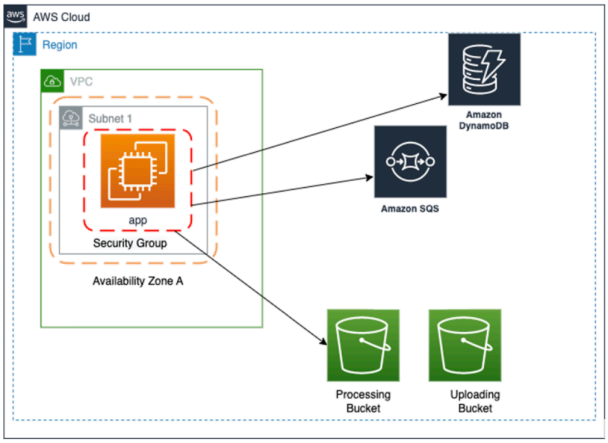
Task 2
Scaling the WordFreq Application

The WordFreq application is a functional tool developed in the Go programming language that analyses text files to identify and count the top ten most frequent words. It processes multiple text files sequentially and integrates several AWS services for its operation.



The system employs two S3 buckets: one for uploading text files from a local machine and another for processing those files. Notifications of file uploads trigger messages in an SQS queue, initiating processing tasks (jobs). A second SQS queue holds the results of the analysis, which are also stored in a DynamoDB NoSQL database for record-keeping. The application operates on an Ubuntu Linux EC2 instance, which is responsible for executing the tasks and coordinating interactions with AWS services.

The following tasks involve setting up and configuring the WordFreq application (Task A), implementing and optimising auto-scaling for dynamic workload management (Task B), performing load testing to evaluate and fine-tune scaling performance (Task C), proposing architectural enhancements for better resilience, cost-efficiency, and security (Task D), and identifying advanced data processing alternatives for improved performance and robustness (Task E).

Task A: Setting Up the EC2 Instance, Database, Storage Buckets, Queues, and Worker Setup

We begin by launching a new EC2 instance, as mentioned with the following metrics

Configuration	Details
Instance Name	wordfreq-dev
Instance Type	t2.micro
Operating System	Ubuntu Server 22.04 LTS
Application Hosted	WordFreq
Security Group	Inbound traffic on port 22 (SSH)
IAM Role	EMR_EC2_DefaultRole
Purpose	Environment for WordFreq application

The purpose of launching the EC2 instance (wordfreq-dev) is to create an environment where the WordFreq application can be hosted and tested.

This instance will be shut down later when we will be processing the application, it is initially needed to set up the necessary infrastructure, configure the environment, and ensure specific communication between the application and other AWS services. The security group allows secure SSH access for management, and the IAM role ensures the instance can interact with AWS services like S3 or DynamoDB as required by the application. Once the environment is configured and validated, the instance has to be stopped as part of the scaling and cost optimisation strategy.

Moving onto creating S3 buckets and SQS jobs with the details mentioned below

S3 Bucket	Purpose	Bucket Name
Uploading Bucket	Configured to accept uploads from the AWS environment.	ts-wordfreq-uploading
Processing Bucket	Used for files in the processing stage.	ts-wordfreq-processing

SQS Queue	Purpose	Queue Name
Jobs Queue	Handles job notifications, triggered when files are uploaded to the processing	wordfreq-jobs
Results Queue	Handles the results of processed files, notifying when processing is complete.	wordfreq-results

The process of copying text files from the Uploading Bucket (ts-wordfreq-uploading) to the Processing Bucket (ts-wordfreq-processing) triggers the WordFreq application. The file is first copied from the uploading bucket to the processing bucket using command (aws s3 cp s3://ts-wordfreq-uploading/ s3://ts-wordfreq-processing/ --recursive --exclude "*" --include "*.txt"). Processing bucket here is acting as an intermediary. Copying txt. files into the processing bucket is important, Whenever a new file is copied an event notification is sent to the Jobs Queue (wordfreq-jobs) via an S3 trigger which in the end instructs the worker process to begin and start processing files. Once the processing is complete, the Results Queue (wordfreq-results) handles the results, and notifies that the job is finished. The worker, which is continuously monitoring the queue for incoming messages, will pick up this notification. Then the worker will store these results in DynamoDB (wordfreq).

In conclusion, Task A includes the setup of the EC2 instance, S3 buckets, SQS queues, and DynamoDB, along with configuring and testing the application's functionality.

