Capstone Project 1 Report

Date: June 25, 2025

Group: 4

Members:

- Anona Solomon Azure
- Clement Owusu Bempah
- Edmund Asamoah Adomako
- Marie-Pearl Otoo
- Fahad Mohammed Gibrine

Contents

1. Executive Summary

This report details Group 4's capstone project, a comprehensive endeavor to design, implement, and validate a robust, secure, and scalable cloud infrastructure using Amazon Web Services (AWS). The project was divided into three interconnected phases: establishing a Virtual Private Cloud (VPC) with secure networking, deploying auto-scaling and load balancing for high availability, and integrating a NoSQL database (DynamoDB) for data logging.

In Part One, we successfully architected a highly available VPC with public and private subnets spanning multiple availability zones, configured precise routing, and implemented multi-layered security with Security Groups, Network ACLs, and NAT Gateways. This foundational work ensured network isolation and controlled access. This section is further clarified by a dedicated architecture diagram.

Part Two focused on enhancing application resilience. We deployed Elastic Load Balancers (ELBs) to distribute incoming traffic efficiently and configured Auto Scaling Groups (ASGs) to automatically adjust compute capacity based on demand, demonstrated through CPU utilization stress tests and detailed CloudWatch monitoring. This phase confirmed the infrastructure's ability to maintain performance under varying loads, visually represented in its own architecture diagram.

Finally, Part Three involved integrating a data logging solution. We leveraged Amazon DynamoDB for its serverless and scalable NoSQL capabilities, preparing an EC2 instance with Python and Boto3, and securing access via IAM roles. We developed and executed a Python script to insert student records into DynamoDB, validating data persistence and accuracy directly from the AWS console. The architecture for this integration is also detailed in a dedicated diagram.

Throughout the project, we encountered and overcame challenges related to metric selection, instance stressing, and script execution, deepening our understanding of cloud best practices, security principles, and the intricacies of AWS service integration. This capstone project not only confirmed our theoretical knowledge but also provided invaluable practical experience in building enterprise-grade cloud solutions. The successful completion of all phases demonstrates our collective ability to design, deploy, and manage a sophisticated cloud environment capable of meeting real-world application demands.

2. Introduction to the Capstone Project

2.1 Project Overview

The Capstone Project for Group 4 aimed to construct a resilient, secure, and highly available cloud-based infrastructure on Amazon Web Services (AWS). This multi-faceted project was meticulously planned and executed over a three-day period, demonstrating a practical application of core cloud computing principles. The project was segmented into three distinct but interdependent parts, each building upon the preceding one to culminate in a fully functional, scalable, and secure system.

The first part focused on establishing the fundamental network architecture through the creation of a Virtual Private Cloud (VPC). This involved defining network segmentation using public and private subnets across multiple availability zones, alongside configuring essential network components like Internet Gateways, NAT Gateways, Security Groups, Network Access Control Lists (NACLs), and route tables. The objective here was to create a secure, isolated network environment that provided controlled access to resources.

The second part advanced the infrastructure by implementing mechanisms for high availability and elasticity. This phase involved deploying Elastic Load Balancers (ELBs) to distribute incoming application traffic efficiently and configuring Auto Scaling Groups (ASGs) to dynamically adjust computing capacity in response to varying workloads. The success of this

phase was validated through stress testing and comprehensive monitoring using AWS CloudWatch, ensuring the application's performance and continuous operation under fluctuating demand.

The final part integrated a backend data storage solution using Amazon DynamoDB, a highly scalable NoSQL database service. This phase involved setting up a DynamoDB table, provisioning an EC2 instance, configuring appropriate Identity and Access Management (IAM) roles for secure database access, and developing a Python application using the Boto3 SDK to interact with the database (inserting and retrieving data). This demonstrated a complete end-to-end data workflow within the secure cloud environment.

2.2 Project Objectives

The overarching objectives of this capstone project were to:

- **Design and Deploy a Secure and Scalable VPC Architecture:** Establish a robust and segmented network foundation that ensures isolation, controlled traffic flow, and future scalability. This included defining CIDR blocks, creating subnets (public and private), and configuring network gateways and security layers.
- **Implement High Availability and Elasticity:** Ensure continuous application uptime and optimal performance by distributing traffic across multiple instances and automatically adjusting compute resources based on demand. This involved setting up Elastic Load Balancers, Target Groups, Auto Scaling Groups, and dynamic scaling policies.
- **Integrate a Robust Database Solution:** Implement a scalable and efficient data storage mechanism capable of handling application data securely. This specifically focused on utilizing Amazon DynamoDB and demonstrating programmatic interaction from an EC2 instance.
- **Demonstrate Cloud Security Best Practices:** Apply principles of least privilege, network segmentation, and secure access management through the judicious use of Security Groups, NACLs, IAM roles, and key pairs.
- **Utilize AWS Monitoring and Logging Tools:** Implement comprehensive monitoring and logging solutions (primarily AWS CloudWatch) to track application health, performance, and infrastructure activity, enabling proactive issue detection and resolution.
- **Gain Practical Experience:** Provide a hands-on learning experience in deploying and managing a multi-tier cloud application, fostering problem-solving skills and teamwork in a real-world scenario.

2.3 Team Collaboration and Learning

The successful completion of this project was a testament to effective team collaboration and a shared learning experience. Each member contributed significantly, sharing ideas, troubleshooting challenges collectively, and leveraging individual strengths. The iterative nature of the project allowed for continuous feedback and refinement of the architecture and implementation strategies. This collaborative environment not only facilitated the technical achievements but also enhanced our collective understanding of complex cloud concepts and operational best practices. The project reinforced the importance of clear communication, task delegation, and mutual support in achieving ambitious technical goals.

3. Part One: Design and Implementation of a Secure and Scalable VPC

3.1 Introduction to Virtual Private Cloud (VPC)

A Virtual Private Cloud (VPC) is a fundamental building block for any robust cloud infrastructure on AWS. It provides an isolated, private section of the AWS cloud where users can launch AWS resources in a virtual network that they define. This isolation offers a significant security advantage, as resources within a VPC are logically separated from other AWS customers' virtual networks. Furthermore, VPCs allow for granular control over network configuration, including IP address ranges, subnets, route tables, and network gateways. This level of control is crucial for designing a secure and scalable environment that aligns with specific application requirements and compliance standards.

For this capstone project, the establishment of a well-designed VPC was the critical first step, laying the groundwork for all subsequent deployments. Our design emphasized high availability, fault tolerance, and multi-layered security, ensuring that our application infrastructure would be resilient and protected from unauthorized access.

3.2 Key Tasks and Methodologies

3.2.1 VPC Creation

The initial task involved creating a new VPC using the AWS Management Console. We selected an IPv4 CIDR block of 10.0.0.0/16. This particular CIDR block provides a large address space,

allowing for 2^{16} = 65,536 private IP addresses. This substantial range offers ample room for future expansion and segmentation into smaller subnets without IP address exhaustion, a critical consideration for scalability.

Upon creation, the VPC automatically came with a default route table. While this default route table serves a basic purpose, our subsequent steps involved creating custom route tables to precisely control traffic flow within our segmented network. The 10.0.0.0/16 CIDR block adheres to RFC 1918 standards for private IP addressing, ensuring that our internal network traffic remains non-routable over the public internet, thereby enhancing security. The use of the AWS console for VPC creation allowed for a visual and straightforward initial setup, confirming that the basic network parameters were correctly configured from the outset.

3.2.2 Subnet Creation for High Availability

To ensure high availability and fault tolerance, a cornerstone of resilient cloud architecture, we strategically divided our VPC's IP address space into multiple subnets. Specifically, we created two public subnets and two private subnets, meticulously distributed across distinct Availability Zones (AZs). This cross-AZ deployment is paramount: if one Availability Zone experiences an outage, resources in another AZ can continue to function, preventing service disruption.

- **Public Subnets:** These subnets are designed for resources that need direct internet access, such as web servers or bastion hosts. They are associated with a route table that includes a route to an Internet Gateway. In our setup, these housed components like the public-facing EC2 instances that would serve web traffic.
- **Private Subnets:** These subnets are intended for resources that do not require direct internet access but may need to communicate with the internet for updates or external services (databases, application servers). They are associated with a route table that routes outbound internet traffic through a NAT Gateway (discussed in Section 3.2.5). Keeping sensitive resources in private subnets significantly enhances security by preventing direct inbound internet connectivity.

By using separate Availability Zones for each public and private subnet pair, we built redundancy at the network level. For example, group4-public-subnet1 and group4-private-subnet1 could reside in eu-north-1a, while group4-public-subnet2 and group4-private-subnet2 could be in eu-north-1b. This design ensures that our application remains operational even if one AZ becomes unavailable, aligning with modern cloud resilience best practices.

3.2.3 Internet Gateway (IGW) Configuration

An Internet Gateway (IGW) is a horizontally scaled, redundant, and highly available VPC component that allows communication between instances in your VPC and the internet. Without an IGW, instances within a VPC cannot send or receive traffic from the public internet.
Our methodology involved creating a single Internet Gateway and then attaching it to our Group4-vpc. This attachment is a critical step, as it makes the VPC capable of routing traffic to and from the internet. However, merely attaching an IGW is not enough; specific subnets must have a route in their route table that points to the IGW for actual internet connectivity. We configured our public subnets to utilize this IGW, ensuring that public-facing resources could be reached from the internet and could access internet resources themselves. This setup is fundamental for any publicly accessible application hosted within the VPC.

3.2.4 Security Group Implementation

Security Groups act as virtual firewalls for EC2 instances, controlling inbound and outbound traffic at the instance level. They operate at the network interface level, allowing or denying traffic based on rules defined by the user. This stateless inspection mechanism is a crucial layer of security, complementing Network ACLs.

3.2.4.1 Public Subnet Security Group

For instances residing in the public subnets, we created a dedicated security group named Group4-public-security-group. The rules for this security group were carefully crafted to permit necessary inbound traffic while restricting all other connections, adhering to the principle of least privilege:

- **HTTP (Port 80):** Allowed inbound traffic from 0.0.0.0/0 (any IPv4 address). This rule is essential for hosting web applications accessible via standard HTTP.
- **HTTPS (Port 443):** Allowed inbound traffic from 0.0.0.0/0. This rule enables secure web communication, crucial for encrypted data transfer and user privacy.
- **SSH (Port 22):** Allowed inbound traffic from a specific IP range (the team's public IP address or a restricted range if known, typically 0.0.0.0/0 in a demo environment for ease of access but highly restricted in production). This rule is necessary for administrative access to the EC2 instances.

All other inbound ports were implicitly denied. For outbound rules, typically all outbound traffic is allowed by default from a Security Group unless explicitly restricted. In our case, we maintained a permissive outbound rule to allow instances to retrieve updates, connect to external APIs, or respond to client requests. This granular control at the instance level ensures that only legitimate traffic reaches our public-facing components.

3.2.4.2 Private Subnet Security Group

For instances hosted within the private subnets, a separate security group, Group4-private-securitygroup, was established. The design philosophy here was stricter, as these instances (databases, internal application servers) should not be directly exposed to the public internet.

- **Inbound Rules:** We configured **zero inbound rules from the internet (0.0.0.0/0)** for this security group. Inbound access would typically be restricted to specific Security Group IDs (allowing traffic from the Public Subnet's Security Group on specific application ports) or internal IP ranges. The goal was to ensure that only authorized services or instances within the VPC could initiate connections to these private resources.
- **Outbound Rules:** While internal communication might be permissive, outbound traffic from instances in private subnets often needs access to the internet for software updates, patches, or external API calls. This outbound access is routed through a NAT Gateway, not directly through an Internet Gateway. Our outbound rule for this security group likely permitted all outbound traffic to 0.0.0.0/0, allowing instances in the private subnet to initiate connections to the internet via the NAT Gateway.

By carefully segregating security groups and their rules based on subnet type and instance function, we established a robust defense-in-depth strategy, minimizing the attack surface for sensitive internal resources.

3.2.5 NAT Gateway Setup for Private Subnet Connectivity

Instances deployed in private subnets, by definition, do not have direct internet access. However, they often require outbound internet connectivity for various purposes, such as fetching software updates, accessing public repositories, or connecting to external APIs (e.g., payment gateways, external services). A NAT Gateway (Network Access Translation Gateway) provides this crucial capability.

A NAT Gateway is deployed to route traffic from instances in **private subnets** to the internet.

When traffic from a private instance passes through the NAT Gateway, the NAT Gateway replaces the private IP address of the instance with its own public IP address (an Elastic IP). This allows the private instance to communicate with the internet while preventing unsolicited inbound connections directly from the internet.
Our implementation involved creating a NAT Gateway in one of our public subnets. An Elastic IP address was associated with this NAT Gateway to provide a static public IP. Subsequently, we updated the route tables associated with our private subnets to route all internet-bound traffic (i.e., 0.0.0.0/0) through this NAT Gateway. This setup ensures that our database instances or application servers residing in the private subnets can securely access external resources without compromising their isolation from direct inbound internet threats. This architecture is a standard and highly recommended practice for secure multi-tier applications on AWS.

3.2.6 Network Access Control Lists (NACLs) for Enhanced Security

Network Access Control Lists (NACLs) offer an optional, additional layer of security for your VPC, operating at the subnet level. Unlike Security Groups, which are stateful (meaning that if you allow an inbound rule, the outbound response is automatically allowed), NACLs are stateless. This means that for any traffic to flow, both inbound and outbound rules must be explicitly defined and permit the traffic. This stateless nature provides a very granular level of control, though it requires more meticulous configuration.
Our project involved associating NACLs with the subnets we created. The primary purpose of NACLs in our design was to provide a broad, coarse-grained filter for traffic entering and leaving our subnets, acting as a "bouncer" at the subnet perimeter before traffic reaches the more granular Security Group rules at the instance level.
Key characteristics and benefits of our NACL implementation:

- **Subnet-level Control:** NACLs control traffic for all instances within a specific subnet.
- **Stateless Filtering:** We had to explicitly define both inbound and outbound rules for all desired traffic flows. For example, if we allowed HTTP inbound, we also needed a corresponding outbound rule to allow the response.
- **Rule Ordering:** NACL rules are evaluated in order, from the lowest numbered rule to the highest. As soon as a rule matches traffic, it's applied, and no further rules are evaluated for that traffic. This allowed us to prioritize specific ALLOW or DENY rules.
- **Default DENY Rule:** Similar to Security Groups, a default deny rule exists for traffic not

explicitly allowed.

- **Complementary to Security Groups:** While Security Groups are typically sufficient for most filtering needs, NACLs provide an extra layer of defense, particularly useful for implementing blanket denies for certain IP ranges or ports across an entire subnet.

By leveraging NACLs in conjunction with Security Groups, we established a defense-in-depth security posture, ensuring that traffic was filtered at both the subnet and instance levels, thereby significantly hardening our network.

3.2.7 Route Tables and Subnet Associations

Route tables are essential components of a VPC, as they contain a set of rules, called routes, that determine where network traffic from your subnets or gateways is directed. Every subnet in a VPC must be associated with a route table.

In our VPC design, we created two primary types of custom route tables:

- **Public Route Table (Group4-public-rtb):**
    - This route table was explicitly associated with our public subnets (group4-public-subnet1, group4-public-subnet2).
    - It contained a route for all internet-bound traffic (0.0.0.0/0) pointing to the Internet Gateway (group4-igw). This rule enables instances in public subnets to communicate directly with the internet.
    - It also implicitly contained a route for local traffic within the VPC CIDR (10.0.0.0/16) to communicate within the VPC itself.
- **Private Route Table (Group4-private-rtb):**
    - This route table was associated with our private subnets (group4-private-subnet1, group4-private-subnet2).
    - It contained a route for all internet-bound traffic (0.0.0.0/0) pointing to the NAT Gateway (Group4-Natgw). This configuration ensures that instances in private subnets can initiate outbound connections to the internet (for updates) but cannot receive unsolicited inbound connections from the internet.
    - Similar to the public route table, it included a local route for intra-VPC communication.

The process of "edit subnet associations" was critical for linking our newly created subnets to their respective custom route tables. This explicit association ensures that traffic originating from or destined for a specific subnet follows the intended routing paths, maintaining network

segmentation and security. Proper route table configuration is foundational for ensuring correct traffic flow and enforcing network policies within the VPC.

3.3 Architectural Decisions and Rationale

The architectural decisions made during Part One were driven by principles of security, scalability, and high availability:

- **Multi-AZ Deployment:** Deploying subnets across multiple Availability Zones was a non-negotiable decision to ensure fault tolerance. This protects against single points of failure at the AZ level, which is crucial for business continuity.
- **Public and Private Subnet Segmentation:** This is a standard security best practice. Placing publicly accessible resources (Load Balancers, web servers) in public subnets and sensitive backend resources (databases, application servers) in private subnets minimizes the attack surface and enforces a clear security boundary.
- **Separate Security Groups:** Instead of using a single, monolithic security group, creating distinct security groups for public and private subnets, each with tailored rules, provides granular control and adheres to the principle of least privilege. This makes it easier to manage and audit network access.
- **NAT Gateway for Outbound Private Connectivity:** Utilizing a NAT Gateway is the secure and recommended way to enable internet access for private instances. It allows necessary outbound communication without exposing internal resources to inbound threats, maintaining their isolation.
- **Layered Security (Security Groups + NACLs):** While Security Groups are often sufficient, adding NACLs provides an additional, stateless layer of control at the subnet boundary. This defense-in-depth strategy enhances overall network security by allowing for broader traffic filtering before reaching individual instances.
- **Custom Route Tables:** Explicitly defining custom route tables for public and private subnets, instead of relying solely on the default VPC route table, ensures precise control over traffic routing, directing it appropriately through the Internet Gateway or NAT Gateway.

3.4 Challenges and Solutions in VPC Design

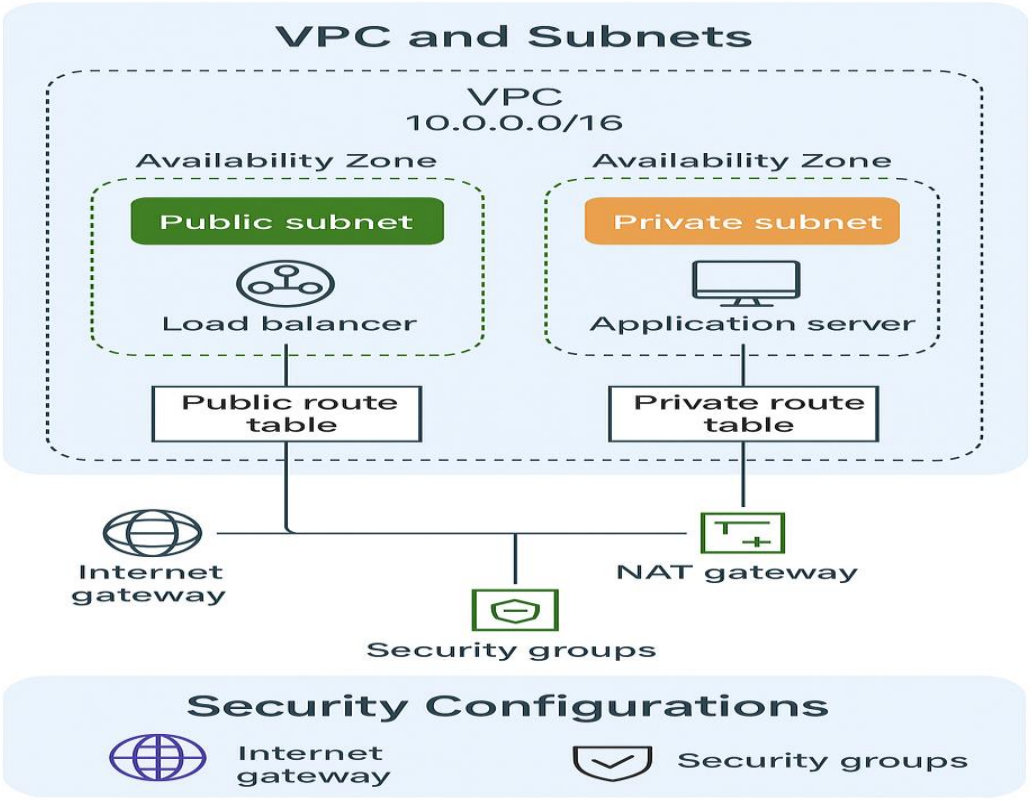
During the VPC design and implementation phase, several common challenges were anticipated and addressed:

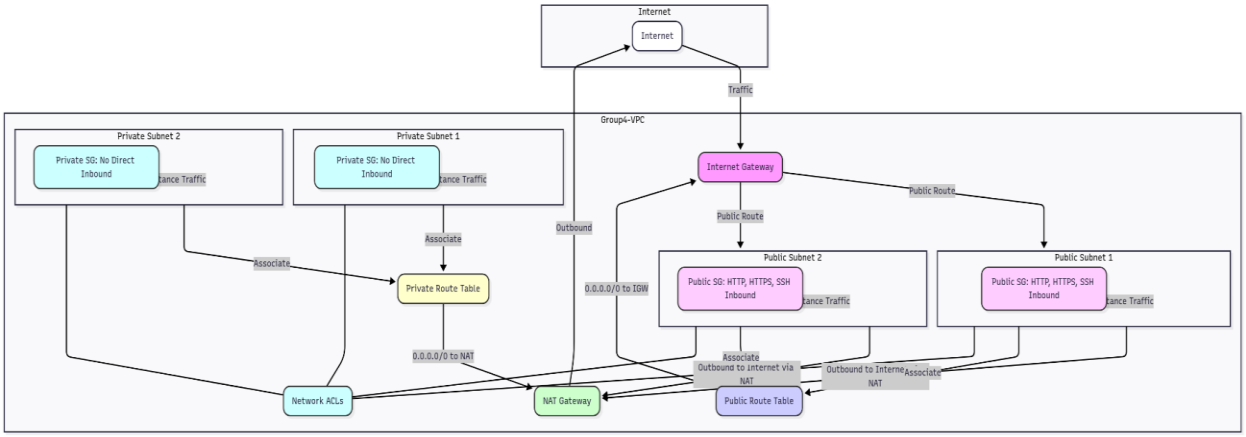
- **IP Address Overlap/Exhaustion:** Carefully selecting the initial /16 CIDR block and then subnetting (/24 or similar for individual subnets) mitigated the risk of running out of IP addresses prematurely, ensuring scalability. Planning for future growth was key.
- **Incorrect Route Table Associations:** A common pitfall is mis associating subnets with the wrong route table, leading to connectivity issues. Our meticulous double-checking of subnet-to-route table links via the "Edit subnet associations" interface ensured correct traffic flow.
- **Misconfigured Security Group Rules:** Overly permissive or restrictive security group rules can either expose resources or prevent legitimate traffic. We adopted a "start with least privilege and open as needed" approach, verifying connectivity with ssh and curl after each rule modification to ensure functionality without compromising security.
- **NAT Gateway Placement and Routing:** Incorrectly placing the NAT Gateway in a private subnet or failing to update the private route table to point to the NAT Gateway were potential issues. We ensured the NAT Gateway was always in a public subnet and validated the private route table's 0.0.0.0/0 route was correctly set to the NAT Gateway target.
- **NACL Statelessness Complexity:** Understanding and correctly implementing both inbound and outbound rules for NACLs, particularly their stateless nature, required careful attention. We relied on a structured approach to define pairs of rules (inbound request, outbound response) for all expected traffic.

