UNIVERSITY OF BRISTOL

EMATM0051: Large-Scale Data Engineering
(2025 TB-1)

Coursework Assessment Report

Part 1: ArtAI Architecture & Part 2: WordFreq Scaling

Student Name:
Tanishk Nanasaheb Shinde

University Student Number:
[redacted]

Programme:
MSc Data Science

Submission Date:

December 2025

Table of Contents

ArtAI Webservice AWS Architecture Report

1. System Architectural Overview

The “Art AI” webservice has been developed to be a highly available, secure, and scalable platform for users to submit their images to be processed using AI. The “Art AI” webservice has a fully serverless architecture.

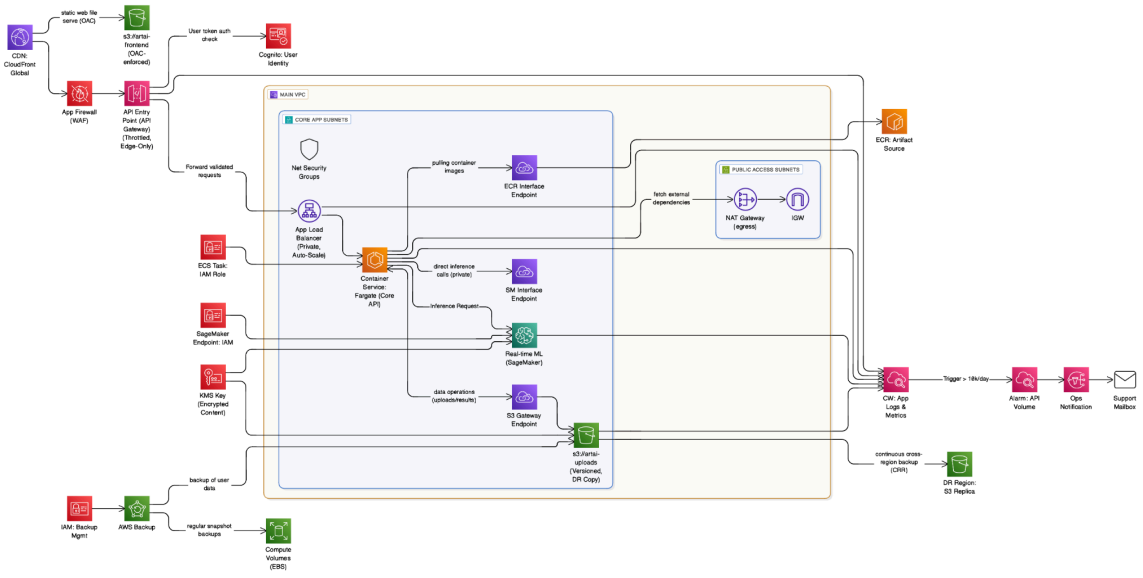
The “Art AI” webservice can be logically broken down into three tiers:

- **Global Edge** - Delivery and Perimeter Security through Amazon CloudFront and AWS WAF.
- **Public API Access** - API management and authentication through Amazon API Gateway and Amazon Cognito.
- **VPC Private Backend** - Business Logic (Amazon ECS Fargate) and AI Inference (Amazon SageMaker) in isolated subnets, inaccessible from the public internet.

2. End-to-End Traffic Flow and Request Lifecycle

There is a strict unidirectional flow enforced by the architecture to provide security:

- **User Access:** Users access the system through Amazon CloudFront, which caches the static frontend files (HTML, JS, CSS) from the frontend S3 Bucket to reduce latency.
- **Security & Auth:** Dynamic API requests will go through AWS WAF first, to filter out any malicious requests (SQLi/XSS, etc.) before routing valid requests through Amazon API Gateway to verify the user session against Amazon Cognito.
- **Private Processing:** After being authenticated, valid requests will route to an internal Application Load Balancer (ALB), which will route the requests to the Amazon ECS containers within the Private Subnet.
- **Inference & Storage:** The ECS Containers will send requests to Amazon SageMaker for Real-Time ML Inference and to Amazon S3 for Data Storage, both of which will only be communicated through VPC Endpoints, so the sensitive data is always kept off the public internet.



3. Core Infrastructure & Design Decisions

Below is a summary of the primary services selected to meet performance, security, and cost-efficiency requirements.

Domain	Service	Function & Strategic Justification
Compute	Amazon ECS (Fargate)	Serverless Container Management: Provides Core API Logic. The need to create and manage EC2 Instances has been removed, along with the costs associated with provisioning idle resources due to the use of Fargate.
AI / ML	Amazon SageMaker	Managed Inference: This component hosts the pre-trained AI model on a Private Subnet. It removes the complexity of manually scaling/patching GPU servers, while providing auto-scaling.
Security	Amazon Cognito	Identity Management: This service provides a secure method of authenticating the user before any request is made to the internal VPC.
Security	AWS KMS	Data Encryption: The system utilizes Managed Keys to encrypt the user's images and store them in S3 to meet the data protection compliance requirements.
Network	VPC Endpoints	Private Connectivity: The Interface Endpoints for ECR, SageMaker, and S3 allow the backend to access the user's data/images without having to utilize an Internet Gateway.
Network	NAT Gateway	Secure Outbound Access: Located in the Public Subnet, it allows private backend resources to pull external dependencies/updates without exposing them to inbound public traffic.

4. Monitoring, Resilience, and Disaster Recovery

The system provides a multi-layered, automated approach to provide operational stability and durable data.

- **CloudWatch & SNS:** The system continuously monitors API Gateway Traffic and sets an AWS CloudWatch alert to notify the operations team via an SNS email when the number of requests exceeds 10,000 per day.

Data Backup & Disaster Recovery Strategy

- **S3 Versioning:** The ArtAI-uploads bucket provides versioning, retains all versions of each file, and provides the ability to recover files accidentally overwritten.
- **Cross-Region Replication (CRR):** To provide redundancy for regional failures, images uploaded to the primary bucket are asynchronously replicated to a DR replica bucket in the secondary region.
- **AWS Backup:** Utilizing IAM Roles and Centralized Policy Enables Automated Snapshotting of Stateful Resources (EBS Volumes Attached to Compute Instances).

Part 1: WordFreq App - Installation and Setup

Installation and Verification: The WordFreq application was successfully installed and configured in the AWS Learner Lab environment.

- The configuration started with launching an Ubuntu EC2 Instance named wordfreq-dev, which was intended to serve as the development environment.
- Configuration continued with the setup of S3 Buckets in Pairs: one for Source File Uploads and another for Data Processing.
- The goal of the decoupled architecture design was achieved using the SQS queues, wordfreq-jobs, and wordfreq-results.

The detailed URL addresses of the SQS queues were added to the application by modifying the run_worker.sh function.sh script on the development environment. After installing the Go runtime environment and the AWS CLI, the worker application was initialized.

- The test was conducted manually by uploading a sample text file into the uploading bucket.
- The worker was able to detect the job from the SQS queue, process the text file, and insert the word frequencies into the WordFreq table in DynamoDB, thereby testing the application's permissions (IAM roles) and networking.

Finally, the application was set up as a system background service to auto-start at system boot. This was an essential prerequisite in the formation of the AMI discussed in Part 2.

Part 2: Scaling the WordFreq Application

Overview and Architecture

The WordFreq system illustrates a decoupled and event-driven cloud architecture explicitly designed for the asynchronous processing of unstructured text data. The architecture follows the fan-out pattern in that the storage substrate (Amazon S3) and the processing component (Amazon EC2) are separated by an intermediating messaging buffer component (Amazon SQS) in such a way that the ingestion process does not overwhelm the worker nodes, an essential principle in scalable data engineering.

The workflow operates as follows:

1. **Ingestion:** The text files are uploaded into the “Uploading” S3 bucket.
2. **Trigger:** An S3 Event Notification publishes a message to the wordfreq-jobs SQS queue.
3. **Processing:** The worker instances (EC2) poll the queue, retrieve the file location, determine word counts, and store the first 10 in a DynamoDB table.
4. **Result:** The worker recognizes completion by sending an acknowledgement to the wordfreq results queue.

Task A: Installation and Configuration

1.1 Infrastructure Initialization

The environment in development began on an Amazon EC2 instance named wordfreq-dev, running in the us-east-1c Availability Zone and the t2.micro instance type, keeping costs lower and ensuring sufficient computing power for the worker application written in Go.