Task B: Design and implement an auto-scaling mechanism for the application to dynamically handle varying workloads, selecting appropriate CloudWatch metrics.

The goal is to add auto-scaling functionality to the WordFreq application, which allows it to process multiple uploaded files simultaneously by dynamically scaling the number of instances.

Scaling ensures that the application can process a larger volume of tasks in parallel when needed, while also conserving resources during low-demand periods. The process involves setting up scaling policies and creating alarms based on relevant performance metrics, i.e. **SQS ApproximateNumberOfMessagesVisible** metric.

Here is a breakdown of the steps implemented:

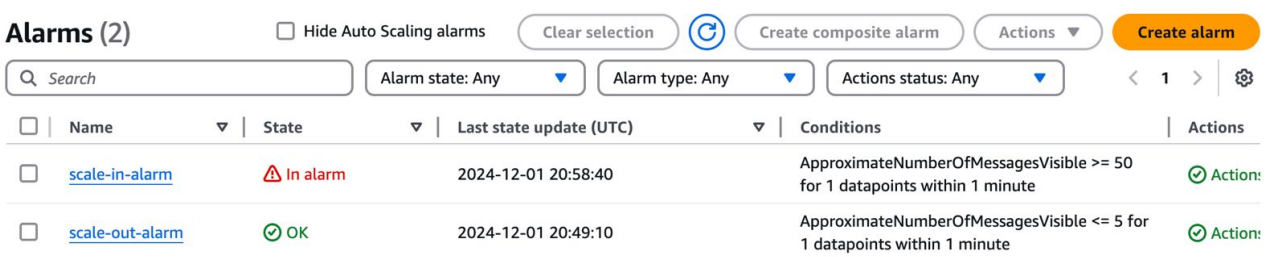
1. Setting Up the Auto-Scaling Group (ASG): The auto-scaling group (ASG) is responsible for managing the number of EC2 instances running the WordFreq worker process.

To begin, we first need to define the **desired capacity**, **minimum capacity**, and **maximum capacity** for the ASG.

Capacity Type	Value	Responsibility
Desired Capacity	1	Ensures at least one worker instance is available to process messages at all times.
Minimum Capacity	1	Guarantees at least one instance is running, even during periods of low activity.
Maximum Capacity	8	Defines the upper limit for scaling up under high load, while preventing over-allocation.

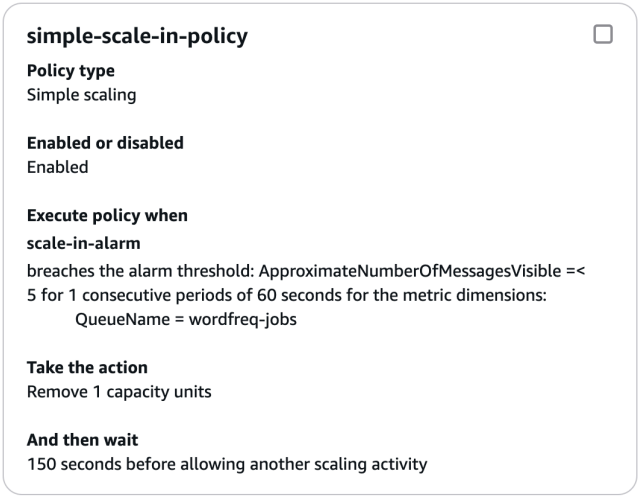
2. Creating Alarms: To trigger the scaling policies, we need to create **CloudWatch** alarms that monitor the performance metrics. These alarms are linked to the **SQS ApproximateNumberOfMessagesVisible** metric to track the queue depth.

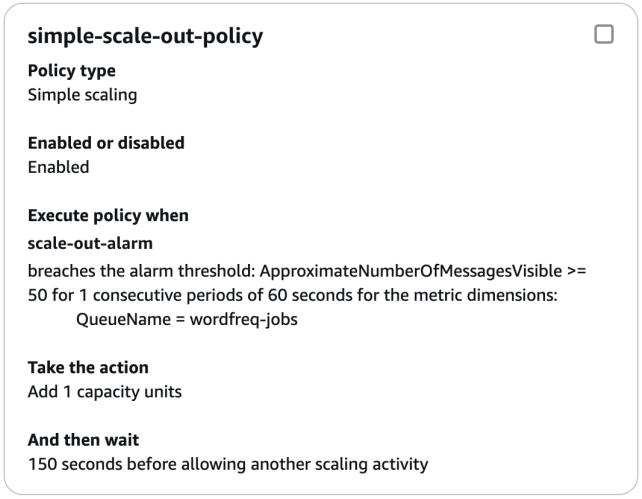
Alarm Type	Threshold	Purpose	Logic
Scale-Out Alarm	>= 50 messages in 1 minute	To handle high loads by adding worker instances when the queue depth increases.	Monitors the wordfreq-jobs queue; when messages exceed 50 in 1 minute, triggers the scale-out policy to add a new worker instance.
Scale-In Alarm	<= 5 messages in 1 minute	To optimize costs by reducing worker instances during low load periods.	Monitors the wordfreq-jobs queue; when messages fall below 5 in 1 minute, triggers the scale-in policy to remove the most recent worker instance.