By adhering to AWS best practices, performing systematic verification steps after each configuration, and maintaining a clear understanding of each service's function, we successfully navigated these challenges, establishing a robust and secure network foundation for our application.

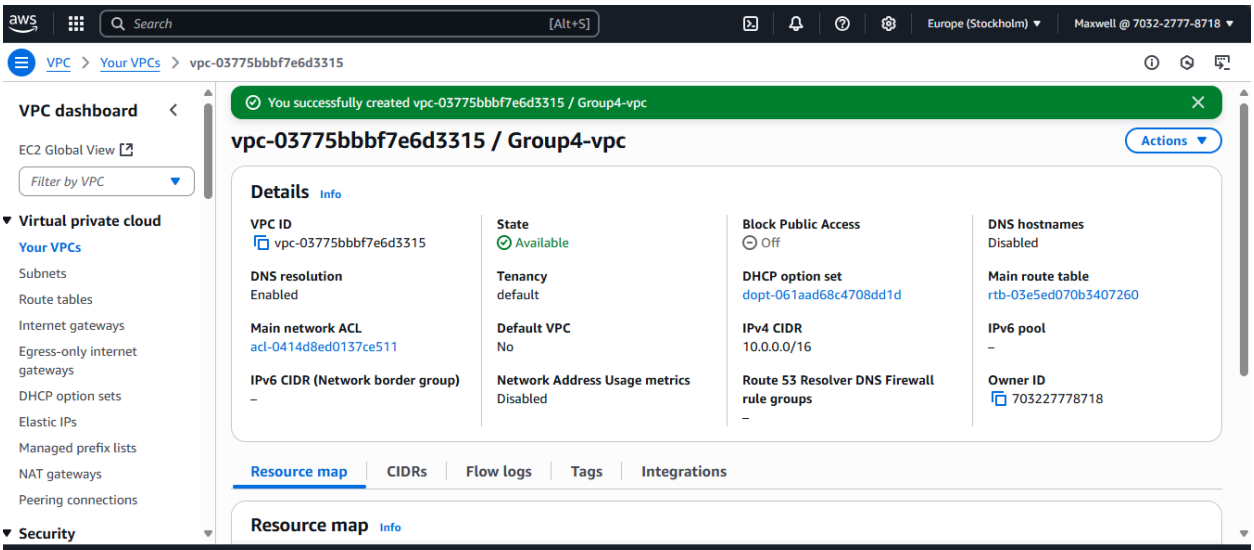
3.5 Architecture Diagram: VPC Network Foundation

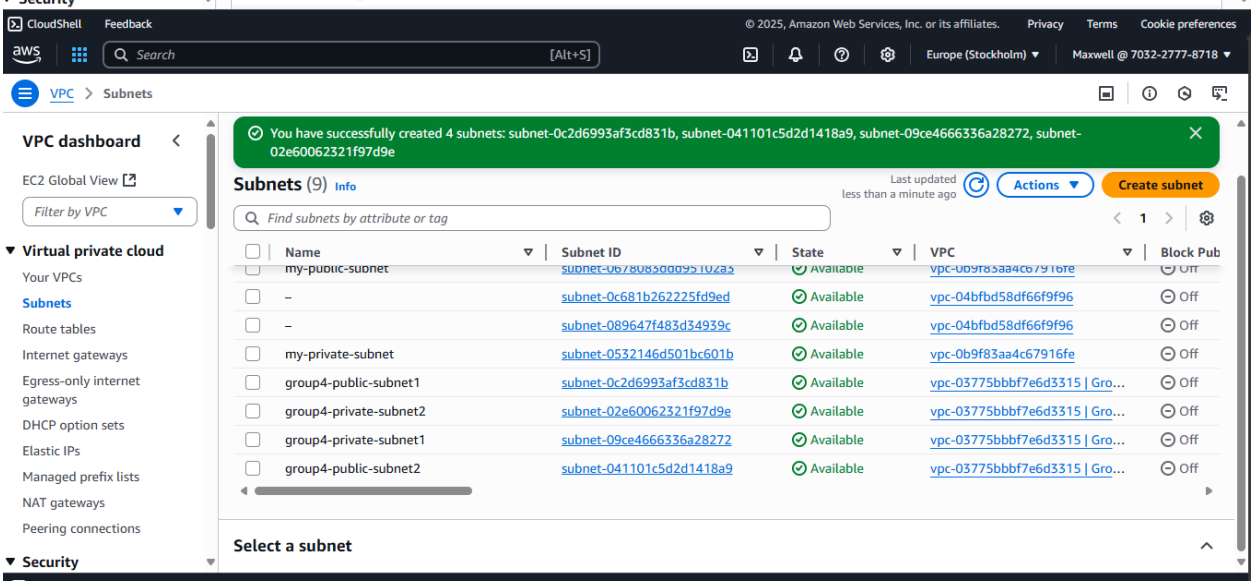
The following diagrams illustrate the foundational VPC network architecture established in Part One, highlighting the segmentation, gateways, and security layers.

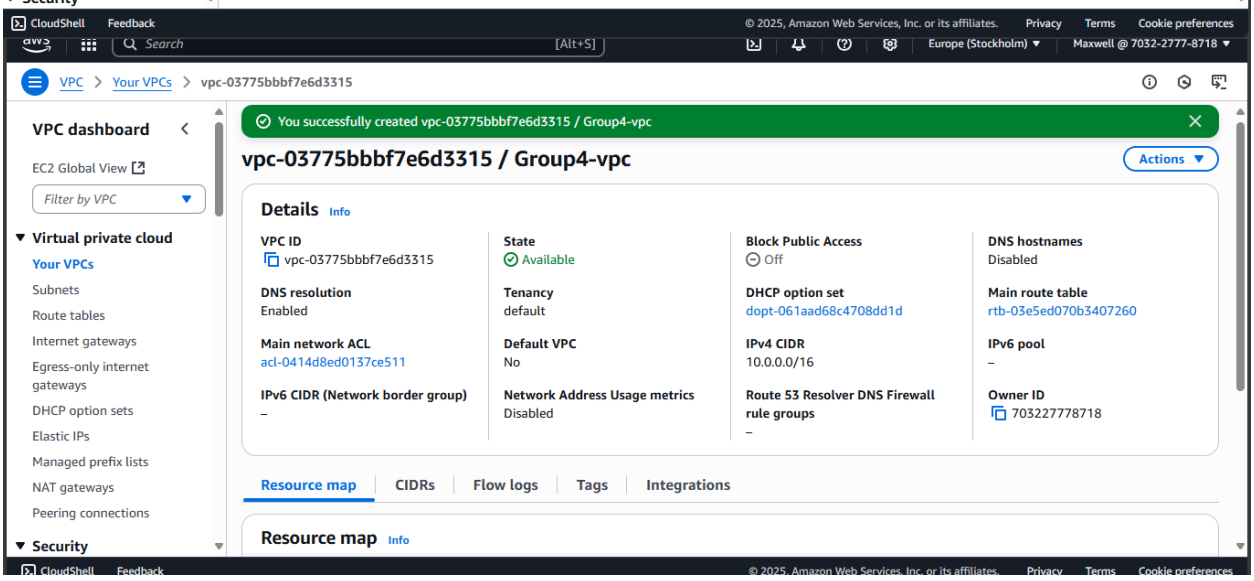


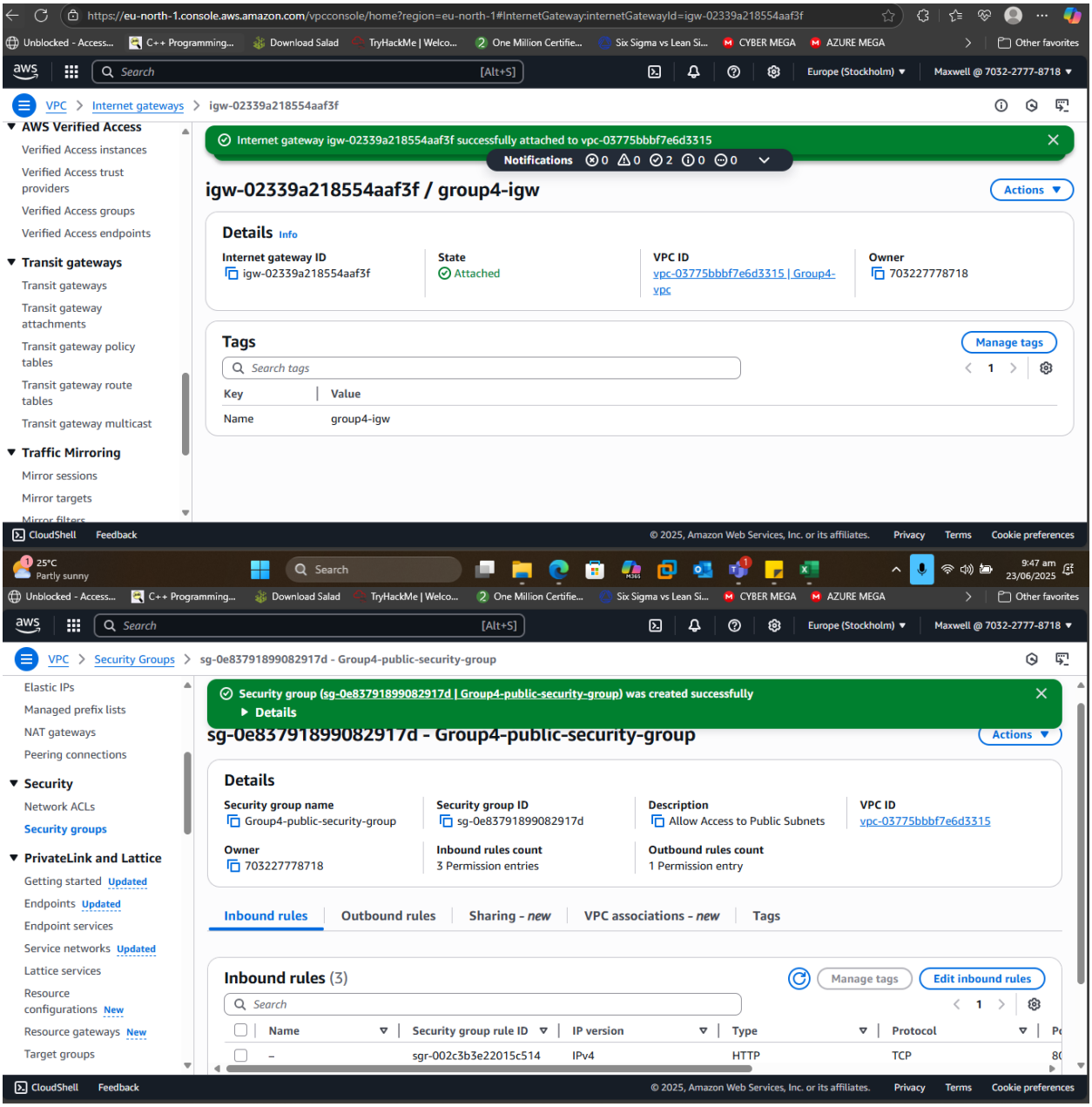


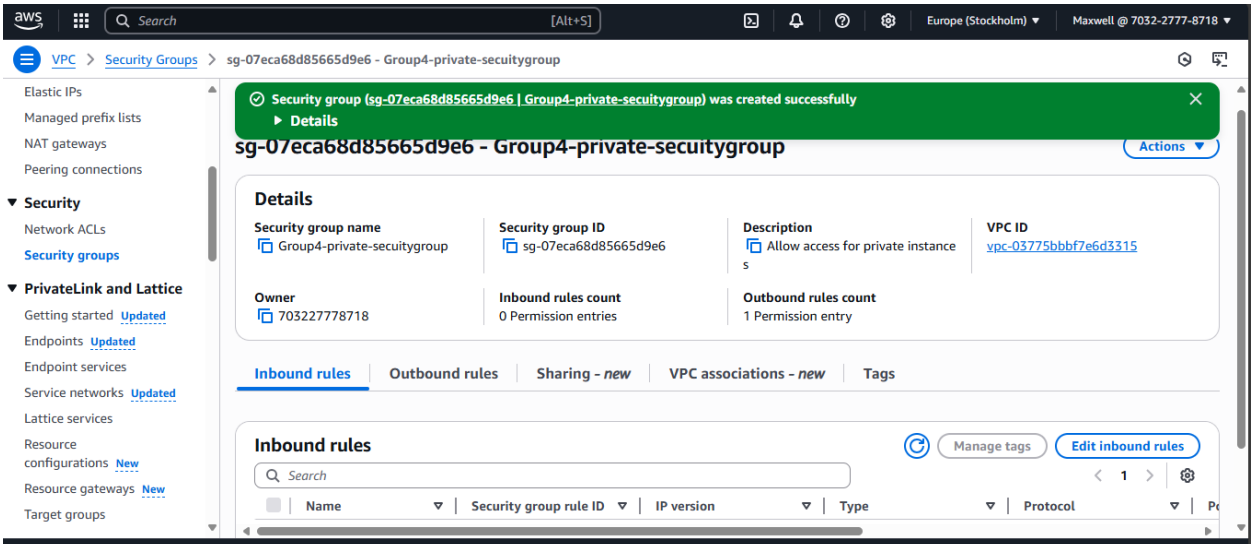
3.6 Method: VPC Network Foundation

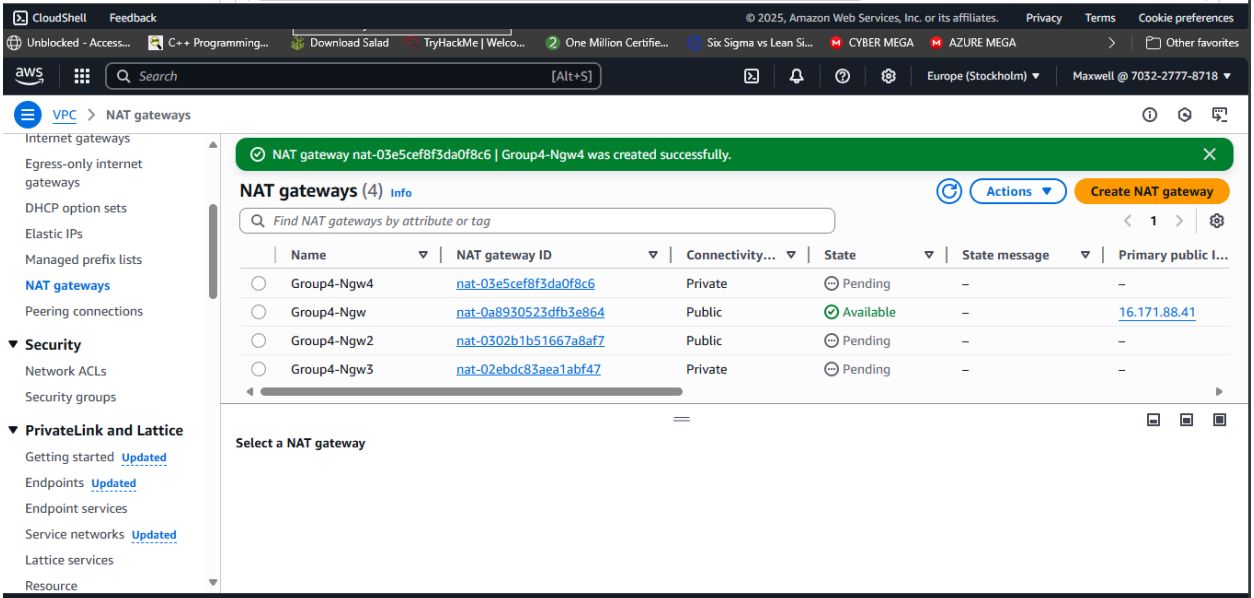

Search [Alt+S]
Europe (Stockholm)
Maxwell @ 7032-2777-8718
VPC > Your VPCs > vpc-03775bbbf7e6d3315
VPC dashboard
EC2 Global View
Filter by VPC
Virtual private cloud
Your VPCs
Subnets
Route tables
Internet gateways
Egress-only internet gateways
DHCP option sets
Elastic IPs
Managed prefix lists
NAT gateways
Peering connections
Security
You successfully created vpc-03775bbbf7e6d3315 / Group4-vpc
vpc-03775bbbf7e6d3315 / Group4-vpc
Actions
Details Info
VPC ID
vpc-03775bbbf7e6d3315
State
Available
Block Public Access
Off
DNS hostnames
Disabled
DNS resolution
Enabled
Tenancy
default
DHCP option set
dopt-061aad68c4708dd1d
Main route table
rtb-03e5ed070b3407260
Main network ACL
acl-0414d8ed0137ce511
Default VPC
No
IPv4 CIDR
10.0.0.0/16
IPv6 pool
–
IPv6 CIDR (Network border group)
–
Network Address Usage metrics
Disabled
Route 53 Resolver DNS Firewall rule groups
–
Owner ID
703227778718
Resource map
CIDRs
Flow logs
Tags
Integrations
Resource map Info

