstone-mysql.c0bku2m4e4ib.us-east-1.rds.amazonaws.com"
vpc_id = "vpc-0f1a2336be5780906"
```

Figure 9: Terraform apply complete - 30 resources created

8 Security Implementation

8.1 Security Groups

Security Group	Inbound	Source	Purpose
frontend-sg	80, 443, 22	0.0.0.0/0	Nginx access
alb-sg	80	0.0.0.0/0	ALB access
app-sg	5000	alb-sg	Flask from ALB only
app-sg	22	0.0.0.0/0	SSH for management
rds-sg	3306	app-sg	DB from app tier only

Table 4: Security group rules

8.2 Network Security

The architecture implements defense in depth by:

- Placing application and database tiers in private subnets with no direct internet access
- Using NAT Gateway for controlled outbound access only
- Restricting database access to the application security group
- Using the ALB as the single entry point to the application tier

9 Reflection

9.1 Challenges Faced

Auto-Scaling Trigger Issues: The initial load tests against the /items endpoint failed to trigger scaling because database queries are I/O-bound rather than CPU-bound. The CloudWatch metrics showed CPU utilization below 1% even under heavy load. This required implementing a dedicated /cpu endpoint with hash computations to generate actual CPU load.

Private Subnet Configuration: Configuring instances in private subnets without public IPs presented connectivity challenges. The solution involved using the Nginx frontend as a bastion host and copying SSH keys for jump connections.

Ansible Module Compatibility: Standard Ansible modules encountered Python deserialization issues on fresh EC2 instances. This was resolved by using raw shell commands for initial configuration.

ALB Health Check Timing: New instances launched by ASG needed time to pass health checks before receiving traffic. The health check grace period of 300 seconds was essential for allowing proper initialization.

9.2 Design Decisions

NAT Gateway vs. Public Subnets: Although NAT Gateway incurs additional costs (approximately $0.045/hour), it was implemented to demonstrate proper production architecture where application servers should not have direct internet exposure.

t3.micro Instance Type: Selected for AWS Free Tier eligibility while providing burstable CPU performance suitable for demonstrating auto-scaling.

Gunicorn with 4 Workers: Chosen to balance concurrency with resource constraints on t3.micro instances.

9.3 Lessons Learned

IaC Benefits: Terraform's declarative approach made infrastructure reproducible. When issues occurred, running terraform destroy and terraform apply provided a clean slate. The modular structure allowed independent testing of components.

Monitoring Importance: CloudWatch metrics were essential for understanding actual system behavior versus assumptions. The CPU utilization data revealed that I/O-bound workloads don't trigger CPU-based scaling policies.

Cost Awareness: Cloud resources accumulate charges quickly. The NAT Gateway alone costs approximately $32/month if left running. Always destroy test infrastructure promptly.

Documentation Value: Maintaining clear documentation of IP addresses, endpoints, and credentials was crucial for multi-session work on this project.

9.4 Future Improvements

- Implement HTTPS with ACM certificates
- Add RDS Multi-AZ for database high availability
- Configure Application Auto Scaling for RDS
- Implement CI/CD pipeline for automated deployments
- Add detailed application logging with CloudWatch Logs
- Implement request-based scaling in addition to CPU-based

10 Conclusion

This capstone project successfully demonstrated the deployment of a three-tier web application using modern IaC practices. The combination of Terraform for infrastructure provisioning and Ansible for configuration management provided a repeatable, version-controlled approach to infrastructure deployment.

The auto-scaling functionality was validated through load testing, with the system successfully scaling from 2 to 4 instances in response to increased CPU utilization. The architecture follows AWS best practices with proper network segmentation, security group isolation, and managed database services.

The project reinforced the importance of understanding workload characteristics when designing scaling policies and highlighted the practical benefits of Infrastructure as Code for managing cloud resources efficiently.

11 Appendix: Repository Structure

```
capstone-project/
    terraform/
        main.tf
        variables.tf
        outputs.tf
        modules/
            networking/
            alb/
            asg/
            rds/
            frontend/
    ansible/
```

```
    ansible.cfg
    inventory/
    playbooks/
    roles/
scripts/
    load_test.py
README.md
```