3. Defining Scaling Policies: Scaling policies define the conditions under which the system will scale in or scale out.

Policy Name	Policy Type	Execute Policy When	Action	Wait Time	Use	Logic
Simple-Scale-In Policy	Simple scaling	ApproximateNumberOfMessagesVisible <= 5 for 1 consecutive period of 60 seconds in the wordfreq-jobs queue.	Remove 1 capacity unit	150 seconds	Optimizes costs by scaling down instances during low activity periods.	Monitors the queue; if visible messages are 5 or fewer, the policy removes 1 worker instance and waits 150 seconds before allowing another scaling action.
Simple-Scale-Out Policy	Simple scaling	ApproximateNumberOfMessagesVisible >= 50 for 1 consecutive period of 60 seconds in the wordfreq-jobs queue.	Add 1 capacity unit	150 seconds	Ensures high load handling by scaling up instances to process increased message volume.	Monitors the queue; if visible messages are 50 or more, the policy adds 1 worker instance and waits 150 seconds before allowing another scaling action.



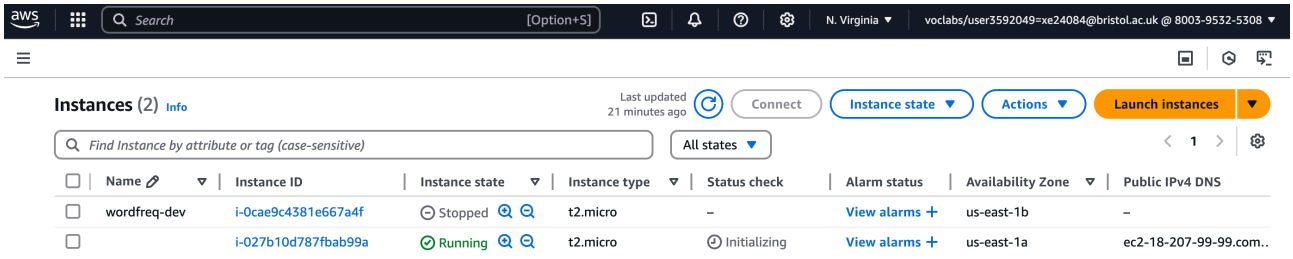


Task C: Implementing and Testing Auto-Scaling Policies for Enhanced Scalability

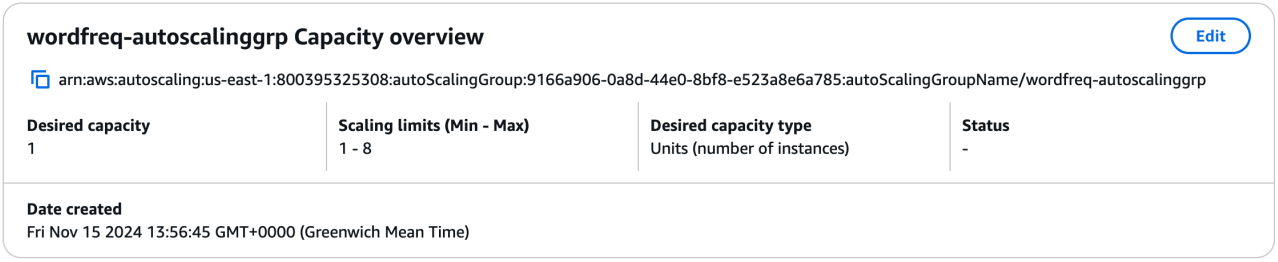
The testing process starts by uploading around 120 text files to the uploading S3 bucket. These files are then copied into the processing bucket to see how well the system handles scaling. The focus is on how the application adds instances during high workloads and removes them when the load decreases. To fine-tune the setup, different EC2 instance types are tested to find the best processing time. Adjustments are also made to the desired, minimum, and maximum capacity settings, ensuring the system scales smoothly and efficiently, no matter the workload. The scaling policies and alarms are further refined to ensure quick responses during busy periods while avoiding unnecessary resource usage during quieter times.