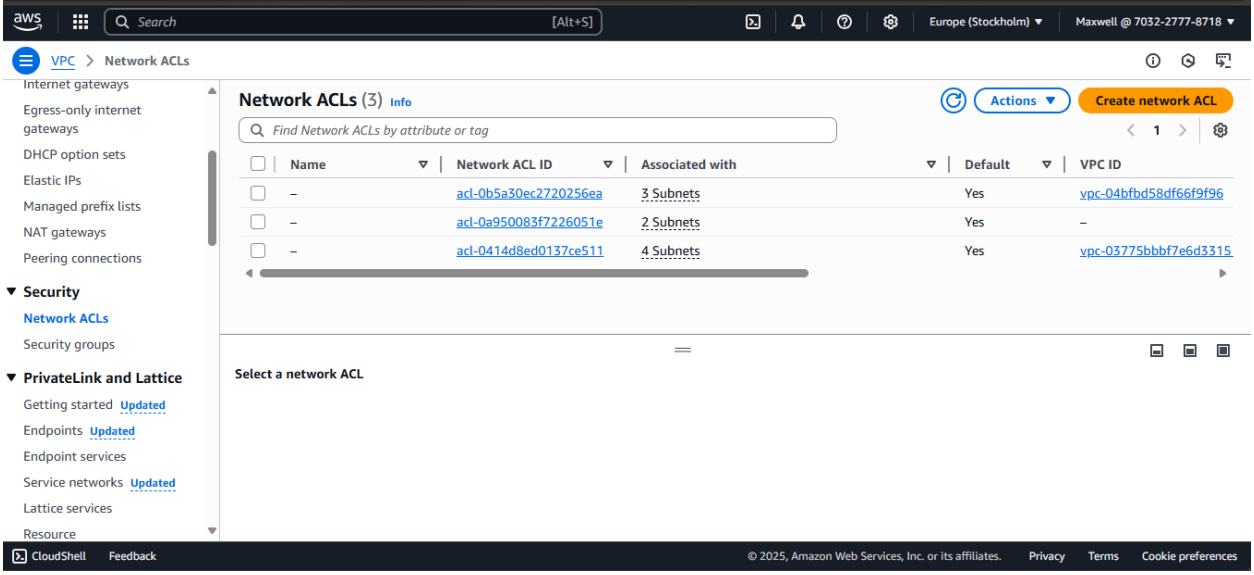

CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences
Search [Alt+S]
Europe (Stockholm)
Maxwell @ 7032-2777-8718
VPC > Subnets
VPC dashboard
EC2 Global View
Filter by VPC
Virtual private cloud
Your VPCs
Subnets
Route tables
Internet gateways
Egress-only internet gateways
DHCP option sets
Elastic IPs
Managed prefix lists
NAT gateways
Peering connections
Security
You have successfully created 4 subnets: subnet-0c2d6993af3cd831b, subnet-041101c5d2d1418a9, subnet-09ce4666336a28272, subnet-02e60062321f97d9e
Subnets (9) Info
Last updated less than a minute ago
Actions
Create subnet
Find subnets by attribute or tag
Name
Subnet ID
State
VPC
Block Pub
my-public-subnet
subnet-0c681b262225fd9ed
subnet-089647f483d34939c
my-private-subnet
subnet-0532146d501bc601b
group4-public-subnet1
subnet-0c2d6993af3cd831b
group4-private-subnet2
subnet-02e60062321f97d9e
group4-private-subnet1
subnet-09ce4666336a28272
group4-public-subnet2
subnet-041101c5d2d1418a9
Available
Off
Select a subnet

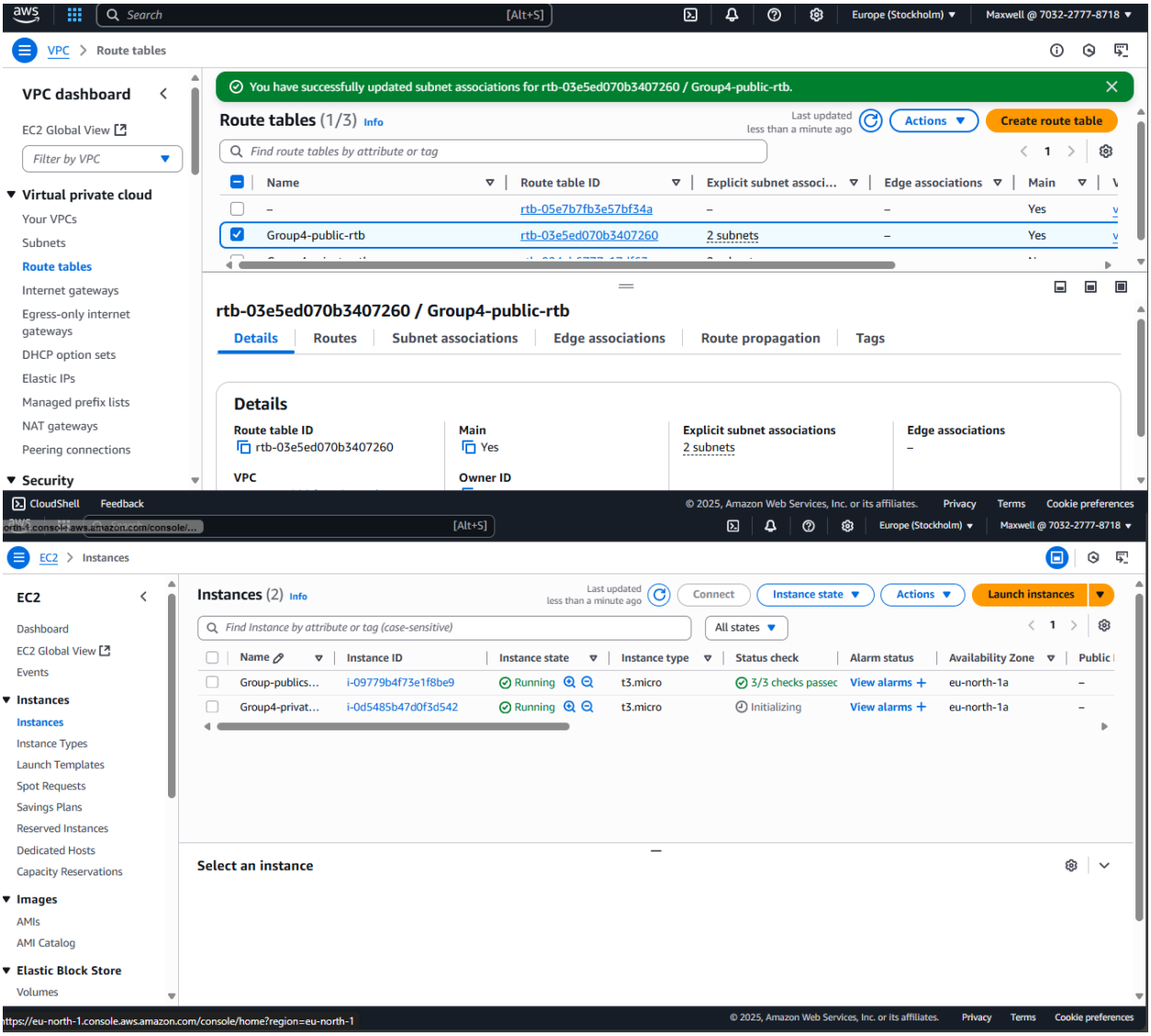

CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences
Search [Alt+S]
Europe (Stockholm)
Maxwell @ 7032-2777-8718
VPC > Your VPCs > vpc-03775bbbf7e6d3315
VPC dashboard
EC2 Global View
Filter by VPC
Virtual private cloud
Your VPCs
Subnets
Route tables
Internet gateways
Egress-only internet gateways
DHCP option sets
Elastic IPs
Managed prefix lists
NAT gateways
Peering connections
Security
You successfully created vpc-03775bbbf7e6d3315 / Group4-vpc
vpc-03775bbbf7e6d3315 / Group4-vpc
Actions
Details Info
VPC ID
vpc-03775bbbf7e6d3315
State
Available
Block Public Access
Off
DNS hostnames
Disabled
DNS resolution
Enabled
Tenancy
default
DHCP option set
dopt-061aad68c4708dd1d
Main route table
rtb-03e5ed070b3407260
Main network ACL
acl-0414d8ed0137ce511
Default VPC
No
IPv4 CIDR
10.0.0.0/16
IPv6 pool
–
IPv6 CIDR (Network border group)
–
Network Address Usage metrics
Disabled
Route 53 Resolver DNS Firewall rule groups
–
Owner ID
703227778718
Resource map
CIDRs
Flow logs
Tags
Integrations
Resource map Info


https://eu-north-1.console.aws.amazon.com/vpcconsole/home?region=eu-north-1#InternetGateway:internetGatewayId=igw-02339a218554aaf3f
VPC > Internet gateways > igw-02339a218554aaf3f
Internet gateway igw-02339a218554aaf3f successfully attached to vpc-03775bbbf7e6d3315
igw-02339a218554aaf3f / group4-igw
Actions
Details Info
Internet gateway ID
igw-02339a218554aaf3f
State
Attached
VPC ID
vpc-03775bbbf7e6d3315 | Group4-vpc
Owner
703227778718
Tags
Manage tags
Search tags
Key
Value
Name
group4-igw
AWS Verified Access
Verified Access instances
Verified Access trust providers
Verified Access groups
Verified Access endpoints
Transit gateways
Transit gateways
Transit gateway attachments
Transit gateway policy tables
Transit gateway route tables
Transit gateway multicast
Traffic Mirroring
Mirror sessions
Mirror targets
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences
VPC > Security Groups > sg-0e83791899082917d - Group4-public-security-group
Security group (sg-0e83791899082917d | Group4-public-security-group) was created successfully
Details
sg-0e83791899082917d - Group4-public-security-group
Actions
Details
Security group name
Group4-public-security-group
Security group ID
sg-0e83791899082917d
Description
Allow Access to Public Subnets
VPC ID
vpc-03775bbbf7e6d3315
Owner
703227778718
Inbound rules count
3 Permission entries
Outbound rules count
1 Permission entry
Inbound rules
Outbound rules
Sharing - new
VPC associations - new
Tags
Inbound rules (3)
Manage tags
Edit inbound rules
Search
Name
Security group rule ID
IP version
Type
Protocol
sgr-002c3b3e22015c514
IPv4
HTTP
TCP
Elastic IPs
Managed prefix lists
NAT gateways
Peering connections
Security
Network ACLs
Security groups
PrivateLink and Lattice
Getting started Updated
Endpoints Updated
Endpoint services
Service networks Updated
Lattice services
Resource configurations New
Resource gateways New
Target groups


Search [Alt+S]
Europe (Stockholm)
Maxwell @ 7032-2777-8718
VPC > Security Groups > sg-07eca68d85665d9e6 - Group4-private-secuitygroup
Elastic IPs
Managed prefix lists
NAT gateways
Peering connections
Security
Network ACLs
Security groups
PrivateLink and Lattice
Getting started Updated
Endpoints Updated
Endpoint services
Service networks Updated
Lattice services
Resource configurations New
Resource gateways New
Target groups
Security group (sg-07eca68d85665d9e6 | Group4-private-secuitygroup) was created successfully
Details
sg-07eca68d85665d9e6 - Group4-private-secuitygroup
Actions
Details
Security group name
Group4-private-secuitygroup
Security group ID
sg-07eca68d85665d9e6
Description
Allow access for private instances
VPC ID
vpc-03775bbbf7e6d3315
Owner
703227778718
Inbound rules count
0 Permission entries
Outbound rules count
1 Permission entry
Inbound rules
Outbound rules
Sharing - new
VPC associations - new
Tags
Inbound rules
Manage tags
Edit inbound rules
Search
Name
Security group rule ID
IP version
Type
Protocol
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


Search [Alt+S]
Europe (Stockholm)
Maxwell @ 7032-2777-8718
VPC > NAT gateways
Internet gateways
Egress-only internet gateways
DHCP option sets
Elastic IPs
Managed prefix lists
NAT gateways
Peering connections
Security
Network ACLs
Security groups
PrivateLink and Lattice
Getting started Updated
Endpoints Updated
Endpoint services
Service networks Updated
Lattice services
Resource
NAT gateway nat-03e5cef8f3da0f8c6 | Group4-Ngw4 was created successfully.
NAT gateways (4) Info
Actions
Create NAT gateway
Find NAT gateways by attribute or tag
Name
NAT gateway ID
Connectivity...
State
State message
Primary public I...
Group4-Ngw4
nat-03e5cef8f3da0f8c6
Private
Pending
Group4-Ngw
nat-0a8930523dfb3e864
Public
Available
16.171.88.41
Group4-Ngw2
nat-0302b1b51667a8af7
Public
Pending
Group4-Ngw3
nat-02ebdc83aea1abf47
Private
Pending
Select a NAT gateway
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


Search [Alt+S]
Europe (Stockholm)
Maxwell @ 7032-2777-8718
VPC > Network ACLs
Internet gateways
Egress-only internet gateways
DHCP option sets
Elastic IPs
Managed prefix lists
NAT gateways
Peering connections
Security
Network ACLs
Security groups
PrivateLink and Lattice
Getting started Updated
Endpoints Updated
Endpoint services
Service networks Updated
Lattice services
Resource
Network ACLs (3) Info
Actions
Create network ACL
Find Network ACLs by attribute or tag
Name
Network ACL ID
Associated with
Default
VPC ID
acl-0b5a30ec2720256ea
3 Subnets
Yes
vpc-04bfbd58df66f9f96
acl-0a950083f7226051e
2 Subnets
Yes
acl-0414d8ed0137ce511
4 Subnets
Yes
vpc-03775bbbf7e6d3315
Select a network ACL
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences



4. Part Two: Implementation of Auto Scaling and Load Balancing for High Availability

4.1 Introduction to High Availability and Elasticity

In the contemporary cloud landscape, where user expectations for continuous service are paramount, "High Availability" (HA) and "Elasticity" are non-negotiable attributes for any production-grade application. High availability refers to the ability of a system to operate continuously without failure for a specified period, typically achieved through redundancy and failover mechanisms. Elasticity, on the other hand, is the system's capacity to automatically scale resources up or down in response to fluctuating demand. Together, HA and elasticity ensure that an application remains responsive and performs optimally, even under

unpredictable traffic loads or in the face of underlying infrastructure failures.
Part Two of our capstone project focused on implementing these critical principles using AWS Elastic Load Balancing (ELB) and Auto Scaling Groups (ASG). This phase aimed to demonstrate an infrastructure that not only intelligently adapts to varying load conditions but also maintains optimal performance and fault tolerance across multiple Availability Zones, building directly upon the secure VPC established in Part One.

4.2 Core Components and Their Roles

To achieve high availability and elasticity, we leveraged several interconnected AWS services:

- **Amazon EC2 (Elastic Compute Cloud):** Provides resizable compute capacity in the cloud. We used EC2 instances as the compute backbone for our application servers.
- **Elastic Load Balancing (ELB):** Automatically distributes incoming application traffic across multiple targets, such as EC2 instances, in multiple Availability Zones. This increases the fault tolerance of your application. We specifically used an Application Load Balancer (ALB).
- **Target Groups:** Used by a load balancer to route requests to one or more registered targets. Each target group is used to route requests to specific registered targets using a specified protocol and port.
- **Auto Scaling Groups (ASG):** Help you ensure that you have the correct number of Amazon EC2 instances available to handle the load for your application. An ASG dynamically scales the number of instances up or down based on defined policies and health checks.
- **Launch Templates:** Define the configuration settings for launching EC2 instances (AMI, instance type, key pair, security groups, user data). ASGs use launch templates to provision new instances.
- **Amazon CloudWatch:** A monitoring and observability service that provides data and actionable insights to monitor your applications, respond to system-wide performance changes, and optimize resource utilization. It was crucial for setting up alarms and tracking scaling activities.

4.3 Key Tasks and Methodologies

4.3.1 Instance Launch and Interconnectivity

Before setting up the load balancer and auto scaling, we ensured the foundational compute resources were in place. We launched two EC2 instances, strategically placing one in each of our public subnets across separate Availability Zones (eu-north-1a and eu-north-1b). These instances were configured with:

- **Application and Operating System Images (AMIs):** We selected a suitable Amazon Linux AMI that would serve as the base for our web application.
- **Key Pair Login:** A secure key pair was generated and associated with the instances for SSH access.
- **Network Settings:** Instances were explicitly placed in their respective public subnets and associated with the Group4-public-security-group (as configured in Part One).
- **User Data (Optional but Recommended):** While not explicitly detailed, in a real-world scenario, user data scripts would be used to install web servers (e.g., Apache, Nginx) and deploy a simple "Hello World" page on instance launch. For demonstration purposes, we confirmed simple HTTP responses like "Hello from Group 4" and "Hello from Group FOUR" to distinguish between instances, implying a basic web server setup.

The existing interconnectivity between the public and private subnets, Internet Gateways, and route tables (as established in Part One) was critical. This confirmed that our EC2 instances in the public subnets could communicate with the internet (to be reached by the load balancer) and potentially with resources in private subnets (databases) if necessary.

4.3.2 Public-facing Instance Connectivity Testing

To validate that our public-facing EC2 instances were correctly configured for external access, we performed a connectivity test. This involved using SSH to connect to the instance deployed in the public subnet.

- **SSH Command:** We executed an ssh command from a local machine, using the private key associated with the instance and its public IPv4 address. For example: ssh -i "group4key.pem" ec2-user@<public-ipv4-address>.
- **Verification:** Successful SSH access confirmed that:
    - The instance was running.
    - The group4key.pem key pair was correctly configured.
    - The Group4-public-security-group was allowing SSH (Port 22) traffic.
    - Network routing through the Internet Gateway was functioning correctly.

The SSH session allowed us to perform initial setup or troubleshoot directly on the instance,

such as installing necessary software or confirming the presence of a web server returning the expected "Hello from Group 4" messages.

4.3.3 Target Group Configuration

A Target Group is a logical grouping of targets (EC2 instances) that a load balancer routes requests to. Each target group is configured to monitor the health of its registered targets.
We created a target group named Group4-target-group. Key configurations included:

- **Protocol and Port:** We specified TCP protocol on Port 80, which is the standard port for HTTP traffic. While an Application Load Balancer typically routes HTTP/HTTPS, setting TCP:80 implies it's listening for generic TCP traffic, which in this context would be interpreted as HTTP.
- **Target Type:** The target type was set to Instance, meaning the load balancer would route traffic to specific EC2 instances.
- **VPC Association:** The target group was explicitly linked to our Group4-vpc.
- **Health Checks:** Configured health checks (HTTP GET on /) to monitor the availability and responsiveness of the registered instances. Only healthy instances receive traffic from the load balancer.

Crucially, we registered our two EC2 instances (one in each public subnet) with this Group4-target-group. The target group's dashboard provided immediate feedback on the health status of these instances, ensuring they were ready to receive traffic.

4.3.4 Elastic Load Balancer (ELB) Setup

For distributing incoming traffic efficiently and ensuring high availability, we set up an Application Load Balancer (ALB), which is a type of ELB. ALBs are best suited for HTTP and HTTPS traffic, offering advanced routing features.
Our Group4-balancer was configured as follows:

- **Load Balancer Type:** Application Load Balancer (ALB).
- **Scheme:** Internet-facing, meaning it accepts requests from clients over the internet.
- **VPC and Subnets:** The load balancer was linked to our Group4-vpc and specifically associated with our **public subnets** across multiple Availability Zones. This multi-AZ deployment is vital for high availability, as the ALB automatically distributes traffic across healthy targets in those AZs.

- **Listeners:** A listener was configured for HTTP traffic on Port 80, routing requests to our Group4-target-group.

Once the ALB was provisioned and its DNS name was available, we tested its connectivity. By navigating to the ALB's DNS name in a web browser, we observed the dynamic distribution of requests between our two instances. Successive refreshes would often show responses like "Hello from Group 4" and "Hello from Group FOUR," demonstrating that the ALB was effectively routing traffic to both underlying EC2 instances. This confirmed that the load balancer was active, healthy, and successfully distributing client requests, ensuring that even if one instance failed, the other could continue serving traffic.

4.3.5 Auto Scaling Group (ASG) Configuration

An Auto Scaling Group (ASG) is a collection of EC2 instances that are treated as a logical grouping for the purposes of automatic scaling and management. ASGs play a pivotal role in maintaining application availability and automatically scaling compute capacity to meet demand. Our Group4-Autoscaling group was configured with the following parameters:

- **Desired Capacity:** 2 instances. This is the number of instances the ASG attempts to maintain.
- **Scaling Limits (Min-Max):** 1-4 instances. This defines the minimum number of instances (ensuring basic availability) and the maximum number of instances (to control costs and resource consumption).
- **Launch Template:** The ASG utilized a pre-configured Launch Template (e.g., group4-launch-template), which specified the AMI ID, instance type (t3.micro), key pair, and the Group4-public-security-group. This ensures that all instances launched by the ASG are consistent in their configuration.

The ASG automatically maintained the desired capacity of two instances. If an instance became unhealthy or was terminated, the ASG would automatically launch a replacement, ensuring the application's continuous availability. This self-healing capability is a cornerstone of resilient cloud design.

4.3.6 Dynamic Scaling Policies and CloudWatch Alarms

To enable true elasticity, we implemented a dynamic scaling policy for our ASG, which automatically adjusts the group's capacity in response to real-time performance metrics.

- **Scaling Policy Type:** We used a Simple scaling policy (though Target tracking scaling is generally recommended for production).
- **Metric and Threshold:** The policy was configured to monitor **CPU Utilization**. If the average CPU utilization across the instances in the ASG exceeded **50%**, the policy would trigger a scale-out event.
- **Action:** When the threshold was crossed, the policy was set to Add 1 capacity unit, meaning one additional EC2 instance would be launched by the ASG.
- **Cooldown Period:** A cooldown period (300 seconds) was set after a scaling activity to prevent rapid, successive scaling actions, allowing the newly launched instances to stabilize and start serving traffic before evaluating further scaling needs.