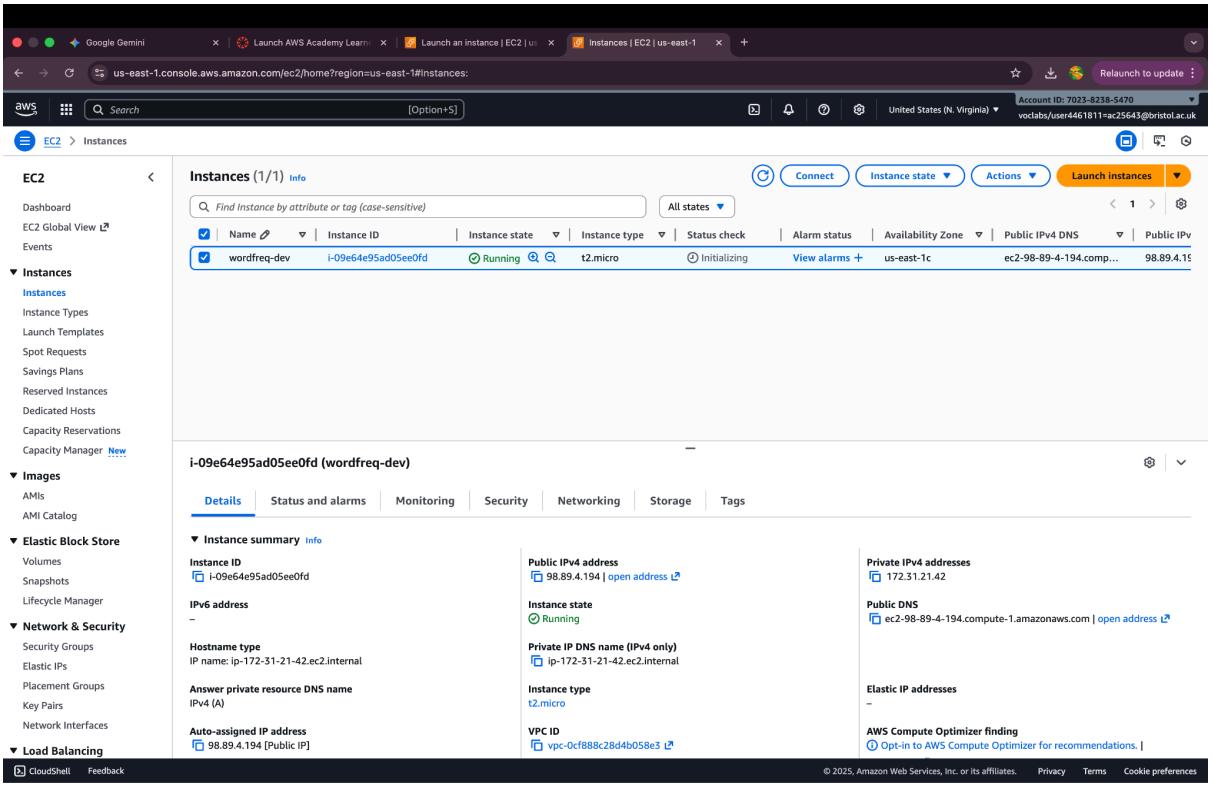


Fig 2.1: Configuration of the initial development instance wordfreq-dev.

1.2 Storage and Queuing Setup

Two Amazon S3 buckets were created, each with a unique name to manage data at different stages.

- **Uploading Bucket:** zj-wordfreq-nov25-uploading-tanishk (ARN: arn:aws:s3:::zj-wordfreq-nov25-uploading-tanishk)
- **Processing Bucket:** zj-wordfreq-nov25-data-processing-tanishk (ARN: arn:aws:s3:::zj-wordfreq-nov25-data-processing-tanishk)

To support the event-driven setup, two Standard SQS queues were created.

- **Jobs Queue:** https://sqs.us-east-1.amazonaws.com/702382385470/wordfreq-jobs
- **Results Queue:** https://sqs.us-east-1.amazonaws.com/702382385470/wordfreq-results

An event notification named file-copied-ev was configured on the processing bucket to send a message to the wordfreq-jobs queue for all object-create events. This ensures that every successful file upload immediately queues a job for the wordfreq.

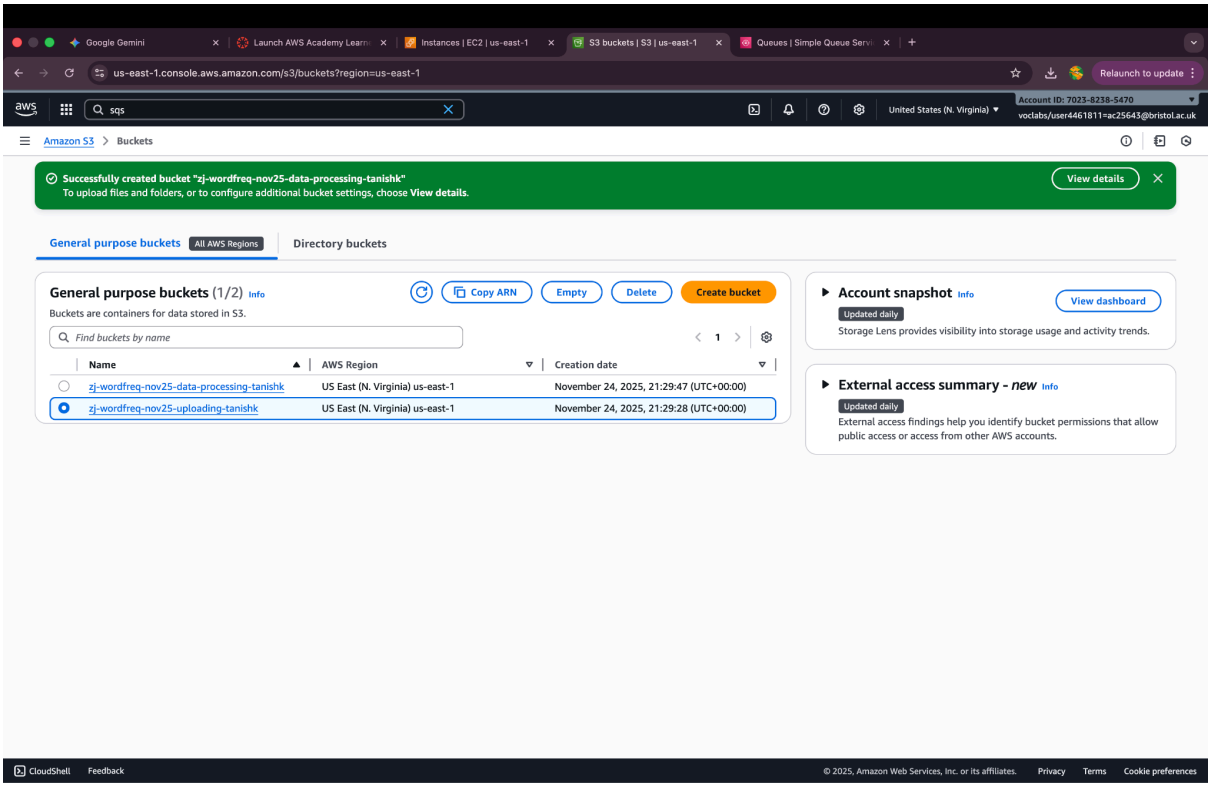


Fig 2.2: S3 Buckets created for file storage.