Initial Optimisation Attempt:

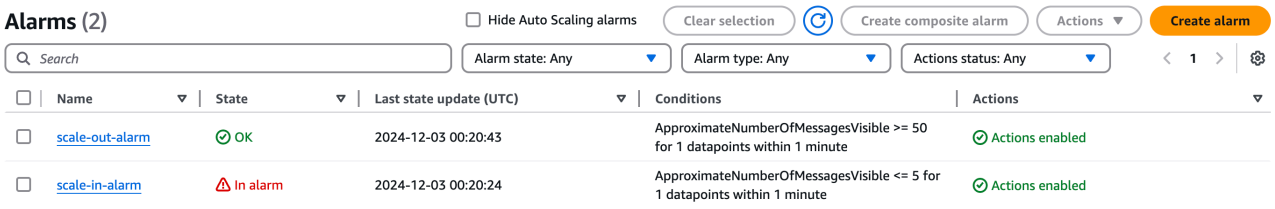
1. EC2: 1 Running t2.micro ASG Instance.



2. ASG: Set the desired, minimum and maximum capacity.



3. CloudWatch Alarms:



4. Setting up Scaling Policies:

Auto Scaling group: wordfreq-autoscalinggrp

simple-scale-in-policy

Policy type
Simple scaling

Enabled or disabled
Enabled

Execute policy when
scale-in-alarm
breaches the alarm threshold: ApproximateNumberOfMessagesVisible =< 5 for 1 consecutive periods of 60 seconds for the metric dimensions:
QueueName = wordfreq-jobs

Take the action
Remove 1 capacity units

And then wait
150 seconds before allowing another scaling activity

simple-scale-out-policy

Policy type
Simple scaling

Enabled or disabled
Enabled

Execute policy when
scale-out-alarm
breaches the alarm threshold: ApproximateNumberOfMessagesVisible >= 50 for 1 consecutive periods of 60 seconds for the metric dimensions:
QueueName = wordfreq-jobs

Take the action
Add 1 capacity units

And then wait
150 seconds before allowing another scaling activity

After updating the setup with the desired metrics and values, the next step is to upload the 120 text files for testing.

5. Upload and copy txt. files from uploading bucket to processing to trigger the job queue.

ts-wordfreq-uploading Info

Objects | Properties | Permissions | Metrics | Management | Access Points

Objects (120) Info

Copy S3 URI | Copy URL | Download | Open | Delete | Actions | Create folder | Upload

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Find objects by prefix

1

Name	Type	Last modified	Size	Storage class
1.txt	txt	December 3, 2024, 01:20:18 (UTC+00:00)	938.3 KB	Standard
10.txt	txt	December 3, 2024, 01:20:23 (UTC+00:00)	894.4 KB	Standard
100.txt	txt	December 3, 2024, 01:19:17 (UTC+00:00)	1.3 MB	Standard

ts-wordfreq-processing Info

Objects | Properties | Permissions | Metrics | Management | Access Points

Objects (120) Info

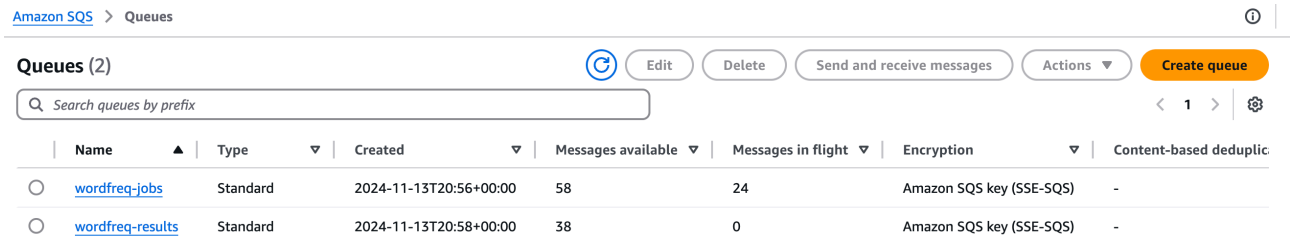
Copy S3 URI | Copy URL | Download | Open | Delete | Actions | Create folder | Upload