This scaling policy was directly linked to a **CloudWatch Alarm**. The alarm continuously monitored the aggregate CPU utilization metric for the ASG. When the CPU over threshold alarm state was triggered (i.e., CPU utilization exceeded 60% for a specified period, typically 1 data point within 5 minutes as seen in the image), it would initiate the scale-out action defined in the dynamic scaling policy.

This setup ensures that our application environment can automatically respond to increases in user traffic. When demand rises, and CPU utilization spikes, the ASG will provision new instances, distributing the load and maintaining performance. Conversely, a corresponding scale-in policy (e.g., when CPU utilization drops below a certain threshold) would remove instances, optimizing costs during periods of low demand.

4.3.7 Monitoring and Logging with CloudWatch

AWS CloudWatch is integral to understanding the health, performance, and operational state of cloud resources. It collects and processes raw data into readable, near real-time metrics and logs. We extensively used CloudWatch for both monitoring and logging during Part Two.

4.3.7.1 CPU Utilization Monitoring

After configuring the dynamic scaling policy, we performed stress tests on our EC2 instances to simulate high load and trigger the scaling actions. During these tests, CloudWatch provided invaluable visual feedback:

- **CPU Utilization Graphs:** We observed real-time graphs showing the percentage of CPU utilization. When instances were stressed, we saw significant spikes in CPU usage, often

exceeding the 50% threshold we set for the alarm.

- **Alarm State:** CloudWatch clearly indicated when the CPU over threshold alarm state was activated, confirming that our monitoring mechanism was working as intended.
- **Scaling Events:** Following the alarm, the ASG's activity history (discussed below) showed the launch of new instances, which then helped bring down the overall CPU utilization as the load was distributed.

4.3.7.2 Network Performance Monitoring

Beyond CPU, CloudWatch allowed us to monitor other critical metrics for our EC2 instances, providing a holistic view of their performance:

- **Network In (Bytes):** Monitored the amount of incoming network traffic to the instances. Spikes here would indicate increased user requests.
- **Network Out (Bytes):** Monitored the amount of outgoing network traffic from the instances. This is vital for observing responses sent to clients or data transferred to other services.
- **Network Packets In/Out (Count):** Provided insights into the volume of network packets processed, useful for identifying potential bottlenecks or unusual traffic patterns.
- **CPU Credit Usage/Balance (for T-type instances):** For t3.micro instances (burst-capable), monitoring CPU credit balance was crucial. It showed if instances were running out of credits, indicating sustained high CPU usage that might necessitate a larger instance type or more aggressive scaling.

The continuous monitoring of these metrics helped us validate the effectiveness of our load balancing and auto-scaling strategies and understand the application's behavior under various loads.

4.3.7.3 Auto Scaling Activity Logging

CloudWatch logs captured the activity history of our Auto Scaling Group, providing an auditable trail of all scaling events. This was critical for troubleshooting and understanding how the ASG responded to changes in demand or instance health:

- **Launch Events:** Logs showed Launching a new EC2 instance... with timestamps, indicating when new instances were added in response to a scale-out policy or to replace an unhealthy instance.

- **Termination Events:** Logs displayed Terminating EC2 instance... when instances were removed due to scale-in policies or health check failures.
- **Cause and Status:** Each log entry detailed the Cause of the activity ("An instance was launched in response to an unhealthy instance needing to be replaced," or "An instance was launched in response to a dynamic scaling policy") and its Status (Successful, Connection draining in progress).

This detailed logging provided irrefutable evidence of the auto-scaling mechanism's functionality, confirming that the system was indeed self-healing and elastic, dynamically adjusting its capacity to maintain performance and availability.

4.4 Performance Validation and Optimization

The ultimate validation of Part Two's implementation came through practical performance tests.

- **Load Simulation:** We simulated increasing load on the application by repeatedly accessing the load balancer's DNS name or by using load testing tools.
- **Observation of Scaling:** As the load increased and CPU utilization crossed the 50% threshold, we observed new instances being launched by the ASG in the CloudWatch metrics and ASG activity history. This confirmed that our dynamic scaling policy was correctly configured and triggered.
- **Distribution via ALB:** The "Hello from Group 4" and "Hello from Group FOUR" messages, alternating upon refresh, consistently demonstrated that the ALB was effectively distributing traffic across multiple instances, ensuring no single instance became a bottleneck.
- **Stability under Load:** Post-scaling, the overall CPU utilization across the ASG would typically stabilize, even with sustained load, indicating that the system was effectively handling the increased demand by adding capacity.

While this project focused on demonstrating the core mechanisms, a production environment would involve more sophisticated load testing, detailed performance benchmarking, and continuous optimization based on deeper insights from CloudWatch and other monitoring tools.

4.5 Challenges and Solutions in Auto Scaling and Load Balancing

Implementing auto scaling and load balancing presented its own set of challenges:

- **Initial Configuration Complexity:** Setting up the ELB, Target Group, Launch Template, and ASG, and ensuring all components were correctly linked, required careful attention to

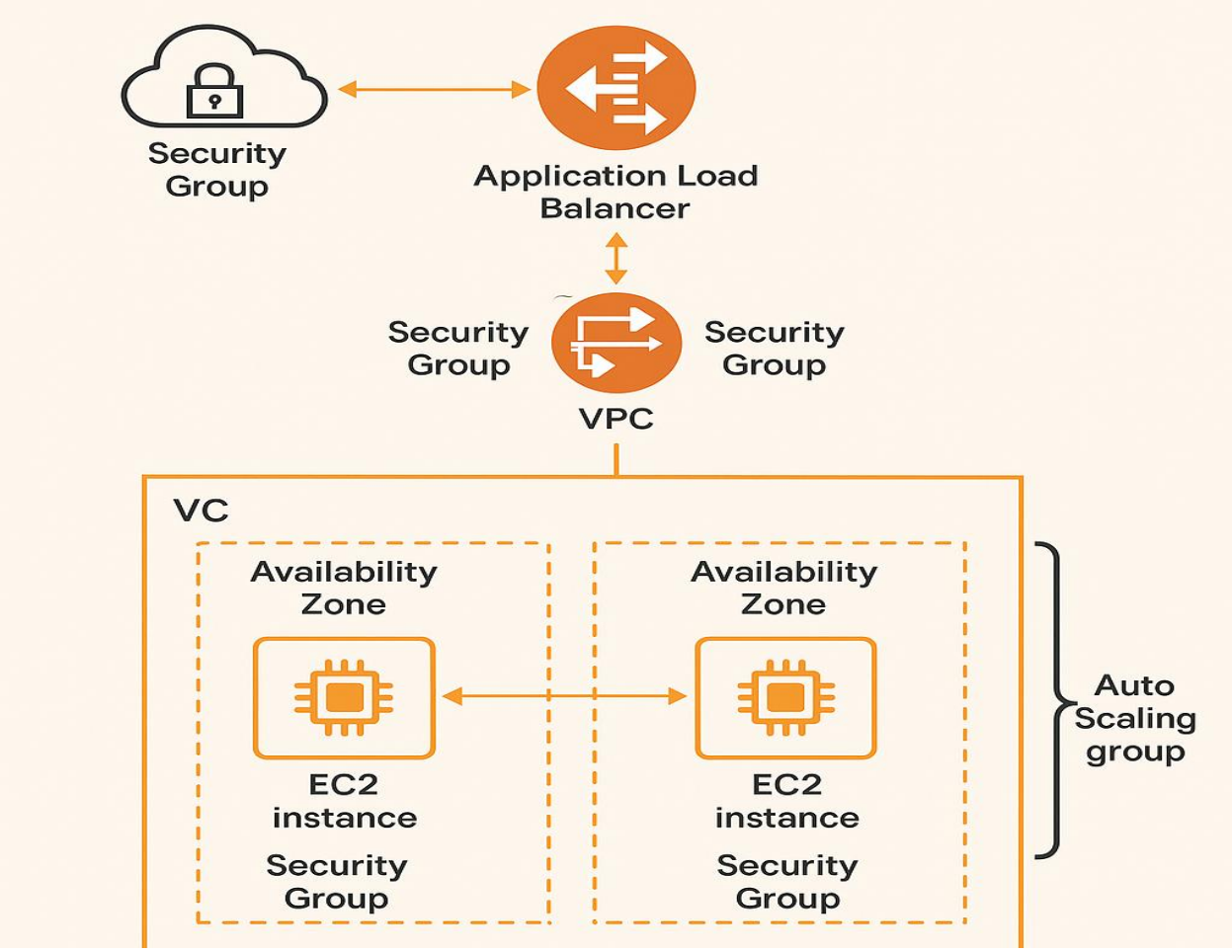
detail. A single misconfiguration (e.g., wrong security group on the launch template, incorrect port in target group) could prevent traffic flow or scaling. Solution involved systematic step-by-step configuration and immediate testing of each component.

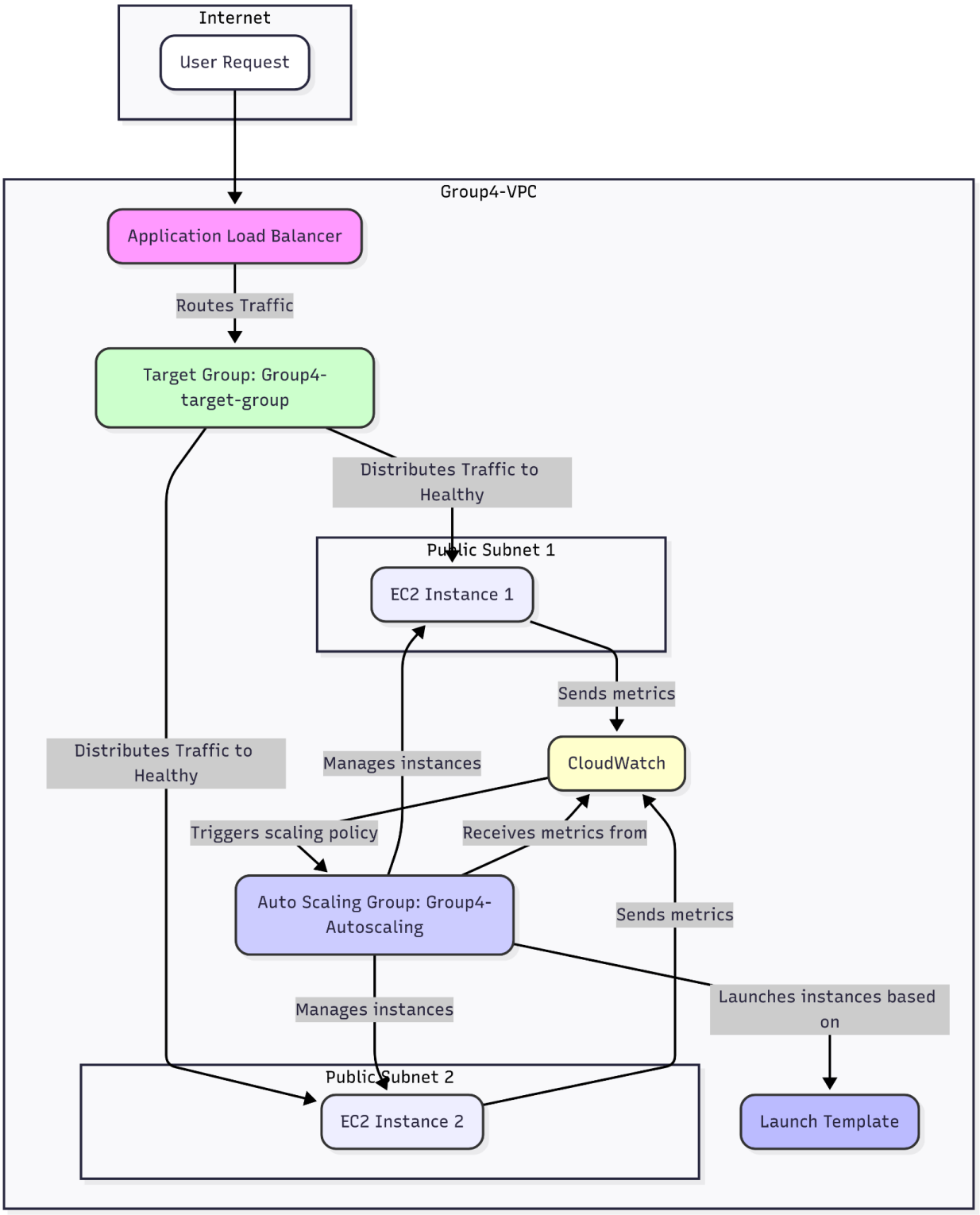
- **Choosing Appropriate Scaling Metrics:** While CPU utilization is a common metric, understanding when and how to apply it, and considering other metrics like Network In/Out, Request Count Per Target, or custom application metrics, is crucial. For this project, CPU was sufficient to demonstrate elasticity.
- **Effectively Stressing Instances:** To prove the auto-scaling mechanism, we needed to generate sufficient load to push CPU utilization above the threshold. Tools like stress (a Linux utility) or simple while true; do ... done loops were used on the EC2 instances to achieve this. Ensuring the stress was consistent enough to trigger the alarm but not so overwhelming as to crash the instance required iterative testing.
- **Debugging Load Balancer Issues:** If the load balancer wasn't routing traffic correctly, troubleshooting involved checking:
    - Security Group rules (ALB to EC2, client to ALB).
    - Target Group health checks.
    - Listener rules.
    - Network ACLs (if active). This systematic approach helped identify and resolve connectivity problems.
- **Understanding ASG Cooldown Periods:** Initially, rapid, unmanaged scaling might occur without proper cooldowns. Understanding how cooldown periods prevent "flapping" (rapid scaling up and down) was important for stable operation, even if a short cooldown was used for demonstration.

By approaching each component systematically, validating configurations step-by-step, and effectively using CloudWatch for diagnosis, we successfully built and demonstrated a highly available and elastic infrastructure.

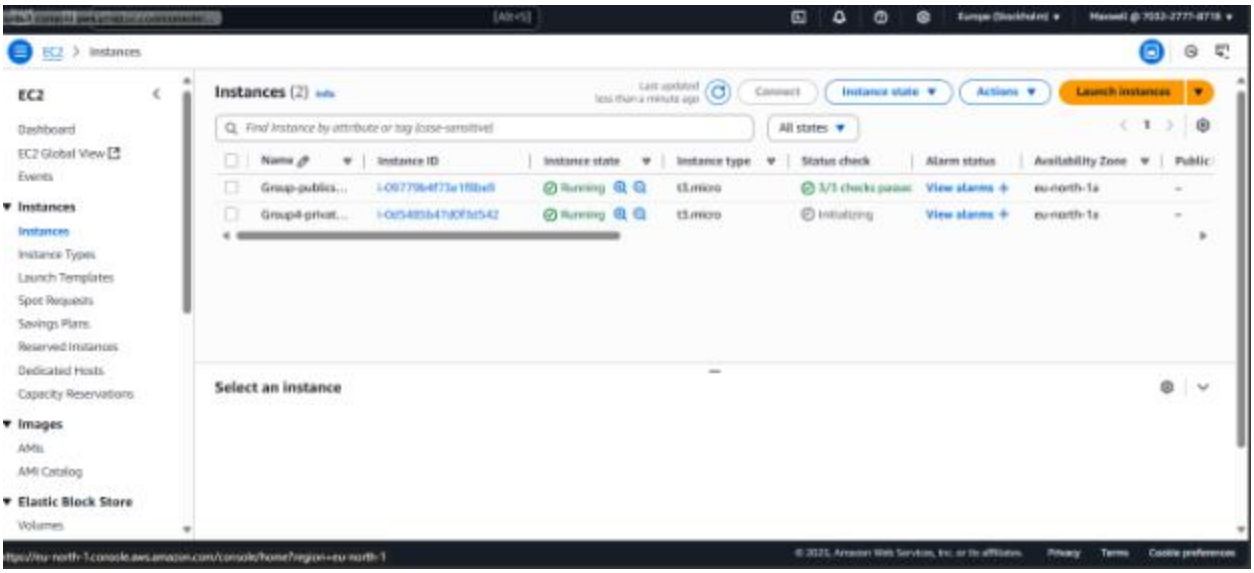
4.6 Architecture Diagram: High Availability and Elasticity

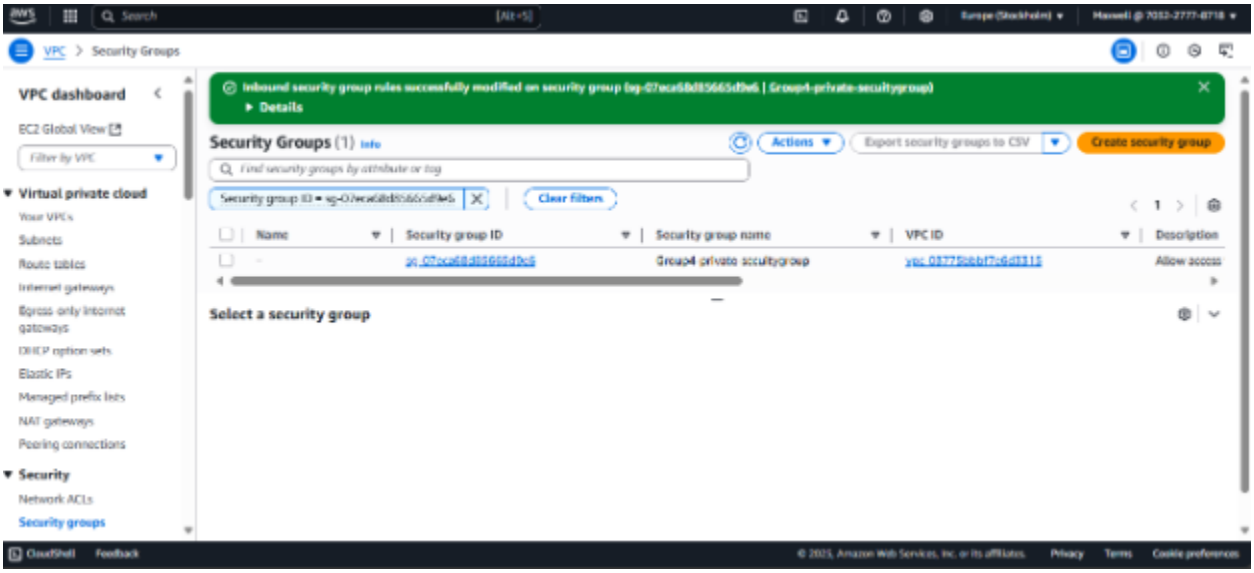
These diagrams illustrate the components responsible for high availability and elasticity, including the Application Load Balancer, Target Group, Auto Scaling Group, and their interaction with EC2 instances and CloudWatch within the VPC.