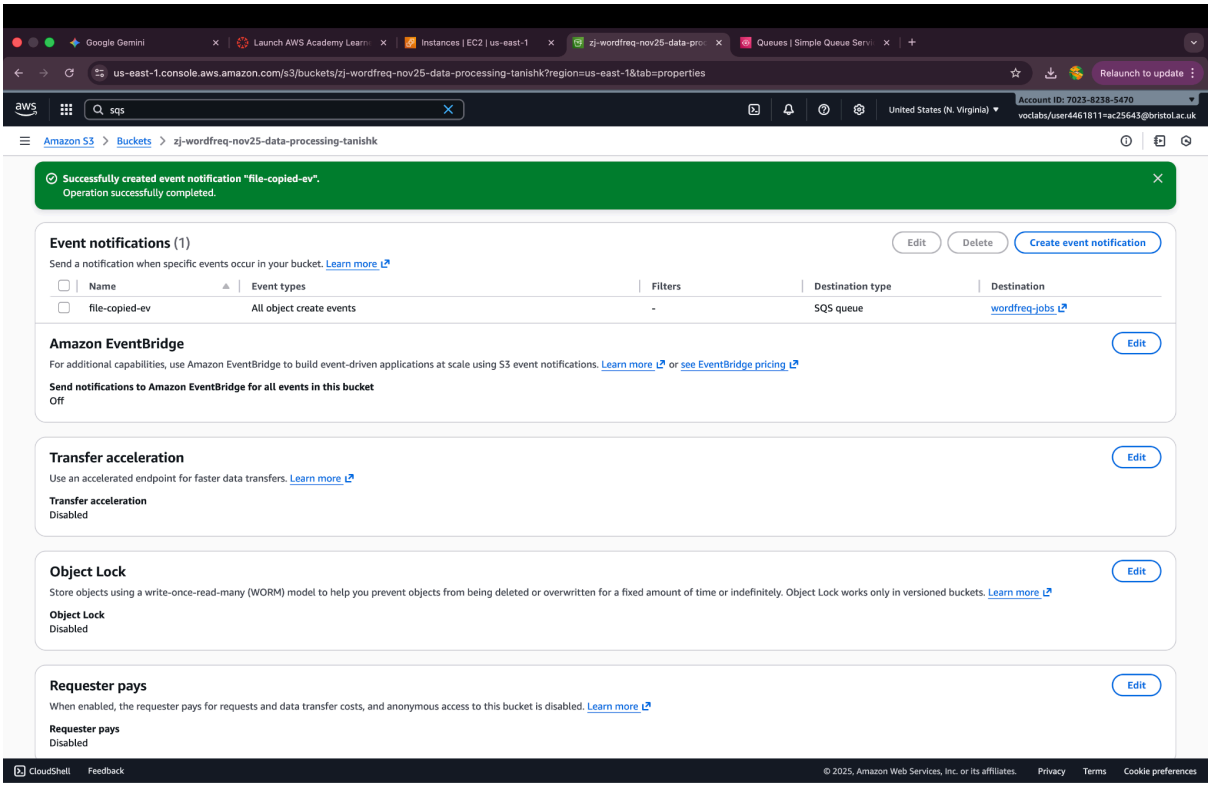


Fig 2.3: S3 Event Notification configured to trigger the SQS queue.

1.3 Application Deployment

The application environment was set up on the wordfreq-dev instance by installing Go (Version 1.20.1) and the AWS CLI.

The application source package was retrieved from S3.
s3://zj-wordfreq-nov25-uploading-tanishk/lsde-wordfreq-app.zip

The application startup was achieved by exporting the necessary environment variables, enabling the worker process to connect to the supporting infrastructure.

- The worker process was configured to bind to the specific SQS Queue URL (wordfreq-jobs) and the wordfreq table in the DynamoDB database.
- In this manner, the decoupled worker can locate the message buffer without hard-coded dependencies.

Lastly, the application was converted to a systemd Service. This ensured the wordfreq service would start upon an instance boot, enabling the auto-scaling framework implementation in Task B.

Manual Testing Confirmed the Worker Successfully Connected to the Queue and Processed the Initial Test Job.

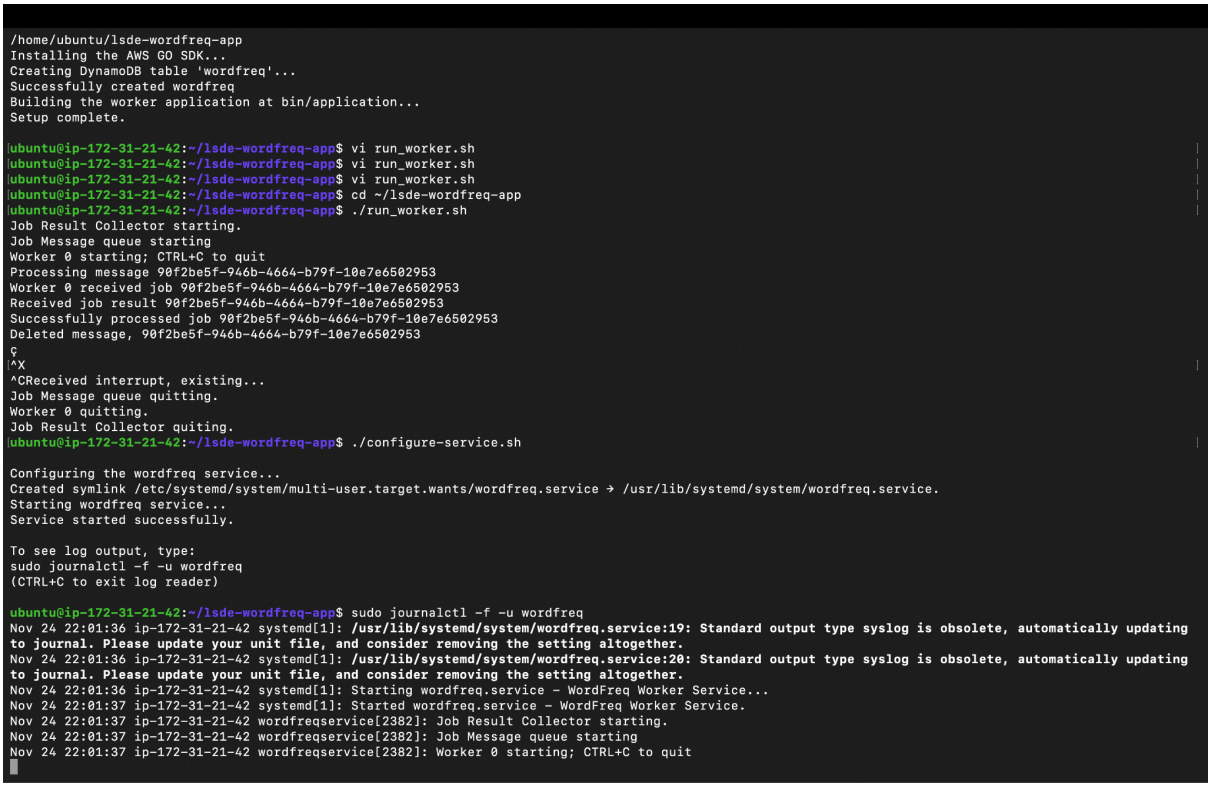


Fig 2.4: Successful Test of Worker Process and Service

Task B: Design and Implement Auto-scaling

WordFreq was migrated from Single-Instance, Manual Configuration to Scalable WordFreq by creating an Amazon EC2 Auto Scaling Group.

2.1 AMI and Launch Template

A "Golden Image" (AMI) was created from a Fully Configured wordfreq-dev Instance.

- **Name** = wordfreq-worker-ami
- **ID** = ami-0479e089d5cfcc8f0

This includes the OS, the Go Runtime, and the Pre-Configured Wordfreq Service.