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

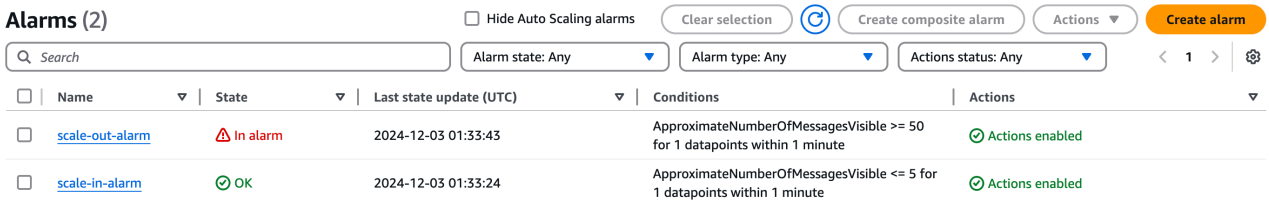
Find objects by prefix

1

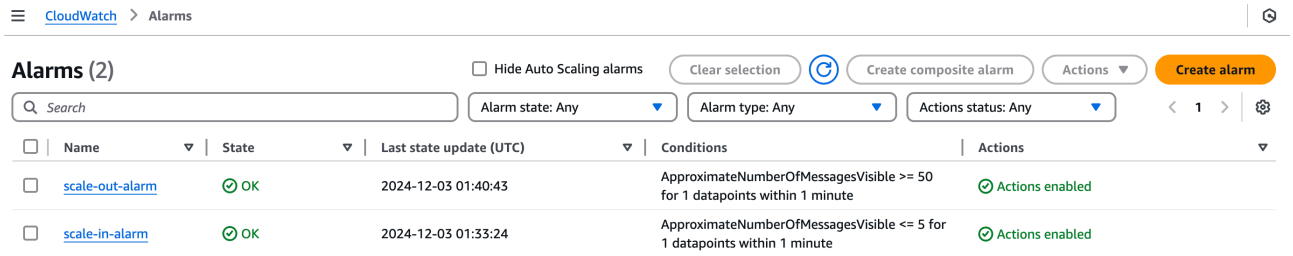
Name	Type	Last modified	Size	Storage class
1.txt	txt	December 3, 2024, 01:30:45 (UTC+00:00)	938.3 KB	Standard
10.txt	txt	December 3, 2024, 01:30:45 (UTC+00:00)	894.4 KB	Standard
100.txt	txt	December 3, 2024, 01:30:45 (UTC+00:00)	1.3 MB	Standard
101.txt	txt	December 3, 2024, 01:30:45 (UTC+00:00)	452.3 KB	Standard