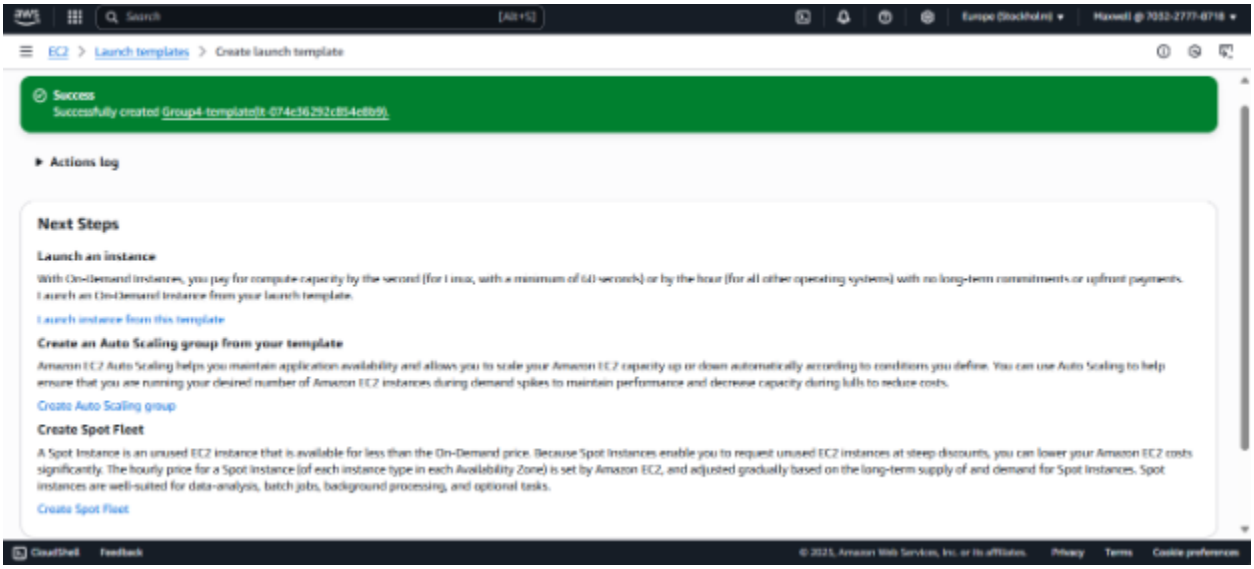

Security Group
Application Load Balancer
Security Group
Security Group
VPC
VC
Availability Zone
Availability Zone
EC2 instance
EC2 instance
Security Group
Security Group
Auto Scaling group

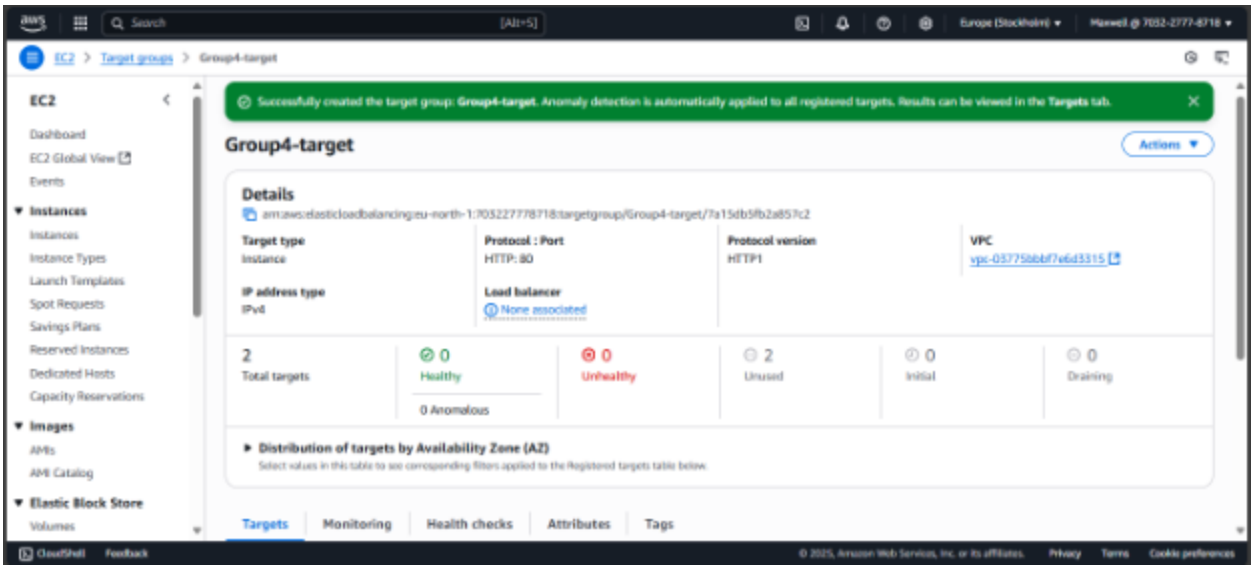


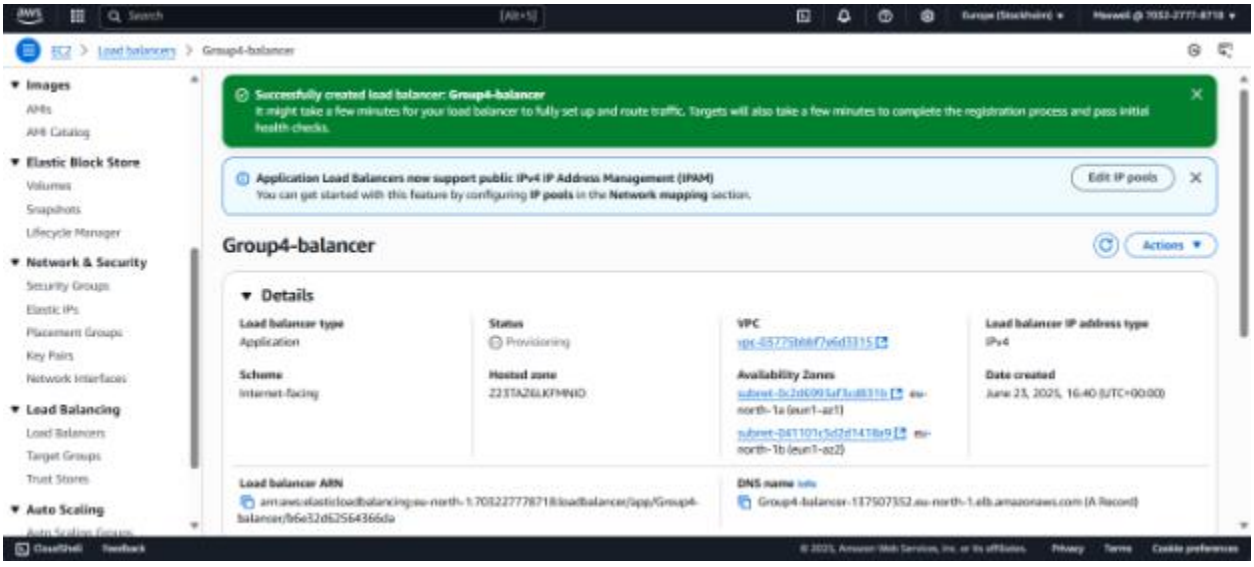
4.7 Method: High Availability and Elasticity


EC2 > Instances
EC2
Dashboard
EC2 Global View
Events
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations
Images
AMIs
AMI Catalog
Elastic Block Store
Volumes
Instances (2) info
Last updated less than a minute ago
Connect
Instance state
Actions
Launch instances
Find instance by attribute or tag (case-sensitive)
All states
Name
Instance ID
Instance state
Instance type
Status check
Alarm status
Availability Zone
Public
Group-public...
i-09779b4f73e1f8be8
Running
t3.micro
3/3 checks passed
View alarms
eu-north-1a
Group4-privat...
i-0d5485b47d0f3d542
Running
t3.micro
Initializing
View alarms
eu-north-1a
Select an instance

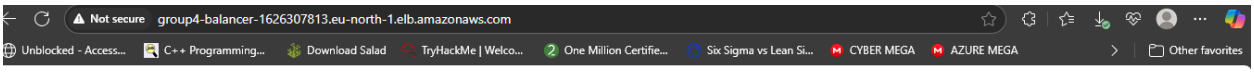

VPC > Security Groups
VPC dashboard
EC2 Global View
Filter by VPC
Virtual private cloud
Your VPCs
Subnets
Route tables
Internet gateways
Egress only internet gateways
DHCP option sets
Elastic IPs
Managed prefix lists
NAT gateways
Peering connections
Security
Network ACLs
Security groups
Inbound security group rules successfully modified on security group (sg-07eca6b85665d9e6 | Group4-private-securitygroup)
Details
Security Groups (1) info
Actions
Export security groups to CSV
Create security group
Find security groups by attribute or tag
Clear filters
Name
Security group ID
Security group name
VPC ID
Description
Group4 private securitygroup
Allow access
Select a security group


EC2 > Launch templates > Create launch template
Success
Successfully created Group4-template(lt-074c16292c854e8b9).
Actions log
Next Steps
Launch an instance
With On-Demand instances, you pay for compute capacity by the second (for Linux, with a minimum of 60 seconds) or by the hour (for all other operating systems) with no long-term commitments or upfront payments. Launch an On-Demand instance from your launch template.
Launch instance from this template
Create an Auto Scaling group from your template
Amazon EC2 Auto Scaling helps you maintain application availability and allows you to scale your Amazon EC2 capacity up or down automatically according to conditions you define. You can use Auto Scaling to help ensure that you are running your desired number of Amazon EC2 instances during demand spikes to maintain performance and decrease capacity during lulls to reduce costs.
Create Auto Scaling group
Create Spot Fleet
A Spot Instance is an unused EC2 instance that is available for less than the On-Demand price. Because Spot Instances enable you to request unused EC2 instances at steep discounts, you can lower your Amazon EC2 costs significantly. The hourly price for a Spot Instance (of each instance type in each Availability Zone) is set by Amazon EC2, and adjusted gradually based on the long-term supply of and demand for Spot Instances. Spot instances are well-suited for data-analysis, batch jobs, background processing, and optional tasks.
Create Spot Fleet

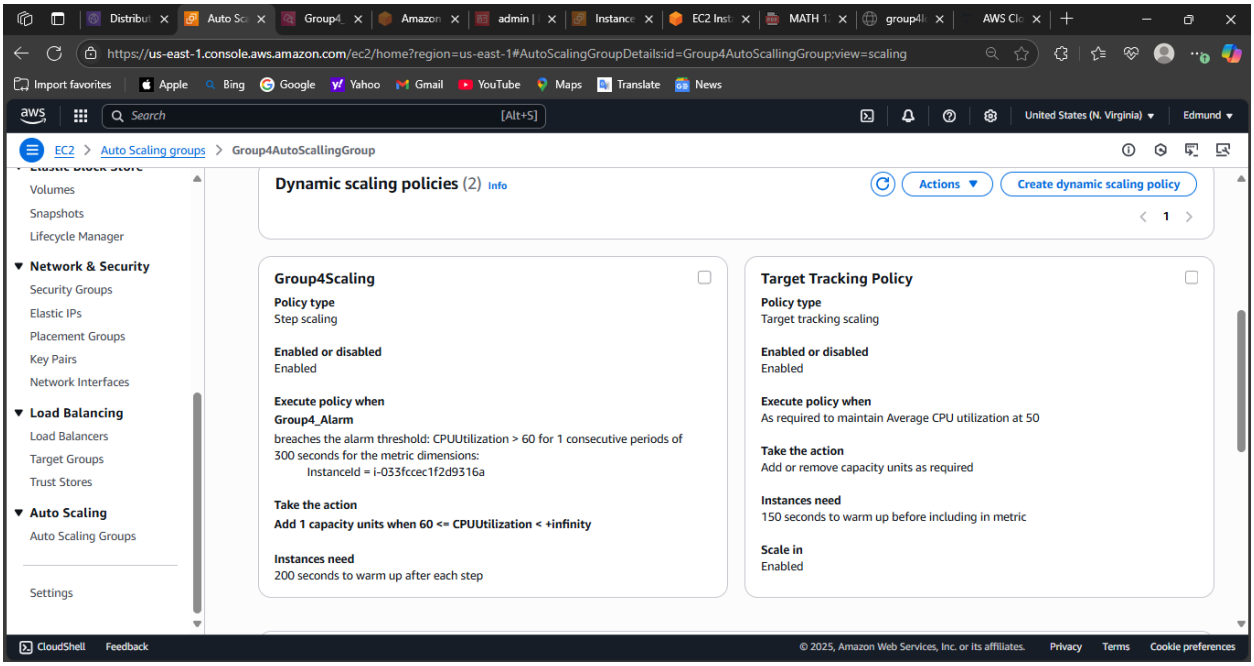

Search
[Alt+S]
Europe (Stockholm)
EC2 > Target groups > Group4-target
EC2
Dashboard
EC2 Global View
Events
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations
Images
AMIs
AMI Catalog
Elastic Block Store
Volumes
Successfully created the target group: Group4-target. Anomaly detection is automatically applied to all registered targets. Results can be viewed in the Targets tab.
Group4-target
Actions
Details
arn:aws:elasticloadbalancing:eu-north-1:703227778718:targetgroup/Group4-target/7a15db5fb2a857c2
Target type
Instance
Protocol : Port
HTTP: 80
Protocol version
HTTP1
VPC
vpc-0377586bbf7a6d3315
IP address type
IPv4
Load balancer
None associated
2
Total targets
0
Healthy
0 Anomalous
0
Unhealthy
2
Unused
0
Initial
0
Draining
Distribution of targets by Availability Zone (AZ)
Select values in this table to see corresponding filters applied to the Registered targets table below.
Targets
Monitoring
Health checks
Attributes
Tags
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


Search
[Alt+S]
Europe (Stockholm)
EC2 > Load balancers > Group4-balancer
Images
AMIs
AMI Catalog
Elastic Block Store
Volumes
Snapshots
Lifecycle Manager
Network & Security
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces
Load Balancing
Load Balancers
Target Groups
Trust Stores
Auto Scaling
Auto Scaling Groups
Successfully created load balancer: Group4-balancer
It might take a few minutes for your load balancer to fully set up and route traffic. Targets will also take a few minutes to complete the registration process and pass initial health checks.
Application Load Balancers now support public IPv4 IP Address Management (IPAM)
You can get started with this feature by configuring IP pools in the Network mapping section.
Edit IP pools
Group4-balancer
Actions
Details
Load balancer type
Application
Status
Provisioning
VPC
vpc-0377586bbf7a6d3315
Load balancer IP address type
IPv4
Scheme
Internet-facing
Hosted zone
Z23TAZ6LKFMNIO
Availability Zones
subnet-0c2d6093af3cd831b eu-north-1a (eun1-az1)
subnet-041101c5d2d1418a9 eu-north-1b (eun1-az2)
Date created
June 23, 2025, 16:40 (UTC+00:00)
Load balancer ARN
arn:aws:elasticloadbalancing:eu-north-1:703227778718:loadbalancer/app/Group4-balancer/b6e32d625643660da
DNS name
Group4-balancer-1375073S2.eu-north-1.elb.amazonaws.com (A Record)
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