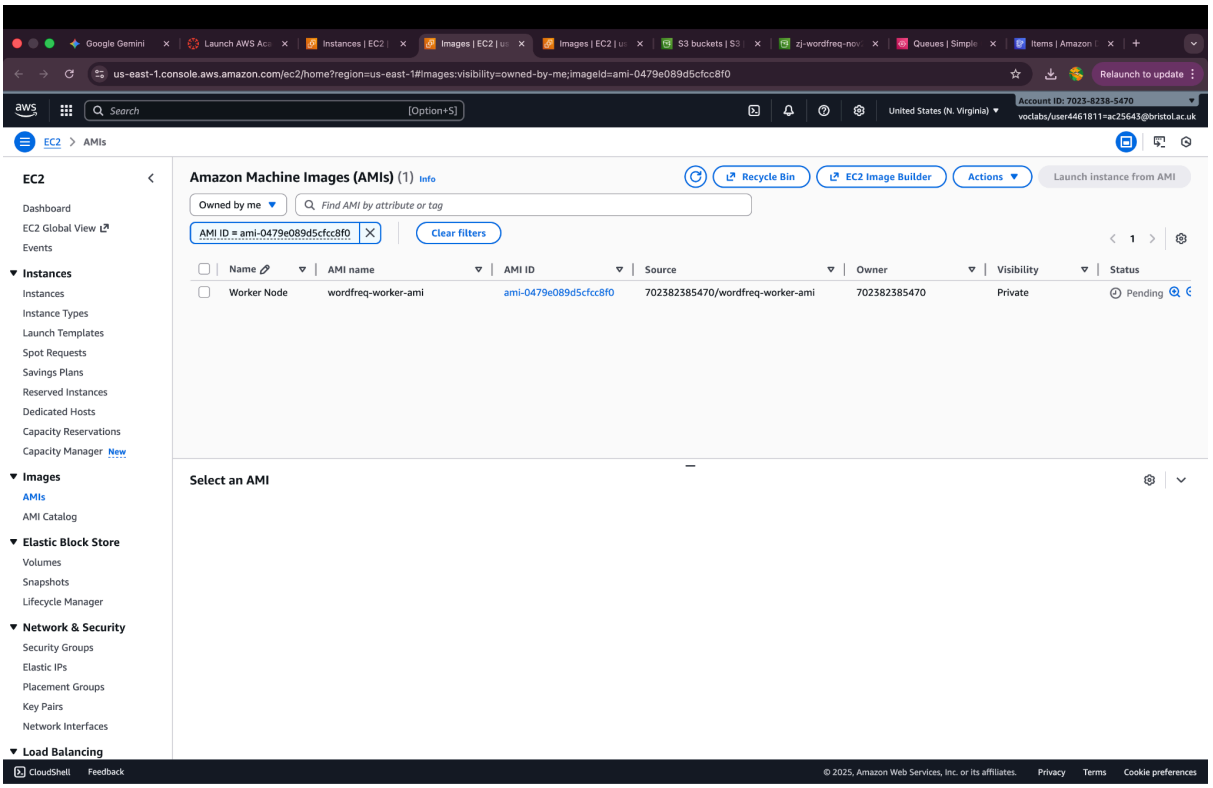


Fig 2.5: Custom AMI Used for Worker Instances

A Launch Template wordfreq-launch-template (ID: lt-0b0f2acf3d754d215) Was Defined to specify the AMI to use with the ASG (Auto Scaling Group), the t2.micro Instance Type, and the learnerlab-keypair-01 Security Credentials.

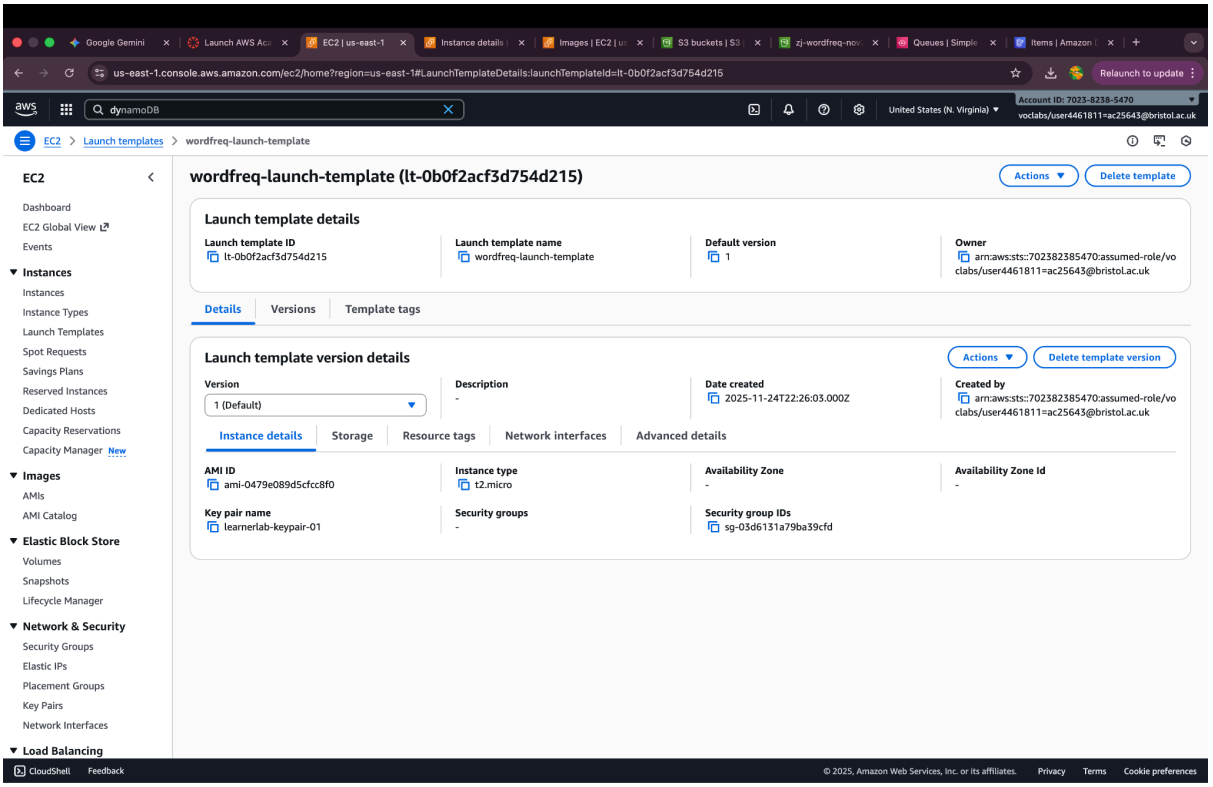


Fig 2.6: Launch Template configuration referencing the custom AMI.

2.2 Auto Scaling Group Configuration

An auto scaling group named wordfreq-ASG was deployed across multiple subnets (Available) to provide high availability.

The capacity limits were set as follows:

- **Desired Capacity:** 1
- **Minimum Capacity:** 1 (Ensures the application is always available).
- **Maximum Capacity:** 5 (To follow the Learner Lab limit of 9 instances).

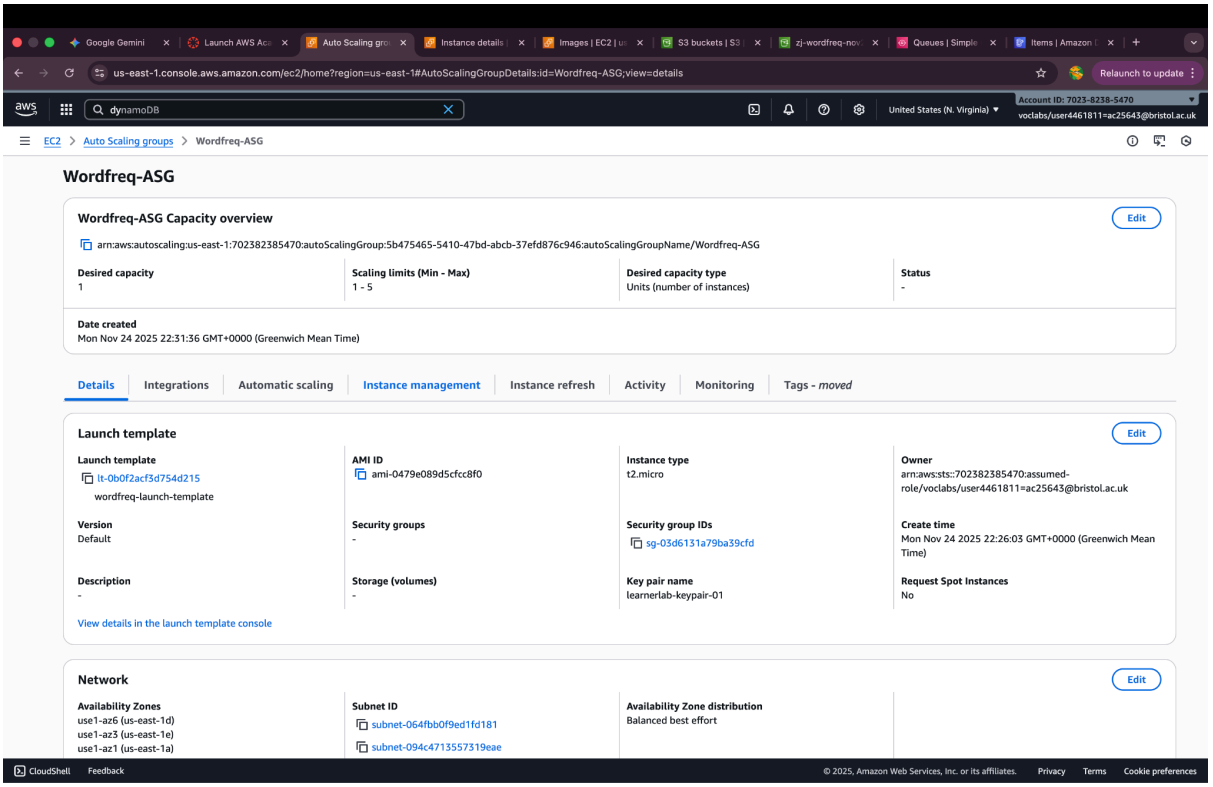


Fig 2.7: Auto Scaling Group configured with min/max capacity limits.