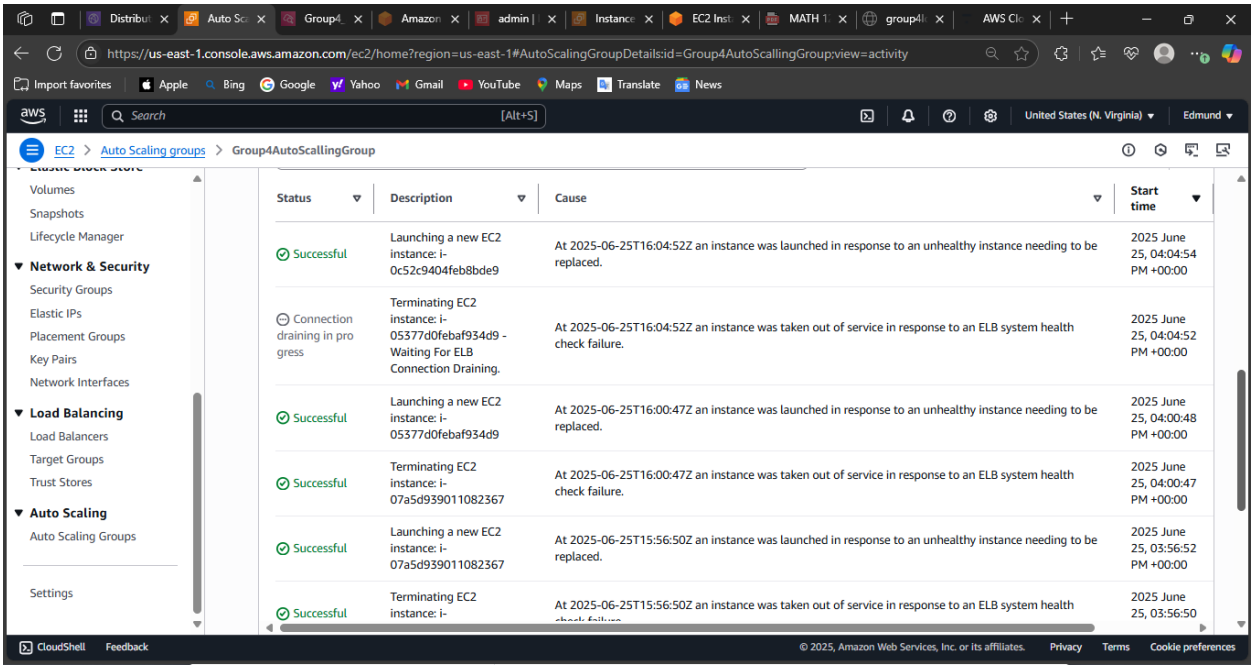
Hello from Group 4

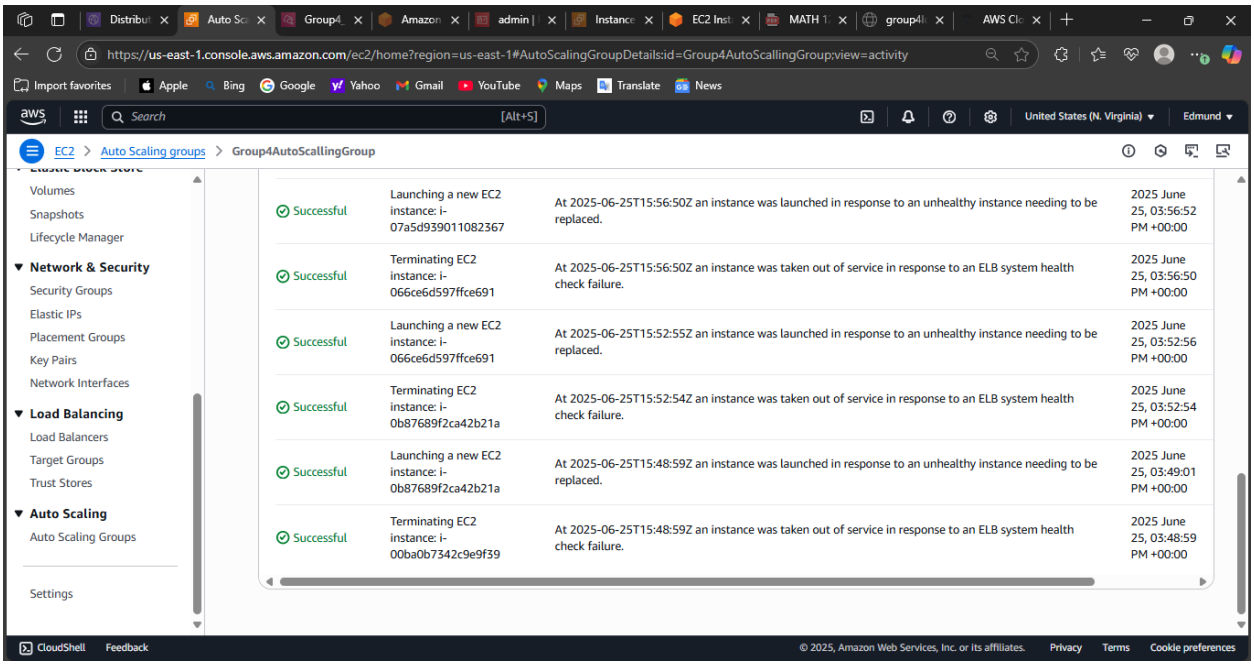


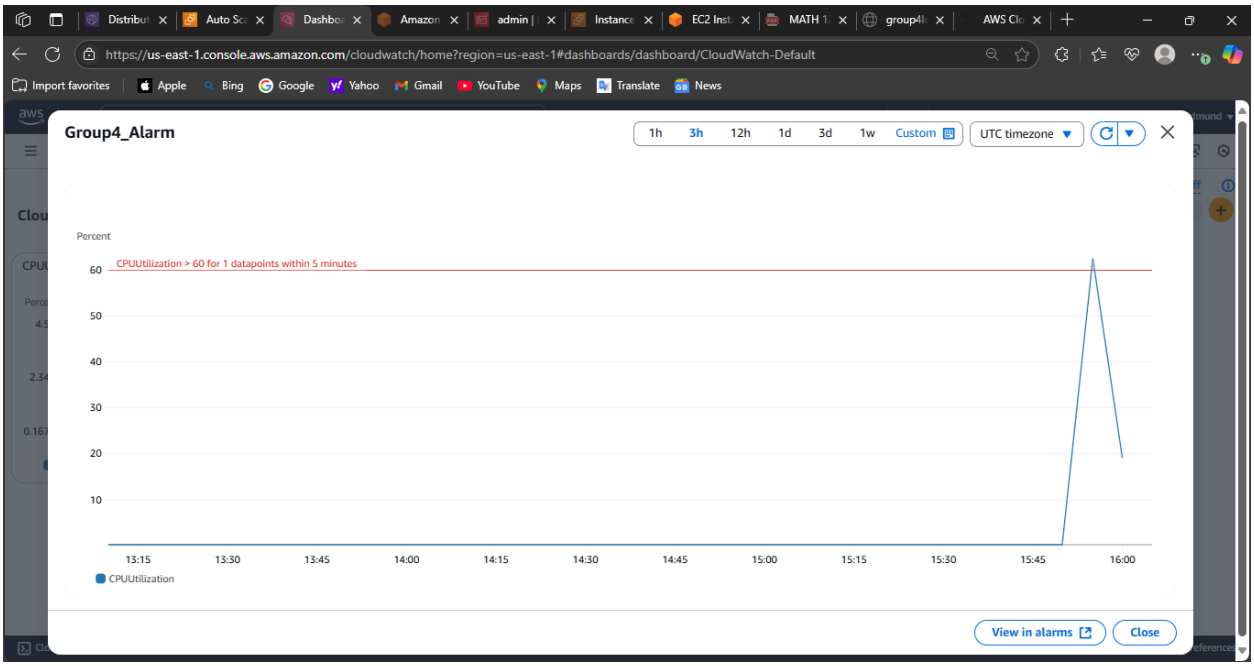
Hello from Group FOUR

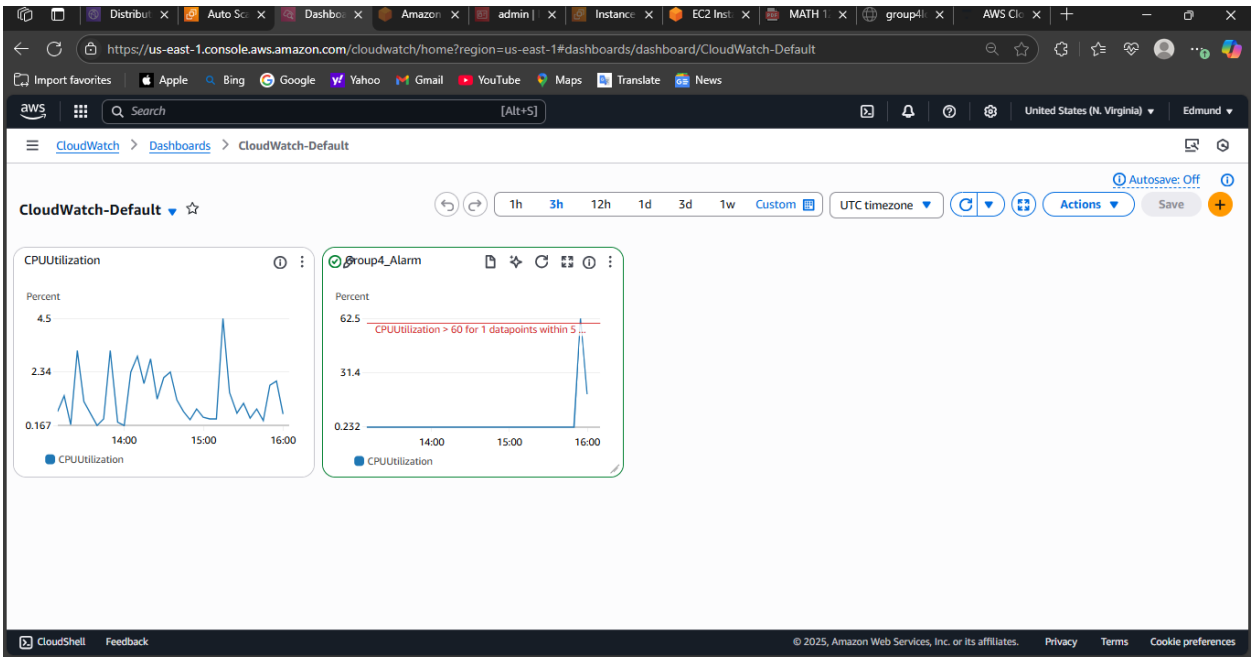



```
2 package(s) needed for security, out of 4 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-10-0-1-122 ~]$ sudo yum install stress -y
Loaded plugins: extras_suggestions, langpacks, priorities, update-motd
amzn2-core                                                 | 3.6 kB  00:00:00
230 packages excluded due to repository priority protections
Package stress-1.0.4-16.el7.x86_64 already installed and latest version
Nothing to do
[ec2-user@ip-10-0-1-122 ~]$ stress --cpu 2 timeout 60
stress: FAIL: [5251] (244) unrecognized option: timeout
[ec2-user@ip-10-0-1-122 ~]$ stress --cpu 2 --timeout 60
stress: info: [5252] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
stress: info: [5252] successful run completed in 60s
[ec2-user@ip-10-0-1-122 ~]$ stress --cpu 2 --timeout 60
stress: info: [5268] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
stress: info: [5268] successful run completed in 60s
[ec2-user@ip-10-0-1-122 ~]$ stress --cpu 2 --timeout 60
stress: info: [5272] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
stress: info: [5272] successful run completed in 60s
[ec2-user@ip-10-0-1-122 ~]$ stress --cpu 2 --timeout 60
stress: info: [5275] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
stress: info: [5275] successful run completed in 60s
[ec2-user@ip-10-0-1-122 ~]$ stress --cpu 2 --timeout 60
stress: info: [5328] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
```


https://us-east-1.console.aws.amazon.com/ec2/home?region=us-east-1#AutoScalingGroupDetails:id=Group4AutoScallingGroup;view=activity
EC2 > Auto Scaling groups > Group4AutoScallingGroup
Volumes
Snapshots
Lifecycle Manager
Network & Security
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces
Load Balancing
Load Balancers
Target Groups
Trust Stores
Auto Scaling
Auto Scaling Groups
Settings
Status
Description
Cause
Start time
Successful
Launching a new EC2 instance: i-0c52c9404feb8bde9
At 2025-06-25T16:04:52Z an instance was launched in response to an unhealthy instance needing to be replaced.
2025 June 25, 04:04:54 PM +00:00
Connection draining in progress
Terminating EC2 instance: i-05377d0febaf934d9 - Waiting For ELB Connection Draining.
At 2025-06-25T16:04:52Z an instance was taken out of service in response to an ELB system health check failure.
2025 June 25, 04:04:52 PM +00:00
Launching a new EC2 instance: i-05377d0febaf934d9
At 2025-06-25T16:00:47Z an instance was launched in response to an unhealthy instance needing to be replaced.
2025 June 25, 04:00:48 PM +00:00
Terminating EC2 instance: i-07a5d939011082367
At 2025-06-25T16:00:47Z an instance was taken out of service in response to an ELB system health check failure.
2025 June 25, 04:00:47 PM +00:00
Launching a new EC2 instance: i-07a5d939011082367
At 2025-06-25T15:56:50Z an instance was launched in response to an unhealthy instance needing to be replaced.
2025 June 25, 03:56:52 PM +00:00
Terminating EC2 instance: i-
At 2025-06-25T15:56:50Z an instance was taken out of service in response to an ELB system health
2025 June 25, 03:56:50
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


https://us-east-1.console.aws.amazon.com/ec2/home?region=us-east-1#AutoScalingGroupDetails:id=Group4AutoScallingGroup;view=activity
EC2 > Auto Scaling groups > Group4AutoScallingGroup
Volumes
Snapshots
Lifecycle Manager
Network & Security
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces
Load Balancing
Load Balancers
Target Groups
Trust Stores
Auto Scaling
Auto Scaling Groups
Settings
Successful
Launching a new EC2 instance: i-07a5d939011082367
At 2025-06-25T15:56:50Z an instance was launched in response to an unhealthy instance needing to be replaced.
2025 June 25, 03:56:52 PM +00:00
Successful
Terminating EC2 instance: i-066ce6d597ffce691
At 2025-06-25T15:56:50Z an instance was taken out of service in response to an ELB system health check failure.
2025 June 25, 03:56:50 PM +00:00
Successful
Launching a new EC2 instance: i-066ce6d597ffce691
At 2025-06-25T15:52:55Z an instance was launched in response to an unhealthy instance needing to be replaced.
2025 June 25, 03:52:56 PM +00:00
Successful
Terminating EC2 instance: i-0b87689f2ca42b21a
At 2025-06-25T15:52:54Z an instance was taken out of service in response to an ELB system health check failure.
2025 June 25, 03:52:54 PM +00:00
Successful
Launching a new EC2 instance: i-0b87689f2ca42b21a
At 2025-06-25T15:48:59Z an instance was launched in response to an unhealthy instance needing to be replaced.
2025 June 25, 03:49:01 PM +00:00
Successful
Terminating EC2 instance: i-00ba0b7342c9e9f39
At 2025-06-25T15:48:59Z an instance was taken out of service in response to an ELB system health check failure.
2025 June 25, 03:48:59 PM +00:00
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences





5. Part Three: Student Data Logger with Amazon DynamoDB Integration

5.1 Introduction to Serverless Databases and DynamoDB

In the evolving landscape of cloud-native development, the demand for simple, scalable, and high-performance data solutions is paramount. Traditional relational databases, while powerful, often come with the overhead of server management, capacity planning, and complex scaling.

Serverless databases, like Amazon DynamoDB, offer a compelling alternative by abstracting away the underlying infrastructure, allowing developers to focus solely on application logic. Amazon DynamoDB is a fully managed, serverless NoSQL database service offered by AWS. It is designed for single-digit millisecond performance at any scale. Key characteristics that make it ideal for modern applications include:

- **NoSQL Nature:** DynamoDB is a key-value and document database, providing flexibility in schema design, which is beneficial for rapidly evolving applications.
- **Serverless:** Users don't provision or manage servers; AWS handles all the operational tasks like patching, backups, and scaling.
- **Scalability:** It can scale to virtually unlimited throughput and storage, making it suitable for applications with unpredictable or high-volume traffic.
- **Performance:** Consistently delivers fast performance, even at massive scale, through its distributed architecture.
- **Durability and Availability:** Offers built-in replication across multiple Availability Zones to ensure high durability and availability.

Part Three of our capstone project showcased the efficient use of DynamoDB to build a lightweight, secure, and performant student data logging solution. This phase demonstrated how to seamlessly integrate a backend database with compute resources within our established secure VPC.

5.2 Project Scope and Objectives for Data Integration

The primary objective for Part Three was to establish a functional data logging mechanism that could reliably store and retrieve student records. This involved:

- **Database Provisioning:** Creating and configuring a DynamoDB table with an appropriate primary key.
- **Secure Access:** Implementing a secure method for an EC2 instance to interact with DynamoDB, adhering to the principle of least privilege using IAM roles.
- **Programmatic Interaction:** Developing a Python script leveraging the Boto3 SDK to perform basic Create, Read, Update, Delete (CRUD) operations, specifically focusing on data insertion.
- **Data Validation:** Verifying that data inserted via the script was accurately persisted and viewable in the DynamoDB console.

This phase aimed to complete the three-tier application architecture: presentation (implied by

web servers in Part Two), application logic (on EC2), and data storage (DynamoDB).

5.3 Key Tasks and Methodologies

5.3.1 DynamoDB Table Creation

The initial step in integrating the database was to create the DynamoDB table itself.

- **Table Name:** We named our table Students. This name clearly reflects the type of data it would store.
- **Primary Key:** We designated studentID as the **partition key** (of type String). The partition key is crucial for distributing data across DynamoDB's partitions and for efficient data retrieval. It ensures that items with the same studentID are stored together, enabling fast lookups. In a real-world scenario, if studentID wasn't unique enough, a sort key could be added to form a composite primary key.
- **Capacity Mode:** While not explicitly shown in the image, the table was likely configured for **On-demand capacity mode**. This mode is ideal for workloads that are unpredictable or bursty, as it automatically scales read and write capacity based on traffic, eliminating the need for manual provisioning and optimizing costs.
- **Point-in-time Recovery (PITR):** The image indicates "Point-in-time recovery (PITR): Off." For a production system, enabling PITR would be a critical decision, as it provides continuous backups of your table data, allowing restoration to any point within the last 35 days, significantly enhancing data durability and disaster recovery capabilities.

This foundational step laid the schema-flexible groundwork for our NoSQL database, optimized for fast read/write operations and inherent scalability.

5.3.2 EC2 Instance Preparation for Development

To host our Python data logger application and interact with DynamoDB, we prepared a public EC2 instance. This instance served as our development and testing environment.

- **Instance Selection:** We utilized an existing or launched a new EC2 instance, preferably within one of our public subnets, to allow for ease of access (via SSH) for development purposes. While the database itself would reside in a private subnet (or be a managed service like DynamoDB, which is accessed via endpoints), the application interacting with it often needs a compute instance.

- **Software Installation:** Critical software was installed on this EC2 instance:
    - **Python 3:** The programming language used for our data logger script.
    - **Boto3:** The Amazon Web Services (AWS) SDK for Python. Boto3 allows Python developers to write software that makes use of AWS services like S3, EC2, and DynamoDB. Its installation was verified using pip3 install boto3.

This preparation ensured that our EC2 environment had all the necessary tools and libraries to seamlessly interact with AWS services, particularly DynamoDB.

5.3.3 IAM Role Attachment for Secure Access

A cornerstone of cloud security is the principle of least privilege, and for programmatic access to AWS services, IAM (Identity and Access Management) roles are the best practice. Instead of embedding AWS access keys directly onto the EC2 instance (which is highly insecure), we attached an IAM role to our EC2 instance.

- **IAM Role Creation:** An IAM role named DynamoUser (or similar descriptive name) was created.
- **Permissions Policy:** This role was granted the AmazonDynamoDBFullAccess managed policy. This policy provides comprehensive permissions to perform all DynamoDB actions (Create, Read, Update, Delete) on any DynamoDB table. While FullAccess was suitable for a project demonstration, in a production environment, a custom policy with highly granular, minimal necessary permissions (e.g., dynamodb:PutItem on a specific table) would be preferred.
- **Trust Policy:** The role's trust policy was configured to allow the EC2 service (ec2.amazonaws.com) to assume this role.
- **Attachment to EC2 Instance:** Once created, the DynamoUser IAM role was attached to our prepared EC2 instance.

When this IAM role is attached to an EC2 instance, temporary security credentials are automatically made available to applications running on that instance. This eliminates the need to hardcode or manually manage credentials, significantly enhancing security and compliance. Our Python script using Boto3 would automatically pick up these temporary credentials from the instance's metadata service, enabling secure, credential-free access to DynamoDB.

5.3.4 Python and Boto3 SDK Installation and Verification

Following the instance preparation, a critical verification step was performed to ensure that Python and Boto3 were correctly installed and functional, and that the IAM role was properly assumed.

- **Installation Confirmation:** The screenshot shows pip3 install boto3 successfully completing, indicating that the SDK and its dependencies were installed on the EC2 instance.
- **Initial Boto3 Test Script:** A simple Python script (e.g., group4.py) was created and executed. A basic test within this script could be:

  ```
  import boto3
  try:
      dynamodb = boto3.resource('dynamodb')
      # Attempt to list tables to verify connectivity
      tables = list(dynamodb.tables.all())
      print("Boto3 is working! DynamoDB tables accessible.")
  except Exception as e:
      print(f"Error: {e}")
  ```

  The output Boto3 is working! confirmed that the Boto3 SDK could initialize and connect to AWS services, implying the IAM role attachment was successful and provided the necessary permissions.
- **AWS CLI Verification:** To further confirm the environment setup and the IAM role's permissions, the AWS Command Line Interface (CLI) was used. While aws configure (for manual credential setup) was initially shown as an option, the subsequent command aws dynamodb list-tables was executed without explicit credentials. The successful output showing "TableNames": ["Students"] confirmed that the EC2 instance, by virtue of its attached IAM role, had the necessary permissions to list DynamoDB tables. This reinforced trust in the secure, role-based access model.

This comprehensive verification ensured that the development environment was ready for advanced database interactions.

5.3.5 Script Development and Testing for Data Operations

A core part of Part Three involved developing a Python script (group4.py) to perform data operations on the Students DynamoDB table. This script was created directly from the EC2 terminal using a text editor like vim.

The script was designed to:

- **Connect to DynamoDB:** Initialize a Boto3 DynamoDB client or resource object.
- **Access the Table:** Specify the Students table.
- **Perform CRUD Operations:** While the primary focus for the demonstration was data insertion, the script was conceptualized to be capable of:
    - **Create/Put Item:** Insert new student records.
    - **Get Item:** Retrieve a specific student record by studentID.
    - **Update Item:** Modify existing student records.
    - **Delete Item:** Remove student records.

For the demonstration, the script focused on inserting sample student records. The initial test, python3 group4.py, which simply printed "Boto3 is working!", served as a quick functional test. This simple script was invaluable as both a functionality verifier (confirming Boto3 was correctly installed and authorized) and a demo tool for the basic connectivity.

5.3.6 Data Insertion and Verification

The main functionality demonstrated was the insertion of student records into the DynamoDB table. The Python script was executed to perform this operation.

- **Script Execution:** The command python3 group4.py was run on the EC2 instance.
- **Data Structure:** The script would define a list of student data, each with attributes like studentID, Name, Age, and Department. For example:

```
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Students')

students_data = [
    {'studentID': '1', 'Name': 'John', 'Age': 21, 'Department':
'Computer Science'},
    {'studentID': '2', 'Name': 'Alice', 'Age': 22, 'Department':
'Mathematics'},
    {'studentID': '3', 'Name': 'Bob', 'Age': 20, 'Department':
'Physics'},
    {'studentID': '4', 'Name': 'David', 'Age': 23, 'Department':
```

```
'Chemistry'}
]

for student in students_data:
    try:
        response = table.put_item(Item=student)
        print(f"Inserted student {student['Name']} with status:
{response['ResponseMetadata']['HTTPStatusCode']}")
    except Exception as e:
        print(f"Error inserting student {student['Name']}: {e}")
```

- **Output Logs:** The terminal output clearly showed the successful insertion of each student record, with status: 200 indicating an OK HTTP response from the DynamoDB service for each put_item operation. This provided immediate confirmation of the script's successful execution and data submission.

This step completed the "round-trip workflow" from the EC2 instance (acting as the application server) to the DynamoDB backend, demonstrating reliable data transfer and storage.

5.3.7 Console-Based Data Validation

The final and crucial step for validating the data integration was to visually confirm the persistence and accuracy of the inserted data.

- **Navigation to DynamoDB Console:** The team navigated to the AWS Management Console, specifically to the DynamoDB service, and selected the Students table.
- **Explore Items:** Within the table details, the "Explore items" or "Items" tab was accessed.
- **Verification:** The console displayed the inserted records (studentID 1, 2, 3, 4 with their corresponding names, ages, and departments). The entries perfectly reflected the data generated and inserted by the Python script.

This direct visual confirmation from the AWS console solidified trust in both the Python script's logic and the backend DynamoDB's ability to store and manage data reliably. It served as the ultimate proof of a successful end-to-end data integration solution.

5.4 Security Considerations for Database Access

Security was a paramount consideration throughout Part Three, particularly concerning

database access.

- **IAM Roles over Access Keys:** The decision to use IAM roles attached to the EC2 instance, rather than hardcoding or distributing AWS access keys/secret keys, is a critical security best practice. IAM roles provide temporary, frequently rotated credentials automatically, significantly reducing the risk of compromised static credentials.
- **Least Privilege Principle:** While AmazonDynamoDBFullAccess was used for demonstration, the understanding that granular IAM policies should be created for production environments was established. This means granting only the specific permissions required for an application's function (dynamodb:PutItem only on the Students table) and nothing more.
- **Network Isolation:** Placing the DynamoDB table (a managed service, conceptually "in" the private subnet via VPC endpoints if used, or accessed over private links) and the EC2 instance within the secure VPC (as designed in Part One) provided network isolation. While DynamoDB is accessed via public endpoints, using VPC Endpoints for DynamoDB (though not explicitly mentioned as implemented) is a production best practice to ensure traffic remains within the AWS network, enhancing security and reducing data transfer costs.
- **No Direct Internet Exposure:** The DynamoDB table itself was never directly exposed to the public internet. Access was mediated through the securely configured EC2 instance and its IAM role.

These security measures ensured that the data logging solution was not only functional but also adhered to robust cloud security principles.

5.5 Challenges and Solutions in Database Integration

Several challenges were addressed during the database integration phase:

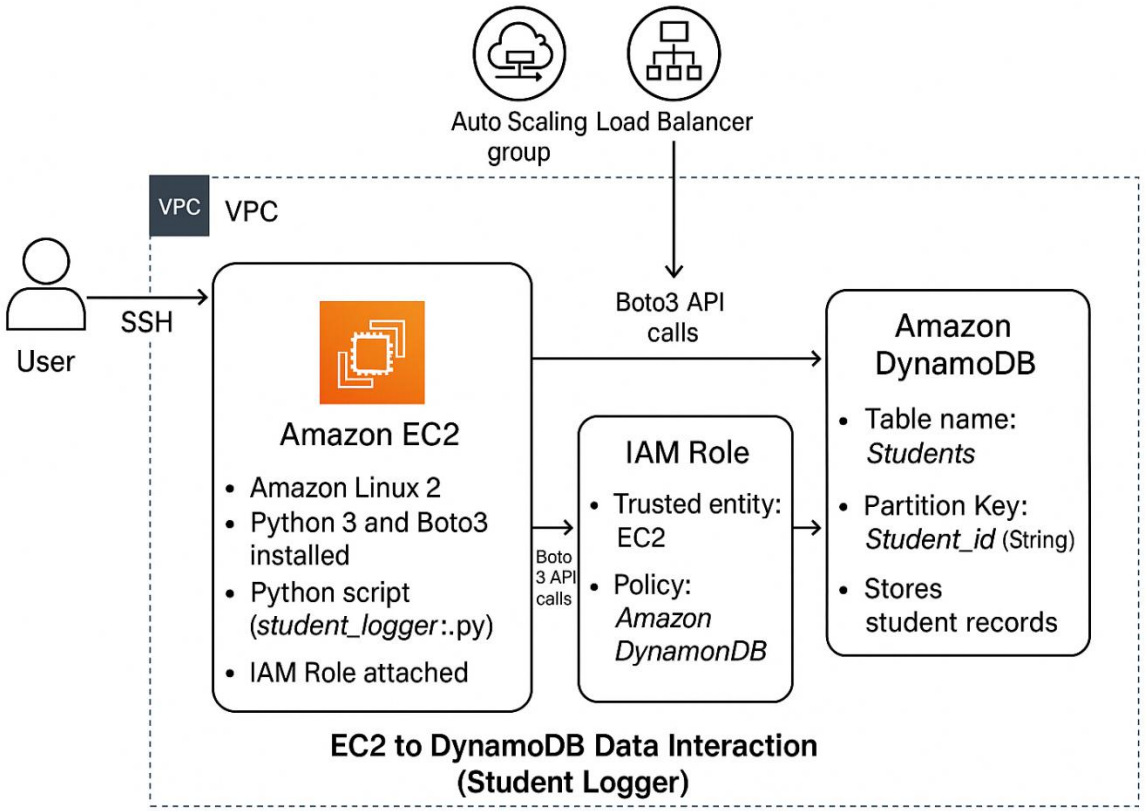
- **Boto3 Installation and Dependencies:** Ensuring all necessary Python packages and Boto3 dependencies were correctly installed on the EC2 instance. Solution: Relying on pip3 install boto3 and verifying installation with simple import statements and API calls.
- **IAM Permissions Misconfiguration:** Incorrect IAM role permissions (missing specific DynamoDB actions, or incorrect resource ARNs) could lead to "Access Denied" errors. Solution: Thoroughly reviewing IAM policies, checking CloudTrail logs for permission issues, and adopting an iterative "add permissions as needed" approach during development.

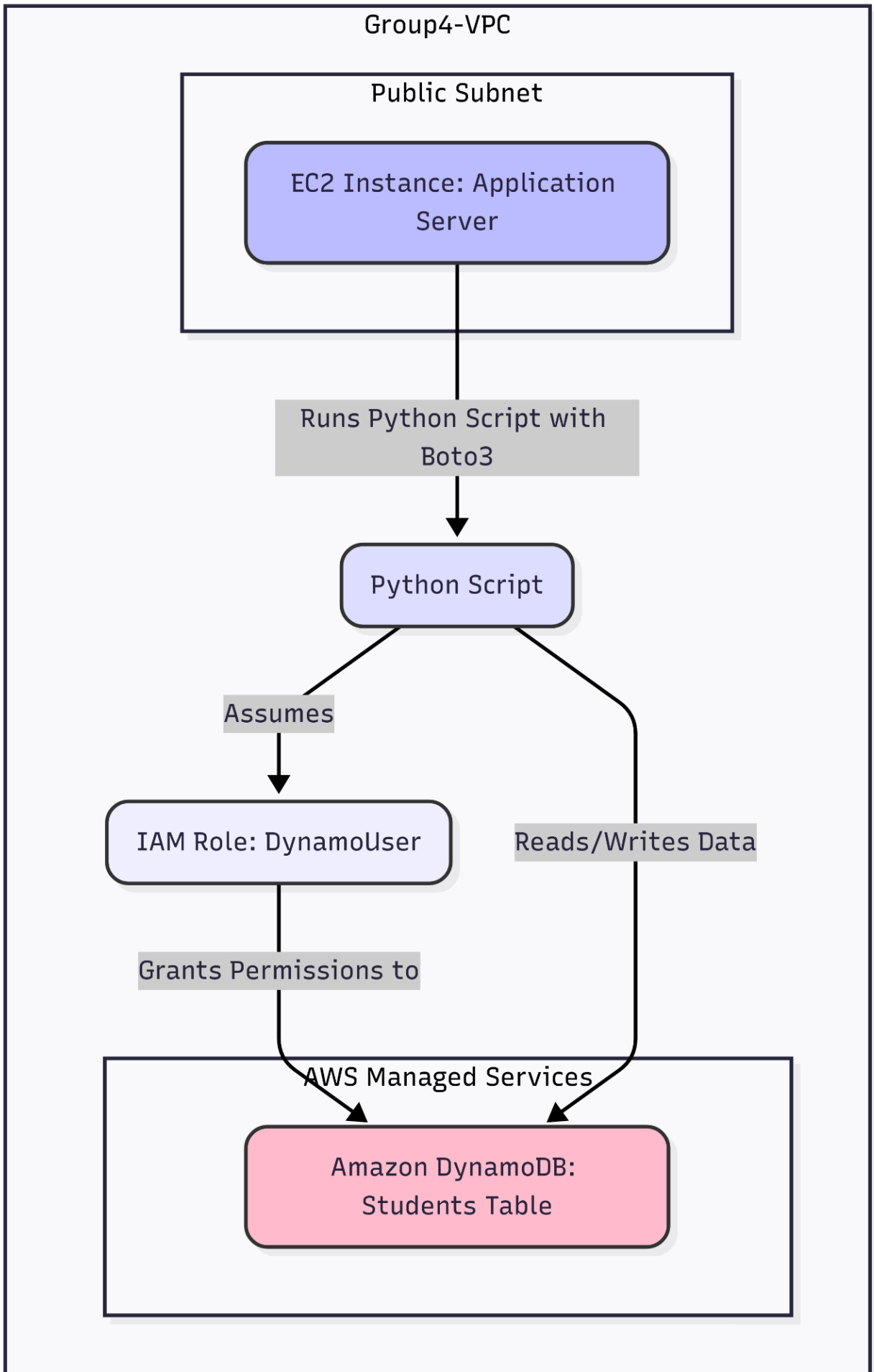
- **DynamoDB Table Schema (Primary Key):** Understanding the implications of the primary key for data distribution and query efficiency was important. Solution: Selecting studentID as the partition key based on anticipated access patterns (lookup by ID). For more complex queries, secondary indexes would be considered.
- **Network Connectivity to DynamoDB:** Although DynamoDB is a managed service, ensuring the EC2 instance could reach its endpoint over the network was crucial. This relied on the correct NAT Gateway and route table setup from Part One for instances in private subnets, or direct internet connectivity for public instances.
- **Python Script Debugging:** Standard Python debugging techniques (print statements, try-except blocks) were used to identify and resolve issues within the data insertion script, ensuring it correctly formatted data and handled API responses.

By systematically addressing these challenges, Group 4 successfully integrated DynamoDB into the cloud infrastructure, completing the backend data persistence layer of the application.

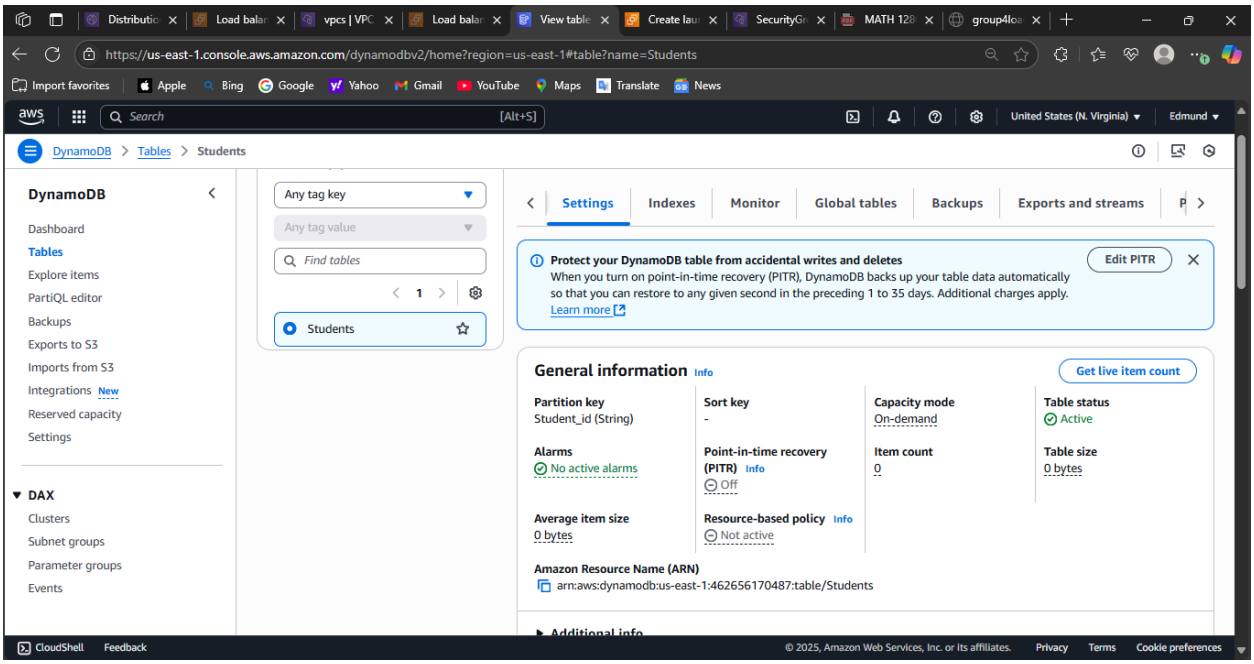
5.6 Architecture Diagram: Database Integration

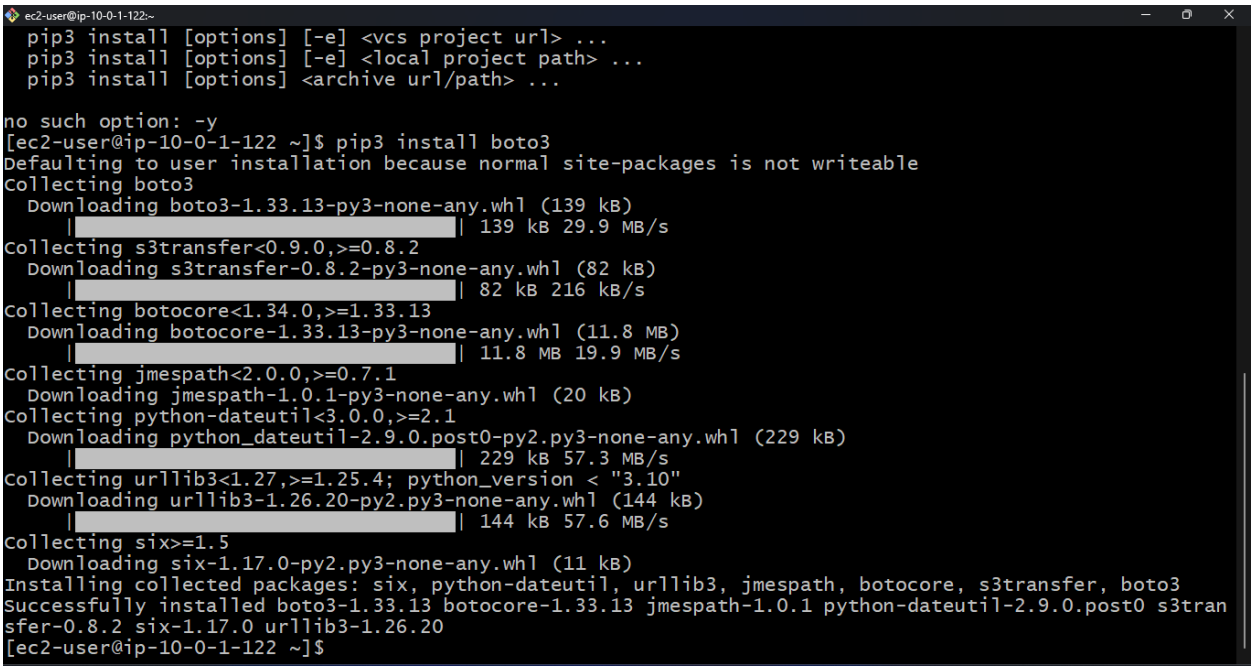
These diagrams illustrates the architecture for integrating the Student Data Logger with Amazon DynamoDB, focusing on the secure programmatic access from the EC2 instance.

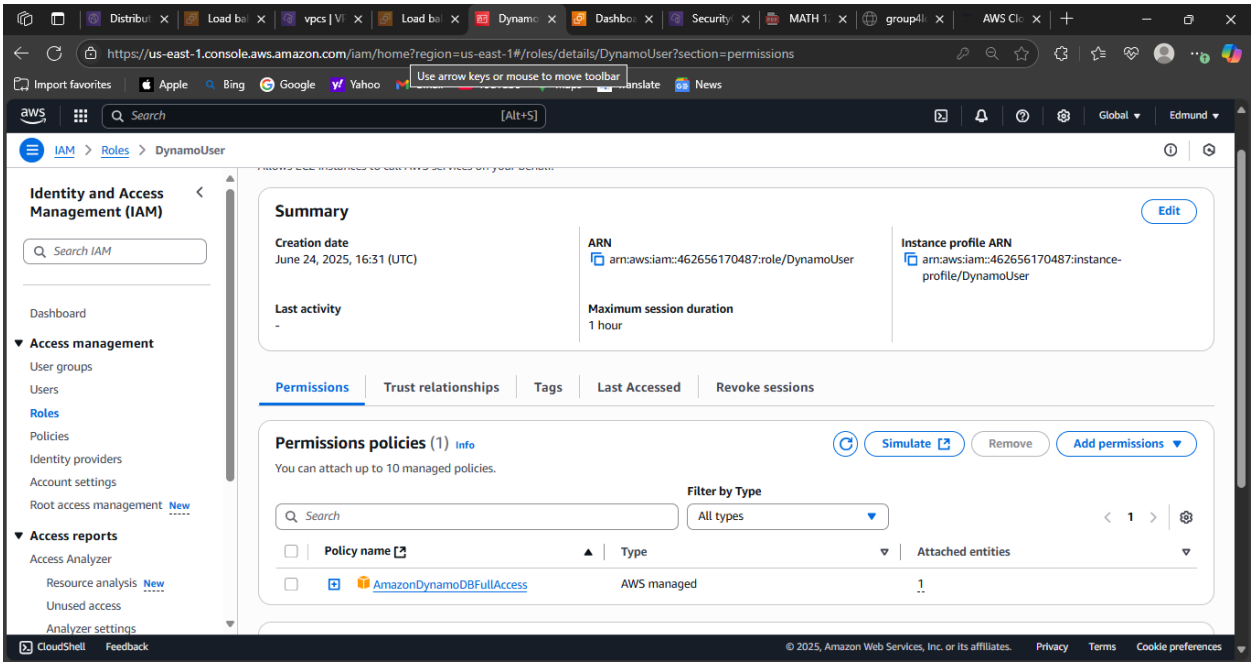


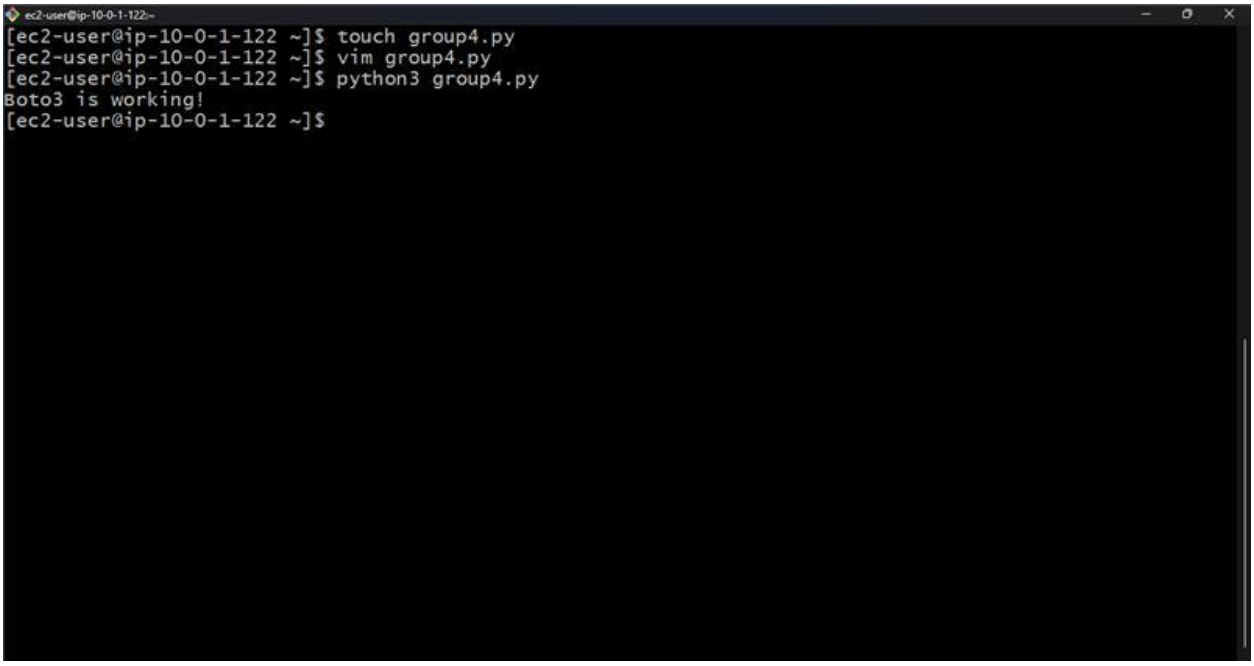

Group4-VPC
Public Subnet
EC2 Instance: Application Server
Runs Python Script with Boto3
Python Script
Assumes
IAM Role: DynamoUser
Reads/Writes Data
Grants Permissions to
AWS Managed Services
Amazon DynamoDB: Students Table