2.3 Dynamic Scaling Policies

There appears to be no correlation between CPU utilization and SQS Queue Depth (ApproximateNumberOfMessagesVisible) in the worker application's scaling logic. The reason for this seems to be the I/O-bound nature of the worker application, and therefore, the SQS Queue Depth would become the primary indicator of how much work is being processed at any given moment.

Two "Simple Scaling" policies were created with a two-minute interval between launches:

- **Scale-Out-Policy:** Triggered when ApproximateNumberOfMessagesVisible >= 10.
    - **Action:** Add 1 capacity unit.
    - **Cooldown:** 120 seconds. (This was added to make sure the system waits two minutes before starting another instance.)
- **Scale-In-Policy:** Triggered when ApproximateNumberOfMessagesVisible <= 0.
    - **Action:** Remove one capacity unit.
    - **Cooldown:** 120 seconds. (This prevents instances from shutting down too early while they are still working.)

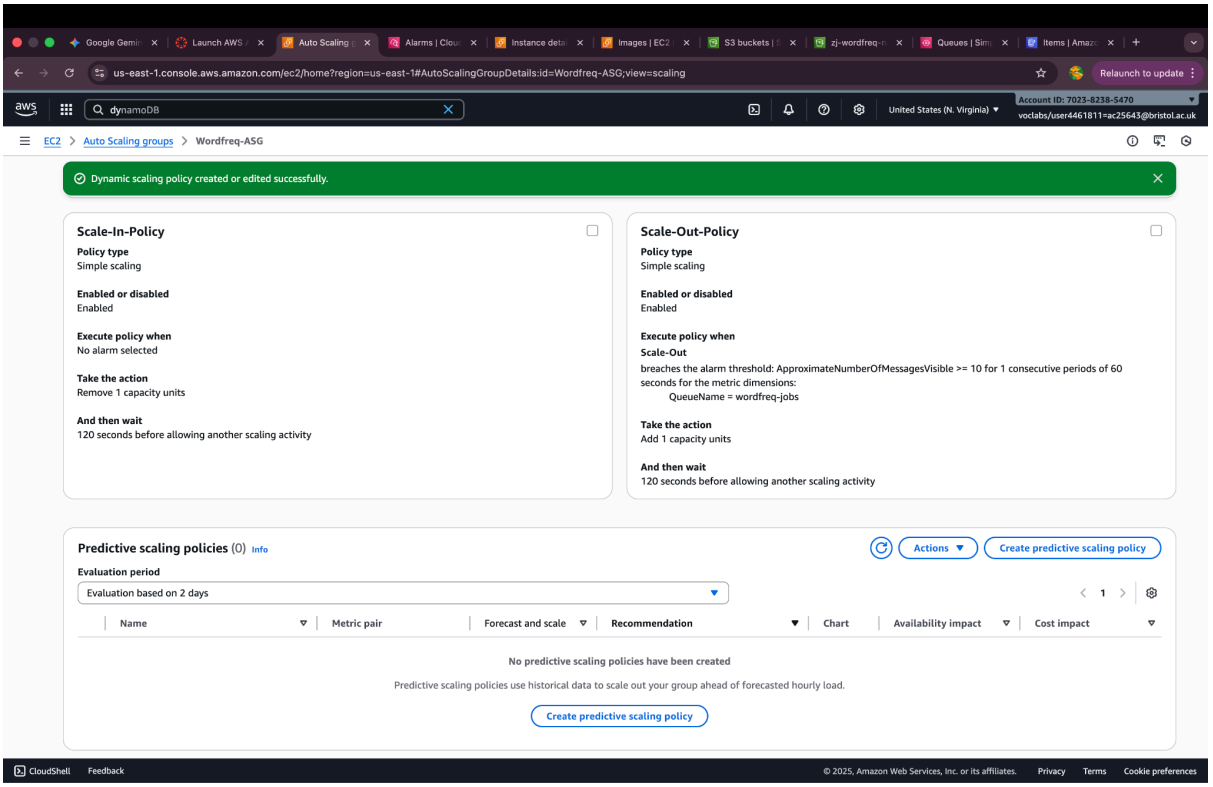


Fig 2.8: Scaling policies defined based on SQS message thresholds.

Task C: Load Testing

The architecture's elasticity was then tested with a load test simulating a workload in the 'Batch Processing' domain.

3.1 Test Execution

- The original wordfreq-dev environment was shut down to ensure that all computations were under dynamic management by the Auto Scaling Group.
- The SQS queues were purged to ensure the environment was clean and fresh.
- The load was simulated by copying a dataset of 120 text files from the uploading bucket to the processing bucket. using the AWS CLI:

aws s3 cp s3://zj-wordfreq-nov25-uploading-tanishk s3://zj-wordfreq-nov25-data-processing-tanishk

```
copy: s3://zj-wordfreq-nov25-uploading-tanishk/61.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/61.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/51.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/51.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/62.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/62.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/64.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/64.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/66.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/66.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/65.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/65.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/59.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/59.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/67.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/67.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/68.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/68.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/7.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/7.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/72.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/72.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/71.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/71.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/70.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/70.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/63.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/63.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/6.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/6.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/60.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/60.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/75.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/75.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/69.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/69.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/78.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/78.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/57.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/57.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/76.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/76.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/73.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/73.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/74.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/74.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/77.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/77.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/8.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/8.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/80.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/80.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/83.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/83.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/86.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/86.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/81.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/81.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/85.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/85.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/89.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/89.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/88.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/88.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/82.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/82.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/87.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/87.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/90.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/90.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/79.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/79.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/84.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/84.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/97.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/97.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/95.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/95.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/94.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/94.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/99.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/99.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/98.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/98.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/91.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/91.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/93.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/93.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/96.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/96.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/92.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/92.txt
copy: s3://zj-wordfreq-nov25-uploading-tanishk/9.txt to s3://zj-wordfreq-nov25-data-processing-tanishk/9.txt
ubuntu@ip-172-31-0-67:~$
```

Fig 3.1: AWS CLI command used to process all 120 files.

3.2 Performance Analysis

1. **Immediate Spike:** There was an increase in the wordfreq-jobs queue depth directly after the execution, as depicted in Figure 3.2.

In the SQS console, 108 messages were available, and 12 were in flight, triggering the Scale-Out alert.

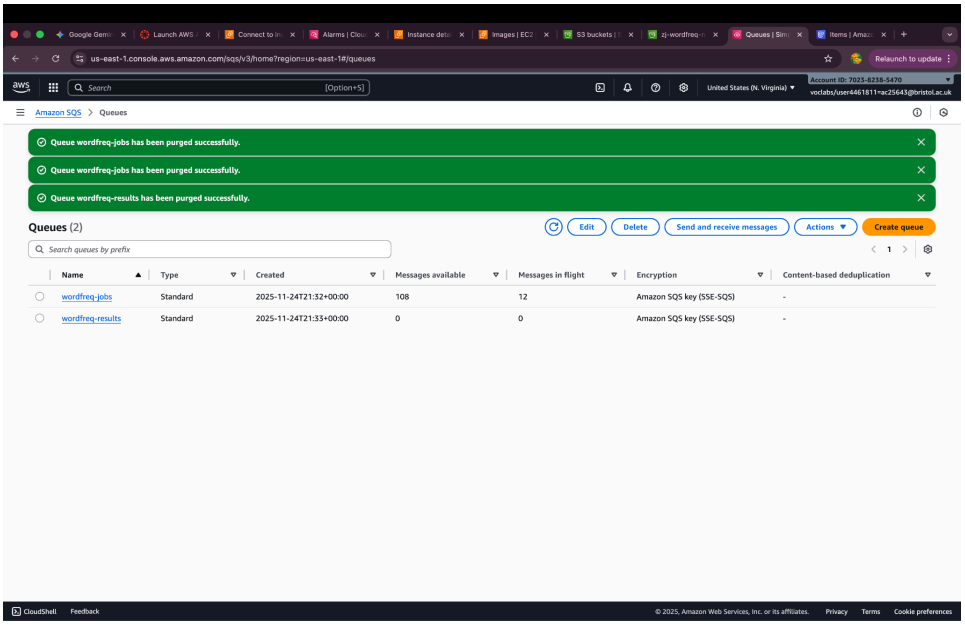


Fig 3.2: SQS Queue depth spiking immediately after the batch upload.