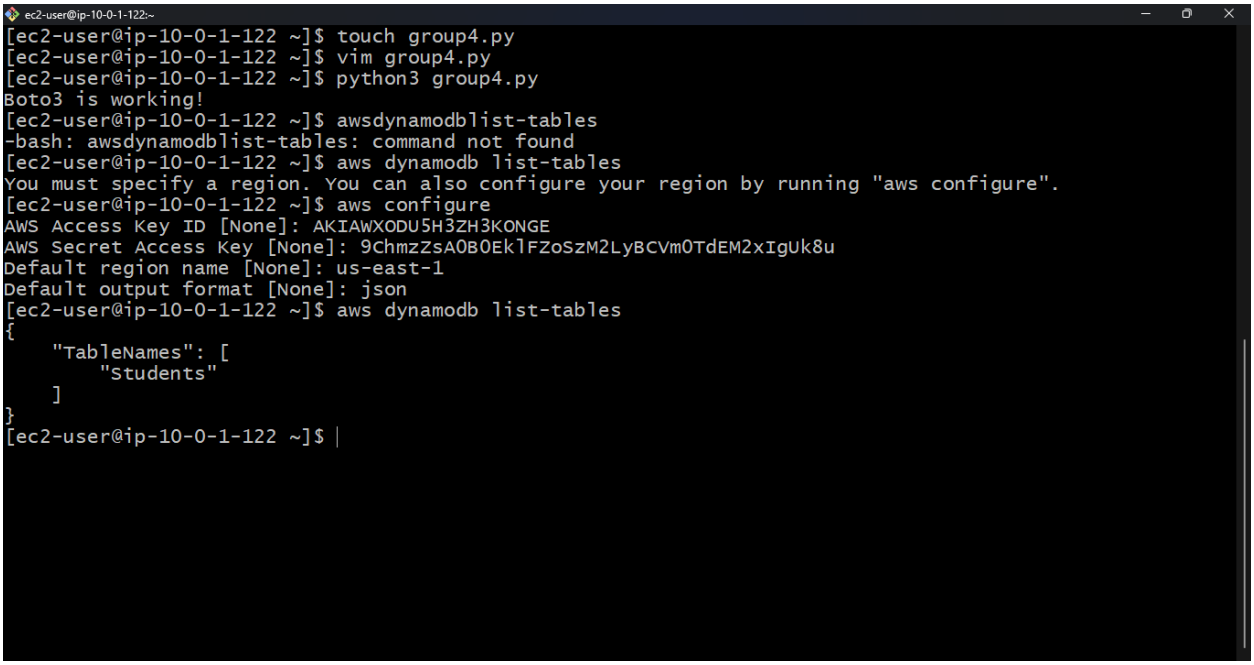
5.7 Method: Database Integration

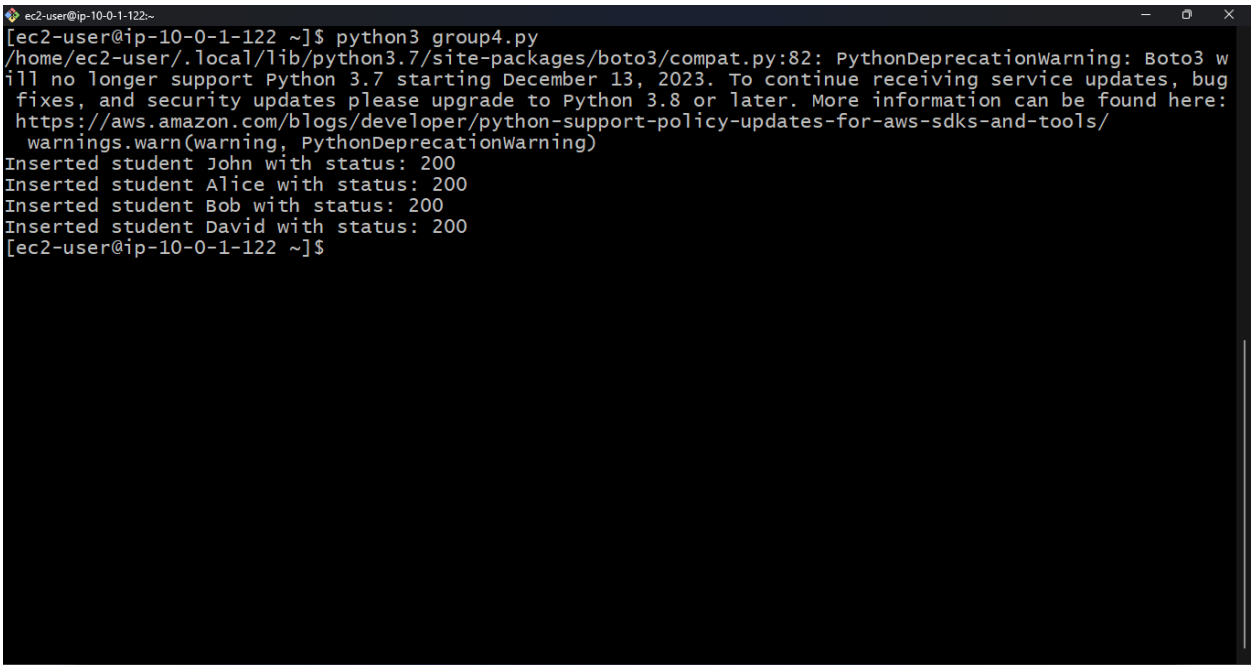


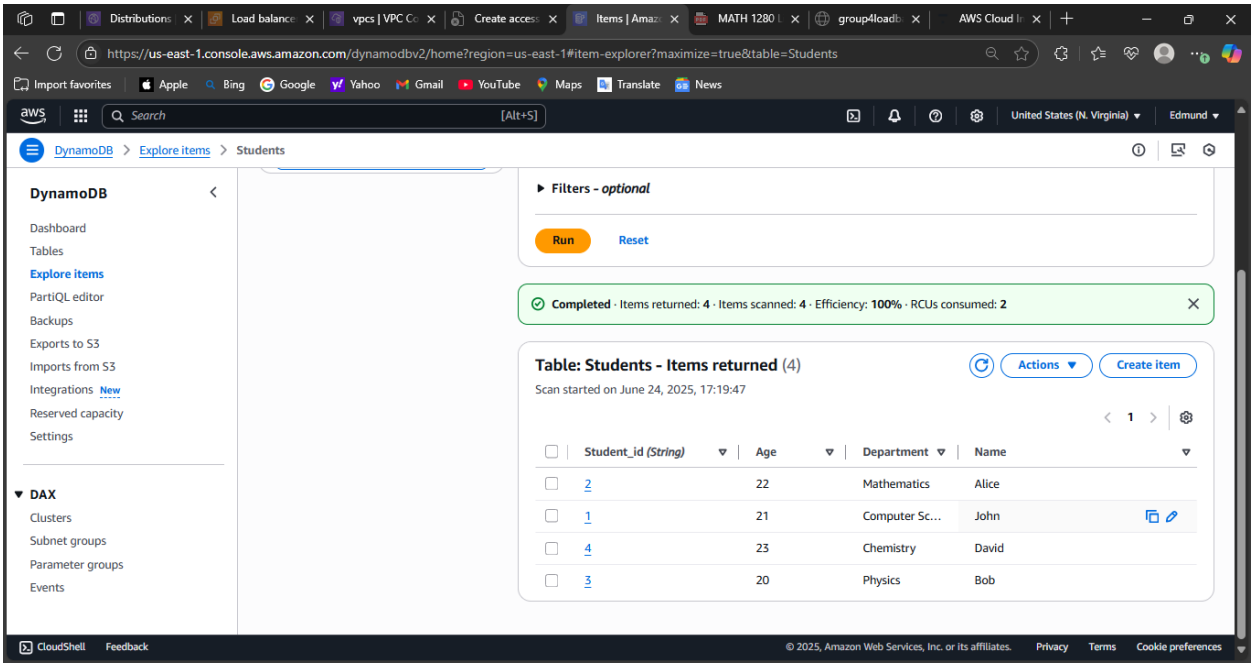



https://us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/roles/details/DynamoUser?section=permissions
Use arrow keys or mouse to move toolbar
IAM > Roles > DynamoUser
Identity and Access Management (IAM)
Search IAM
Dashboard
Access management
User groups
Users
Roles
Policies
Identity providers
Account settings
Root access management New
Access reports
Access Analyzer
Resource analysis New
Unused access
Analyzer settings
Summary
Edit
Creation date
June 24, 2025, 16:31 (UTC)
ARN
arn:aws:iam::462656170487:role/DynamoUser
Instance profile ARN
arn:aws:iam::462656170487:instance-profile/DynamoUser
Last activity
-
Maximum session duration
1 hour
Permissions
Trust relationships
Tags
Last Accessed
Revoke sessions
Permissions policies (1) Info
You can attach up to 10 managed policies.
Simulate
Remove
Add permissions
Filter by Type
Search
All types
Policy name
Type
Attached entities
AmazonDynamoDBFullAccess
AWS managed
1
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


[ec2-user@ip-10-0-1-122 ~]$ touch group4.py
[ec2-user@ip-10-0-1-122 ~]$ vim group4.py
[ec2-user@ip-10-0-1-122 ~]$ python3 group4.py
Boto3 is working!
[ec2-user@ip-10-0-1-122 ~]$


ec2-user@ip-10-0-1-122:~
[ec2-user@ip-10-0-1-122 ~]$ touch group4.py
[ec2-user@ip-10-0-1-122 ~]$ vim group4.py
[ec2-user@ip-10-0-1-122 ~]$ python3 group4.py
Boto3 is working!
[ec2-user@ip-10-0-1-122 ~]$ awsdynamodblist-tables
-bash: awsdynamodblist-tables: command not found
[ec2-user@ip-10-0-1-122 ~]$ aws dynamodb list-tables
You must specify a region. You can also configure your region by running "aws configure".
[ec2-user@ip-10-0-1-122 ~]$ aws configure
AWS Access Key ID [None]: AKIAWXODU5H3ZH3KONGE
AWS Secret Access Key [None]: 9ChmzZsAOBOEklFZoSzM2LyBCVmOTdEM2xIgUk8u
Default region name [None]: us-east-1
Default output format [None]: json
[ec2-user@ip-10-0-1-122 ~]$ aws dynamodb list-tables
{
"TableNames": [
"Students"
]
}
[ec2-user@ip-10-0-1-122 ~]$


ec2-user@ip-10-0-1-122:~
[ec2-user@ip-10-0-1-122 ~]$ python3 group4.py
/home/ec2-user/.local/lib/python3.7/site-packages/boto3/compat.py:82: PythonDeprecationWarning: Boto3 will no longer support Python 3.7 starting December 13, 2023. To continue receiving service updates, bug fixes, and security updates please upgrade to Python 3.8 or later. More information can be found here: https://aws.amazon.com/blogs/developer/python-support-policy-updates-for-aws-sdks-and-tools/
warnings.warn(warning, PythonDeprecationWarning)
Inserted student John with status: 200
Inserted student Alice with status: 200
Inserted student Bob with status: 200
Inserted student David with status: 200
[ec2-user@ip-10-0-1-122 ~]$



6. Overall Project Challenges and Learnings

Throughout the capstone project, Group 4 encountered several significant challenges that, when overcome, led to deeper learning and a more robust understanding of cloud architecture. These challenges reinforced the practical complexities of deploying and managing applications in a production-like AWS environment.

6.1 Determining Appropriate Metrics for Autoscaling

Challenge: Selecting the most effective and representative metric to trigger auto-scaling events for an application. While CPU Utilization is a common and straightforward choice, in a real-world scenario, relying solely on CPU might not always accurately reflect application load or user experience. For example, a CPU-light application might be I/O bound or memory constrained, or its performance bottleneck could be network throughput or database latency, none of which are directly captured by CPU utilization. The challenge was to balance simplicity for demonstration with the conceptual understanding of more nuanced metrics.

Solution/Learning: For the purpose of this project, CPU Utilization at 50% served as a clear and verifiable metric to demonstrate the auto-scaling mechanism. However, the team learned the importance of analyzing application-specific performance indicators. In a production setting,

one might consider:

- **Request Count per Target:** If traffic is the primary driver of load.
- **Latency:** To scale based on user experience.
- **Queue Lengths:** For message queue-based architectures.
- **Custom Metrics:** Published to CloudWatch by the application itself (e.g., number of active users, transactions per second). The learning was not just *how* to set the alarm, but *what* metrics truly matter for a given application's health and scalability, and how to effectively analyze them to predict scaling needs.

6.2 Stressing Instances for Performance Testing

Challenge: Effectively generating enough load on the EC2 instances to consistently push their CPU utilization beyond the 50% threshold, thereby triggering the auto-scaling policy, without crashing the instances or making the test environment unstable. Manual methods or simple scripts might not always create a sustained, high-enough load to reliably demonstrate scaling in a short timeframe.

Solution/Learning: We leveraged command-line tools like stress (if available on the AMI) or simple infinite loops (while true; do ... done) in shell scripts to artificially inflate CPU usage. This allowed us to observe the immediate impact on CPU metrics in CloudWatch. The learning here extended to:

- **Controlled Load Generation:** Understanding the need for more sophisticated load testing tools (e.g., Apache JMeter, Locust, k6) for accurate and repeatable stress tests in production environments.
- **Monitoring Feedback Loop:** The immediate visual feedback from CloudWatch graphs (CPU utilization spikes, alarm state changes) was crucial. It allowed us to adjust the intensity and duration of the stress tests until the scaling events were consistently triggered.
- **Observation of Scaling Effect:** Observing the CPU utilization drop after new instances were launched by the ASG provided concrete evidence that the scaling mechanism was effective in distributing the load and improving overall system performance.

6.3 Python Script Execution and Environment Management

Challenge: Ensuring the correct Python environment, including the Boto3 SDK and its dependencies, was properly set up on the EC2 instance, and that the Python script had the

necessary permissions to interact with DynamoDB. Issues could arise from missing packages, incorrect Python versions, or insufficient IAM role permissions.

Solution/Learning:

- **Systematic Installation:** Following clear steps for pip3 install boto3 and verifying the installation immediately.
- **IAM Role Validation:** The critical learning was the power and necessity of IAM roles for secure access. Initial troubleshooting often involved verifying that the IAM role was correctly attached to the EC2 instance and that the policies granted precisely the required permissions. Using aws dynamodb list-tables without explicit credentials was a powerful way to confirm the IAM role's efficacy.
- **Environment Isolation (Conceptual):** While not fully implemented for a small project, the concept of virtual environments (venv) for Python was implicitly reinforced to prevent dependency conflicts and ensure project portability.
- **Error Handling in Script:** Implementing try-except blocks in the Python script for DynamoDB operations was crucial. This allowed the script to gracefully handle API errors (e.g., AccessDeniedException, ResourceNotFoundException) and provide informative output, aiding in debugging.
- **Output Validation:** The clear output Inserted student X with status: 200 and subsequent verification in the AWS console were vital for confirming the script's successful execution and data persistence.

These challenges, rather than being roadblocks, served as valuable learning opportunities, solidifying our practical skills in cloud infrastructure deployment, automation, and security.

7. Conclusion

7.1 Summary of Achievements

This capstone project successfully demonstrated the design, implementation, and management of a secure, scalable, and highly available cloud infrastructure on Amazon Web Services. Group 4 achieved significant milestones across three interconnected phases:

1. **VPC Architecture:**
    - Designed, deployed, and secured a robust Virtual Private Cloud (VPC) architecture with an appropriate CIDR block.
    - Successfully segmented the network into multiple public and private subnets,

strategically placed across distinct Availability Zones to ensure high availability and fault tolerance.
    - Configured essential network components including an Internet Gateway for public connectivity, a NAT Gateway for secure outbound internet access from private subnets, and precise route tables to manage traffic flow effectively.
    - Implemented multi-layered security using granular Security Groups at the instance level and Network Access Control Lists (NACLs) at the subnet level, enforcing the principle of least privilege and controlled access.
2. **Autoscaling and Load Balancing:**
    - Set up Elastic Load Balancers (specifically an Application Load Balancer) to efficiently distribute incoming application traffic across multiple EC2 instances, enhancing performance and resilience.
    - Configured Auto Scaling Groups (ASGs) with dynamic scaling policies, enabling the infrastructure to automatically adjust compute capacity based on predefined metrics (CPU Utilization).
    - Validated the elasticity and high availability through simulated stress tests, observing successful scale-out events and consistent load distribution.
    - Leveraged AWS CloudWatch for comprehensive monitoring of application health, performance metrics, and detailed logging of auto-scaling activities, providing crucial insights into system behavior.
3. **Database Integration:**
    - Successfully integrated a scalable NoSQL database solution using Amazon DynamoDB, demonstrating its capabilities for fast, serverless data storage.
    - Prepared an EC2 instance as an application server, ensuring it had the necessary Python environment and Boto3 SDK installed for programmatic interaction with AWS services.
    - Implemented secure access to DynamoDB by attaching an appropriate IAM role to the EC2 instance, eliminating the need for hardcoded credentials and adhering to cloud security best practices.
    - Developed and executed a Python script to perform data insertion operations into the DynamoDB table, and rigorously verified data persistence and accuracy directly from the AWS Management Console.

7.2 Future Enhancements and Recommendations

While the project successfully met all its objectives, several enhancements could further improve the architecture and functionality for a production environment:

- **Refined IAM Policies:** Transition from FullAccess IAM policies to highly granular, least-privilege policies for DynamoDB and other services, limiting permissions to only what is absolutely necessary for the application's function.
- **VPC Endpoints for DynamoDB:** Implement VPC Endpoints for DynamoDB to ensure that all traffic between the EC2 instance and DynamoDB remains within the AWS network, improving security and reducing data transfer costs.
- **Advanced Load Testing:** Utilize dedicated load testing tools (e.g., Apache JMeter, Locust) to conduct more realistic and sustained load simulations, gathering comprehensive performance metrics for further optimization.
- **Logging and Monitoring Enhancement:** Integrate AWS CloudTrail for API call logging, CloudWatch Logs for application-level logging, and consider centralized log management with services like Amazon OpenSearch Service. Implement custom CloudWatch dashboards and alarms for critical application metrics beyond just CPU utilization.
- **Database Backup and Recovery Automation:** Implement automated backup strategies for DynamoDB (e.g., continuous backups with Point-in-Time Recovery enabled, or scheduled on-demand backups).
- **Infrastructure as Code (IaC):** Transition from manual console-based deployments to IaC tools like AWS CloudFormation or Terraform. This would enable version control, automation, reproducibility, and easier management of the entire infrastructure.
- **Application Deployment Automation:** Integrate Continuous Integration/Continuous Delivery (CI/CD) pipelines (e.g., AWS CodePipeline, Jenkins) for automated code deployment to EC2 instances, ensuring faster and more reliable updates.
- **Security Auditing:** Regularly review AWS Security Hub findings and implement AWS Config rules for continuous compliance monitoring.
- **Cost Optimization:** Implement EC2 Reserved Instances or Savings Plans for predictable workloads, and fine-tune ASG policies to scale in more aggressively during off-peak hours to optimize costs.

7.3 Personal and Team Learnings

This capstone project was an invaluable learning experience that extended beyond theoretical knowledge:

- **Deepened Understanding of AWS Ecosystem:** Gained hands-on proficiency with a wide array of AWS services and how they interoperate to form a cohesive cloud solution.
- **Practical Application of Cloud Principles:** Applied core cloud concepts such as high availability, fault tolerance, elasticity, network segmentation, and defense-in-depth security in a practical setting.
- **Problem-Solving and Troubleshooting:** Developed critical troubleshooting skills by systematically identifying and resolving configuration issues, network connectivity problems, and application errors.
- **Importance of Documentation and Verification:** Learned the value of meticulous planning, step-by-step implementation, and thorough verification at each stage of the project.
- **Collaboration and Communication:** Enhanced teamwork abilities, fostering effective communication, task division, and collective problem-solving within the group.
- **Adaptability to Cloud Dynamics:** Understood that cloud environments are dynamic and require continuous monitoring, adaptation, and optimization.

The successful completion of this project has equipped Group 4 with practical skills and confidence in designing, deploying, and managing robust cloud infrastructures, preparing us for future challenges in cloud computing.

8. Appendices

8.1 AWS Services Utilized

- **Amazon VPC (Virtual Private Cloud):** For creating an isolated virtual network.
- **Amazon EC2 (Elastic Compute Cloud):** For scalable compute capacity (virtual servers).
- **Amazon S3 (Simple Storage Service):** (Implicit, for storing objects like instance boot scripts, or potentially data if not using DynamoDB exclusively).
- **Amazon Elastic Load Balancing (ELB) - Application Load Balancer (ALB):** For distributing incoming application traffic.
- **Amazon EC2 Auto Scaling:** For automatic scaling of EC2 instances based on demand.
- **Amazon CloudWatch:** For monitoring and logging.
- **AWS Identity and Access Management (IAM):** For managing access to AWS services and resources.
- **Amazon DynamoDB:** For fully managed NoSQL database service.

- **AWS CLI (Command Line Interface):** For interacting with AWS services from the command line.
- **Boto3 (AWS SDK for Python):** For programmatic interaction with AWS services using Python.

8.2 Glossary of Terms

- **Availability Zone (AZ):** One or more discrete data centers with redundant power, networking, and connectivity in an AWS Region.
- **Auto Scaling Group (ASG):** A collection of EC2 instances treated as a logical grouping for automatic scaling and management.
- **Boto3:** The Amazon Web Services (AWS) SDK for Python.
- **CIDR (Classless Inter-Domain Routing):** A method for allocating IP addresses and routing IP packets.
- **CloudWatch:** An AWS service that monitors AWS resources and applications.
- **CRUD Operations:** Create, Read, Update, Delete - fundamental operations for persistent storage.
- **DynamoDB:** A fully managed NoSQL database service from AWS.
- **EC2 (Elastic Compute Cloud):** A web service that provides resizable compute capacity in the cloud.
- **Elastic IP:** A static, public IPv4 address designed for dynamic cloud computing.
- **Elastic Load Balancer (ELB):** Distributes incoming application traffic across multiple targets.
- **High Availability (HA):** A characteristic of a system designed to ensure a high level of operational performance for a given period.
- **IAM (Identity and Access Management):** Controls who is authenticated and authorized to use resources.
- **Internet Gateway (IGW):** A horizontally scaled, redundant, and highly available VPC component that allows communication between instances in your VPC and the internet.
- **Launch Template:** Specifies the configuration for launching an EC2 instance.
- **NACL (Network Access Control List):** An optional layer of security that acts as a firewall for controlling traffic in and out of one or more subnets.
- **NAT Gateway (Network Access Translation Gateway):** Enables instances in a private subnet to connect to the internet or other AWS services, but prevents the internet from

initiating connections with those instances.

- **Partition Key:** A component of the primary key in DynamoDB that determines the logical partition in which a data item is stored.
- **Route Table:** Contains a set of rules, called routes, that are used to determine where network traffic from your subnet or gateway is directed.
- **Security Group:** Acts as a virtual firewall for your EC2 instances to control inbound and outbound traffic.
- **SSH (Secure Shell):** A cryptographic network protocol for operating network services securely over an unsecured network.
- **Subnet:** A range of IP addresses in your VPC.
- **Target Group:** Used by a load balancer to route requests to one or more registered targets.
- **VPC (Virtual Private Cloud):** An isolated section of the AWS cloud where you can launch AWS resources in a virtual network that you define.