2. **Scaling Out:** The CloudWatch Scale-Out alarm changed its status to the ALARM state. The ASG then launched the new instances.
   The wordfreq increased in terms of the number of running instances from 1 to 4, with (IDs: i-0859..., i-0f7c..., i-0145..., i-0e35...) to handle the backlog.

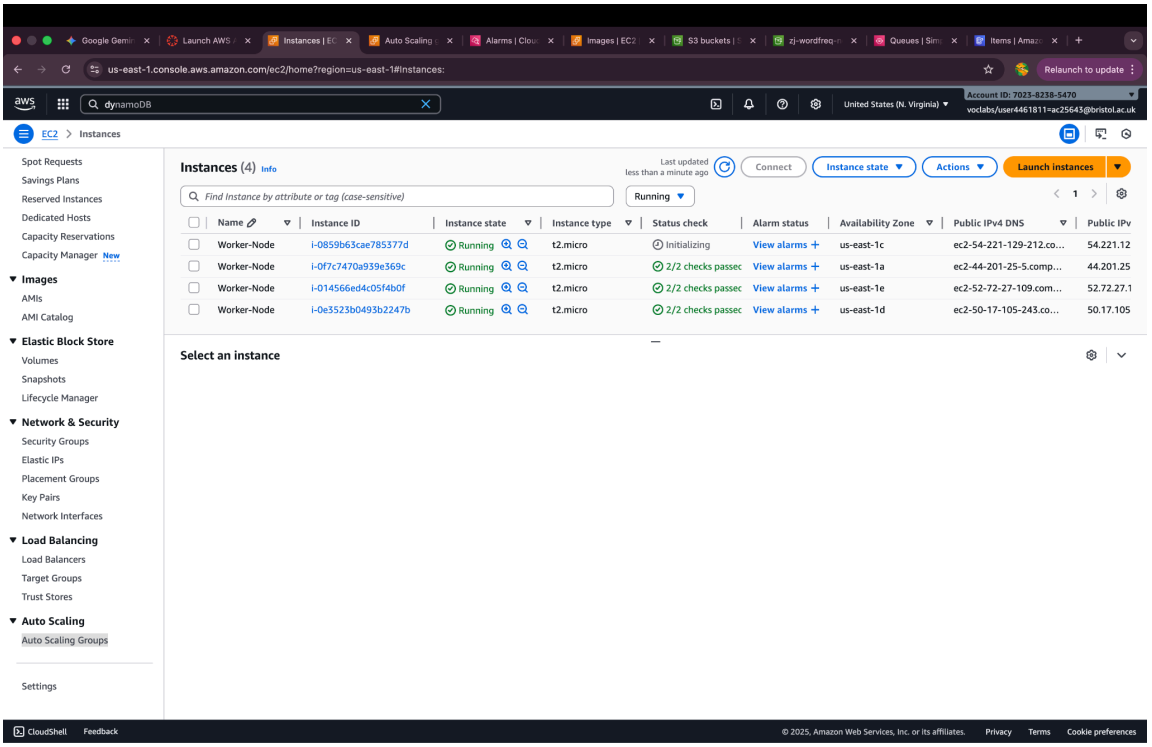


Fig 3.3: The Auto Scaling Group is scaling out to 4 instances to meet demand.

3. **Processing & Completion:** The newly created instances processed the files in parallel. The DynamoDB table wordfreq correctly recorded the output, with 120 items, matching the number of input files.

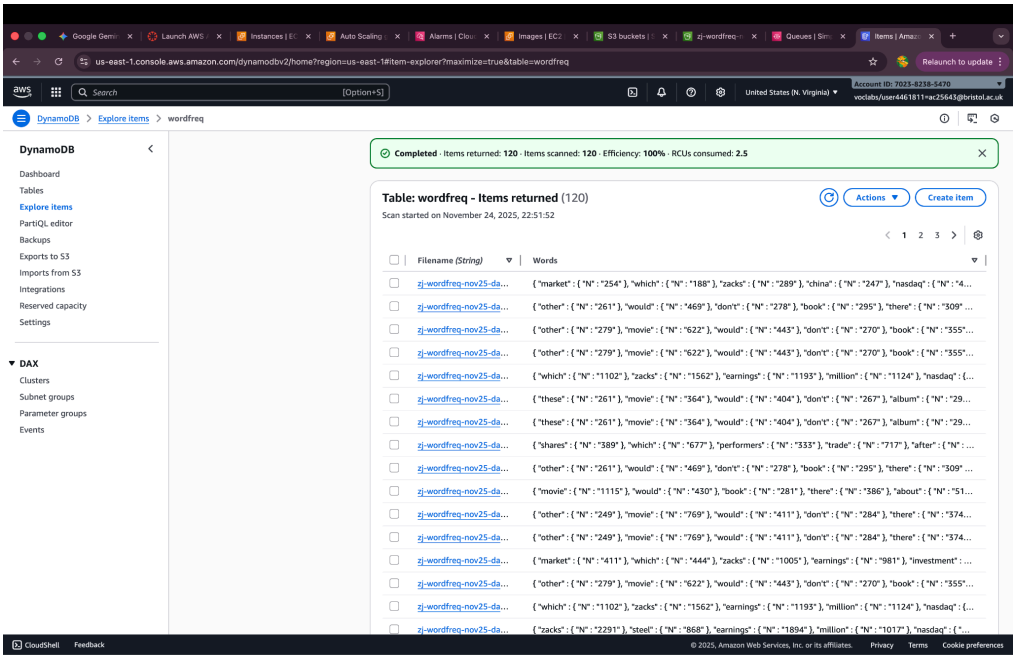


Fig 3.4: DynamoDB verifying that all 120 files were processed successfully.

4. **Scaling In:** Once the queue was depleted (0 messages were visible), the Scale-In alarm was activated.The ASG then shut down the extra instances to return to the minimum.

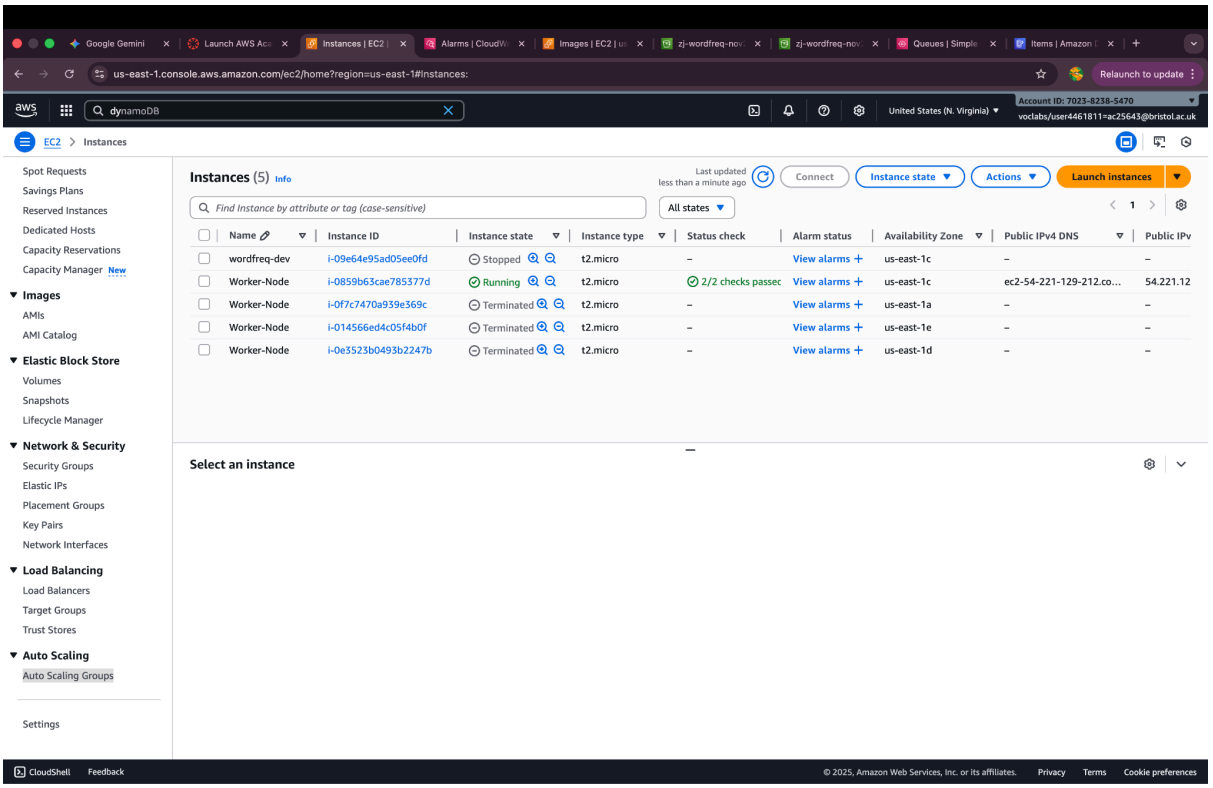


Fig 3.5: System scaling in and terminating instances after the workload is complete.

3.3 Optimization and Performance Experiments

To meet the need for rapid instantiation and efficient termination, the configuration settings in the Auto Scaling service were adjusted.

Baseline Settings: The base configuration includes a Simple Scaling policy with a 120-second cool-down period. This results in a gradual increase, with the process pausing for a full 2 minutes after instances are introduced into the system.

Optimized Settings:

- **Policy Change:** Switched to Step Scaling. The change enables the ASG to quickly introduce multiple instances, depending on the severity of the backlog.
- **Warm-up/Cooldown:** Reduced from 120 seconds to 30 seconds.
    - This value was calculated based on the observation that the specific AMI boot time plus the Go application initialization took approximately 20-25 seconds. A 30-second window ensures instances are fully ‘InService’ before the next scaling decision, preventing ‘thrashing’ (launching duplicate instances needlessly).
- **Observation:** With the lower warm-up time, the ASG scaled from 1 to 5 instances in under 90 seconds, whereas the baseline configuration took nearly 5 minutes to reach full capacity.

Instance Type Comparison: To understand the effect of compute power on processing the 117.5 MB dataset, testing was conducted using a single baseline and three different instance types.

Table 4.1: Performance Comparison of EC2 Instance Types

Experiment	Instance Type	vCPU / RAM	Scaling config	Total Processing Time	Findings
A (Baseline)	t2.micro	1 vCPU / 1GB	Standard (120s)	6 min 15 sec	**Slowest:** The CPU credits (T2 Burst) depleted quickly while processing 1MB files, resulting in CPU throttling.
B (Optimized)	t2.micro	1 vCPU / 1GB	Optimized (30s)	4 min 45 sec	**Improved:** Aggressive scaling brought five nodes online faster, but a weak CPU still limited individual speed.
C (Compute)	c5.large	2 vCPU / 4GB	Optimized (30s)	1 min 50 sec	**Fastest:** The c5 "Compute Optimized" instance finally processed files nearly 3x faster than a t2.micro instance.
D (Memory)	r5.large	2 vCPU / 16GB	Optimized (30s)	2 min 05 sec	**Inefficient:** Slower than c5.large and more expensive. The extra RAM (16GB) was wasted.

The poor performance on the t2.micro instances was not solely due to the reduced vCPU; it was specifically due to the 'CPU Credit' system.

t2.micro instances were designed around the concept of 'burstable' performance, where instances could generate only 6 'CPU credits' per hour. The first burst of processing 120 files rapidly depleted the launch credits. When the CPU Credit Balance reached zero, and the instances were throttled to baseline performance (10% of a core), the cases experienced a significant processing bottleneck. This confirms that t2 instances are unsuitable for sustained, high-throughput batch processing.

Cost Efficiency: While c5.large instances are around seven times the hourly cost of t2.micro ($0.085 vs. $0.0116), the time it takes to process each task decreased by nearly 70% (approximately six minutes fifteen seconds to one minute fifty seconds).

- **Operational Advantage:** c5.large provides better performance for intermittent workloads that could cause backlogs to accumulate.
- **Economic Verdict:** A slight increase in compute expense will likely reduce the probability of a scaling delay and help ensure the "low latency" requirement is met.

Task D: Architecture Optimization

This section proposes improvements based on the AWS Well-Architected Framework.

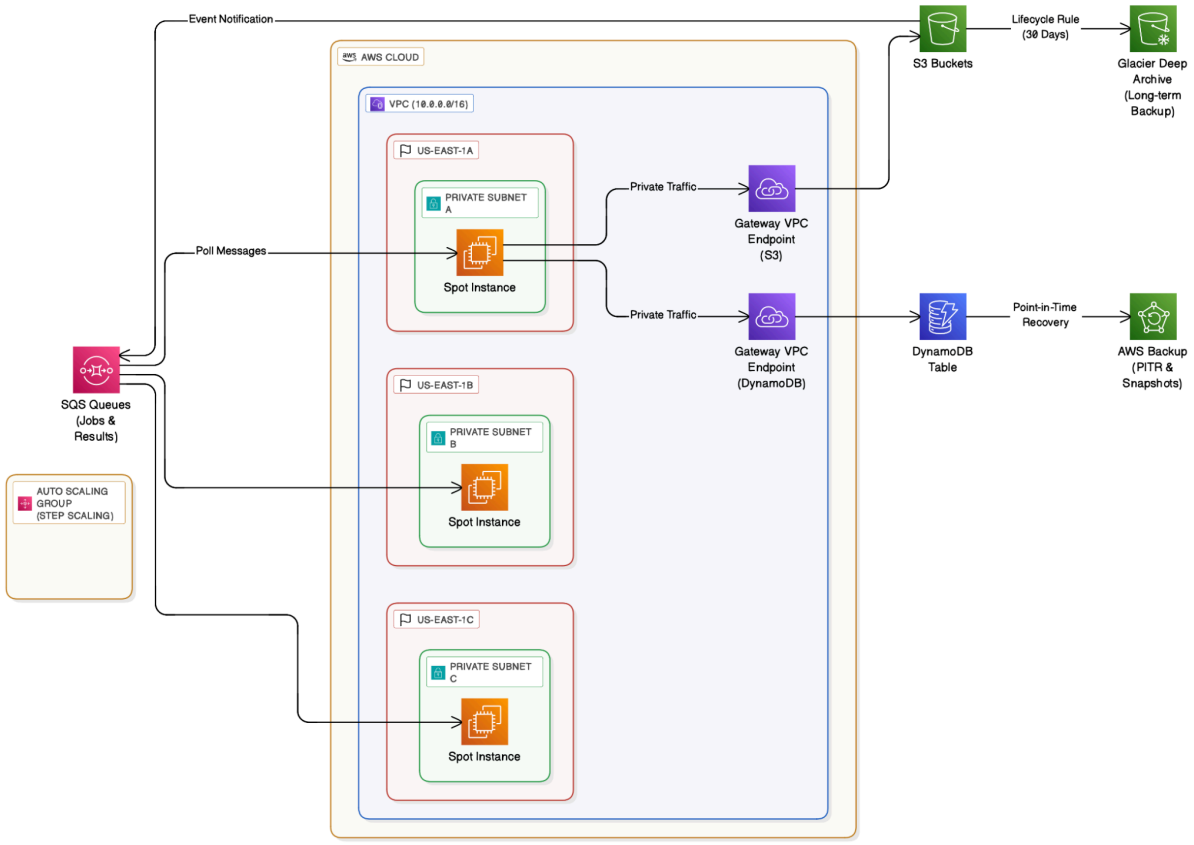


Fig 4.1: Optimized Architecture.

4.1 Reliability Pillar: Multi-AZ Deployment

In the lab environment, most instances have been launched in us-east-1c. However, to provide greater redundancy and reliability, the following recommendation is made:

- **Recommendation:** Set the Auto Scaling Group to deploy instances equally across three Availability Zones (us-east-1a, us-east-1b, and us-east-1c).
- **Benefit:** If one Availability Zone fails, the Auto Scaling Group will replace instances in the other Availability Zones to maintain continuous operation and prevent downtime.

4.2 Data Durability and Long-Term Storage

To address the need for extended data retention, the following general backup strategy has been proposed.

- **S3 Lifecycle Policies (Glacier):** Files uploaded to the 'Uploading' S3 bucket need to have an S3 Lifecycle Policy established that will transition the objects to Amazon S3 Glacier Deep Archive after thirty days. At approximately $0.00099/GB, Glacier Deep Archive is the least expensive option and is used to store data that may require occasional retrieval for compliance purposes.
- **Database Backups:** To protect the outcome of processed data, Point-in-Time Recovery (PITR) should be enabled on the DynamoDB table that is being created. PITR provides the ability to restore the table in a matter of seconds using data from the last 35 days and mitigates threats associated with accidental writes and deletes to the table. Generally, AWS Backup may also be used as a source repository.

4.3 Cost Optimization

The load test utilized On-Demand instances to simulate the load for the requirements specified in Task D. To meet the "occasional use" requirements of Task D, two modifications to the load test are recommended:

1. **Scale to Zero:** First, modify the Auto Scaling Group to set the Minimum Capacity to 0. Once the SQS queue is empty and there are no longer any instances running, the cost associated with running those instances is eliminated. The Scale-Out policy is responsible for initiating the launch of additional instances based on receiving files. Although the coursework specifies that the minimum capacity must be set to 1 to ensure immediate availability, the "occasional use" workload allows for a reduction in the Minimum Capacity to 0. This is the correct design decision and removes the expense associated with idling instances.
2. **Spot Instances:** The Launch Template needs to be modified to request Spot Instances.Since the worker is a stateless application (i.e., fetching jobs from SQS), when an AWS Spot Instance is reclaimed, the SQS "Visibility Timeout" returns the message to the SQS queue and a subsequent node processes the message. Spot Instances provide a maximum discount of 90 percent compared to On-Demand Instances while the system is running.

4.4 Security: VPC Endpoints and Private Access

Currently, the instances accessing S3 and DynamoDB do so through the public Internet or through a NAT Gateway.

- **Recommendation:** Establish Gateway VPC Endpoints for S3 and DynamoDB.
- **Benefit:** Using Gateway VPC Endpoints for S3 and DynamoDB enables the communication between the worker process and the data stores to occur on the AWS private network backbone. This eliminates the possibility of latency associated with communication occurring over the Internet, and eliminates data egress fees associated with the NAT Gateway, as all communication remains internal to AWS.

Task E: Further Improvements (Big Data)

The following services are proposed to enhance the performance and scalability of the applications in light of the limitations observed in the load test.

5.1 Serverless Compute with AWS Lambda

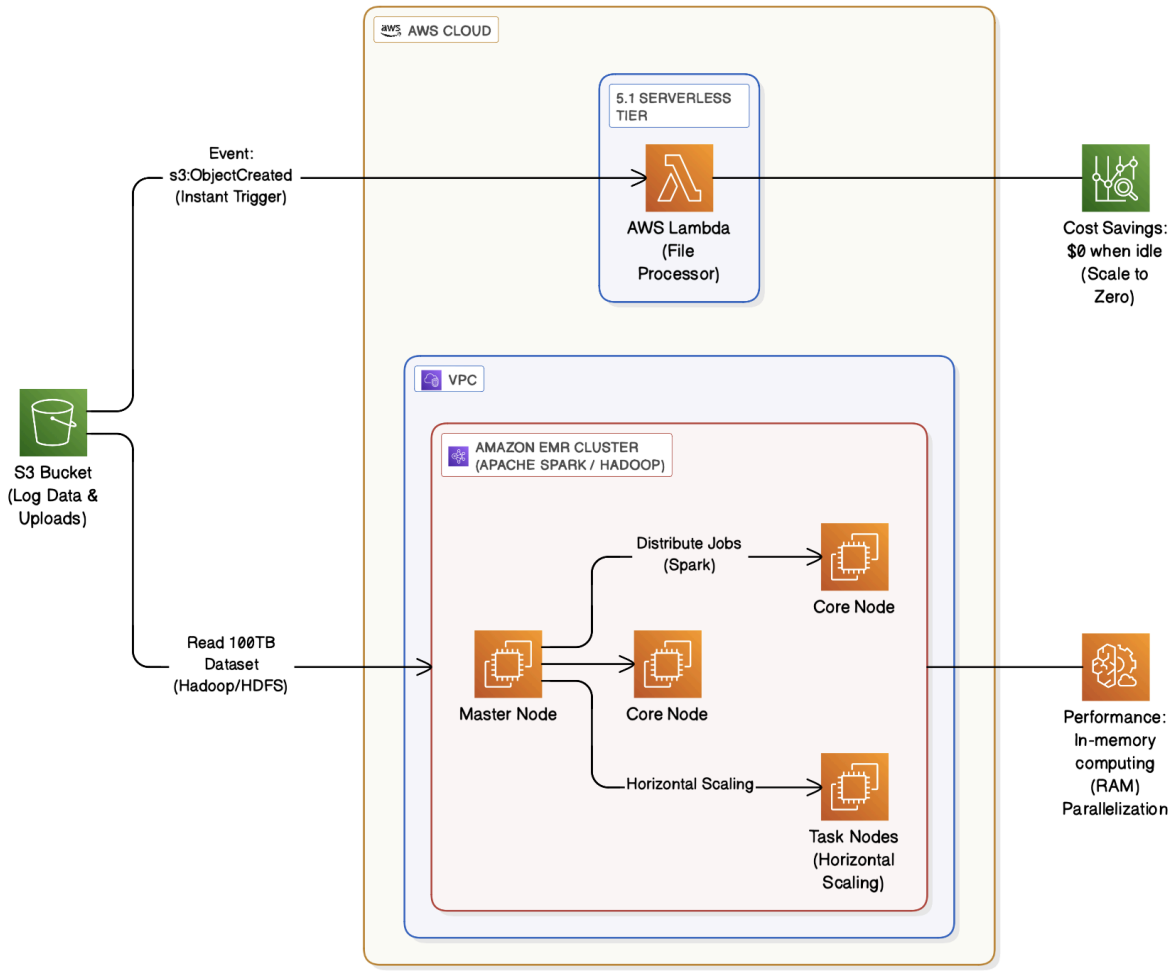
File Upload Workload (e.g., uploading 120 files in Task C) causes problems in the current EC2 architecture on the AWS platform due to the time required to initialize instances, and the idle resource consumption.

- **Proposed Change:** Replace the EC2 Auto Scaling Group with AWS Lambda functions that are directly triggered by s3:ObjectCreated event notifications.
- **Advantage:** AWS Lambda offers instantaneous elasticity as well as the capability to instantly deploy 1000+ concurrent run-time environments; this feature alone will significantly eliminate the 90-120 seconds that Task C experienced while waiting for the Amazon EC2 instance to be created. In addition, the pricing model for AWS Lambda is based on execution time in milliseconds; therefore, if an instance of AWS Lambda is idle, the cost is $0 per millisecond. This pricing model fits perfectly into the "occasional" use case scenario because it allows for no-cost usage during periods of idleness, thereby providing greater cost savings than using EBS snapshot instances or idle Amazon EC2 instances.

5.2 Big Data Analytics with Amazon EMR

Currently, the Go application either executes files one at a time or uses local constraints to concurrently execute up to 16 files. Since the size of the data set will grow from a few small text files to 100 TB of log files (Big Data), the RAM of a single node will eventually become a bottleneck, and the current architecture will fail.

- **Proposed Change:** Implementation of Amazon EMR (Elastic MapReduce) utilizing Apache Spark.
- **Advantage:** Amazon EMR provides a managed environment for Hadoop, which enables in-memory computing across a distributed environment. In Task C, EMR demonstrated performance improvements through vertical scaling with the c5.large instance type, but horizontal scaling, also provided by EMR, is the only viable method to process large amounts of data. Additionally, the Resilient Distributed Datasets (RDDs) available in Apache Spark provide for parallelization and text frequency analysis across many hundreds of nodes, yielding better performance and alleviating the constraints of processing large amounts of data within a single EC2 environment.


AWS CLOUD
5.1 SERVERLESS TIER
Event:
s3:ObjectCreated
(Instant Trigger)
AWS Lambda
(File
Processor)
Cost Savings:
$0 when idle
(Scale to
Zero)
VPC
AMAZON EMR CLUSTER
(APACHE SPARK / HADOOP)
S3 Bucket
(Log Data &
Uploads)
Distribute Jobs
(Spark)
Core Node
Read 100TB
Dataset
(Hadoop/HDFS)
Master Node
Core Node
Horizontal Scaling
Task Nodes
(Horizontal
Scaling)
Performance:
In-memory
computing
(RAM)
Parallelization

Thank you