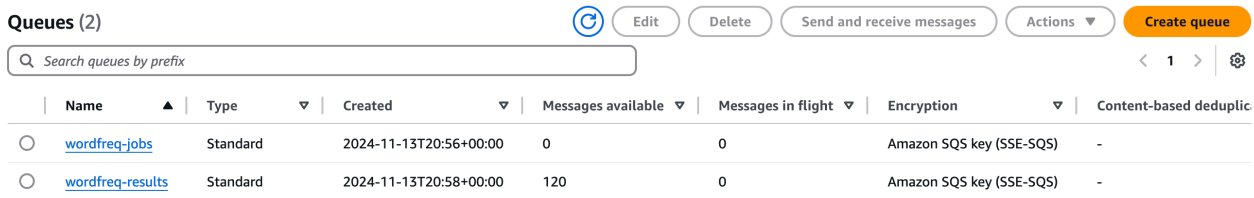
Moving onto **CloudWatch Alarms**



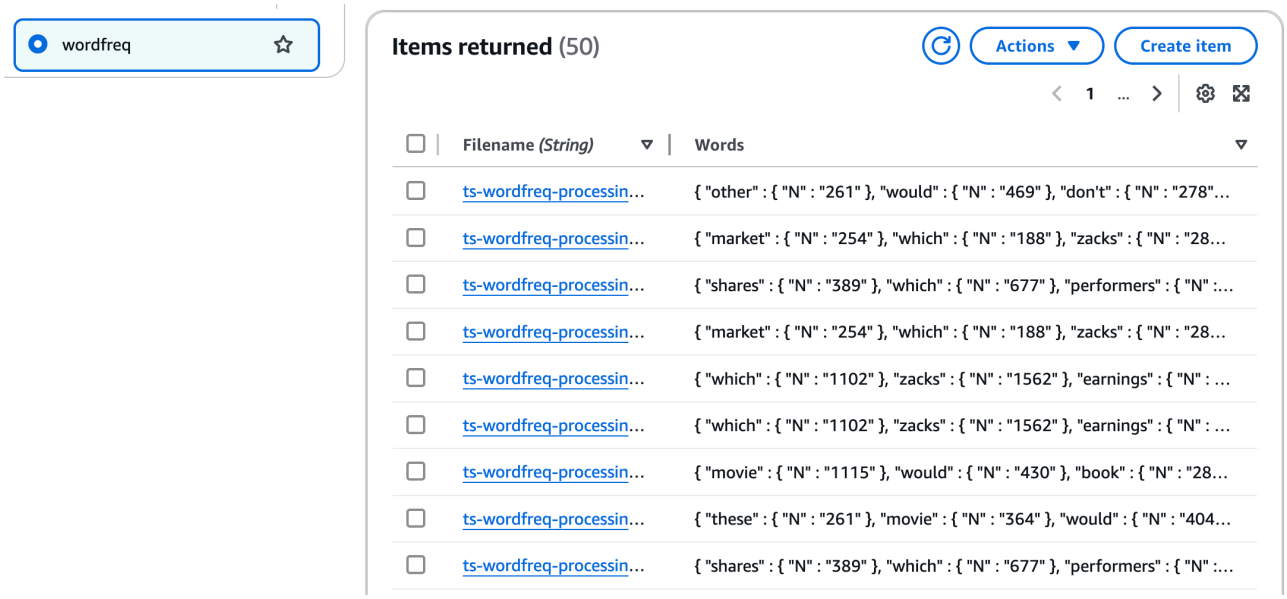
The **"scale-out-alarm"** is in an **"In alarm"** state because the condition specified for the alarm was triggered. The alarm is set to activate when the **"ApproximateNumberOfMessagesVisible"** metric in the SQS queue exceeds **50 messages** for 1 minute. Since this threshold was crossed, indicating a backlog of messages, the alarm activated to scale out and add more instance to handle the load.



The **"scale-out-alarm"** and **"scale-in-alarm"** are in the **"OK"** state because their conditions to scale out and in were not met during the observation period.



Once all the txt. files (120 txt. Files) are processed, the application's process is terminated and then we proceed to check the **DynamoDB** for the entries.

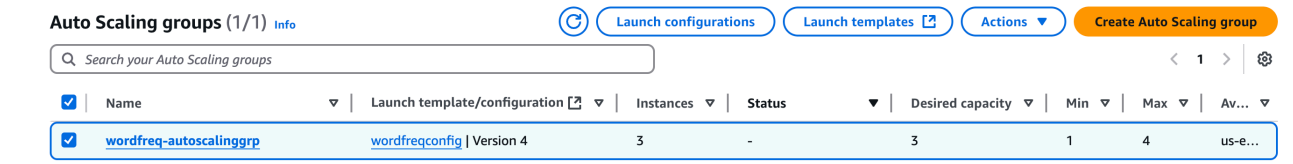


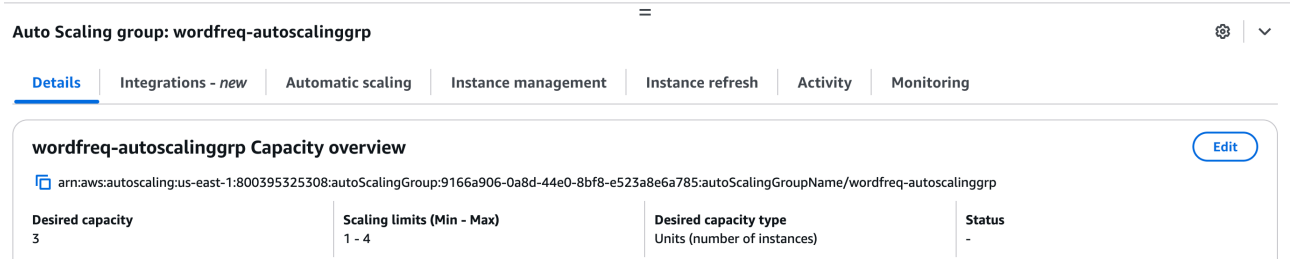
Field	Value/ Description
Instance Type	t2.micro
Scaling Policy	Simple scaling policy
Scaling Metrics	SQS ApproximateNumberOfMessagesVisible
Desired Capacity	1
Min. Capacity	1
Max. Capacity	8
Wait Time	2m 30s i.e. 150 seconds
Capacity Units	Scale Out: +1, Scale In: -1
Processing Time	12m 43s
Processing Time for Individual Files	6s

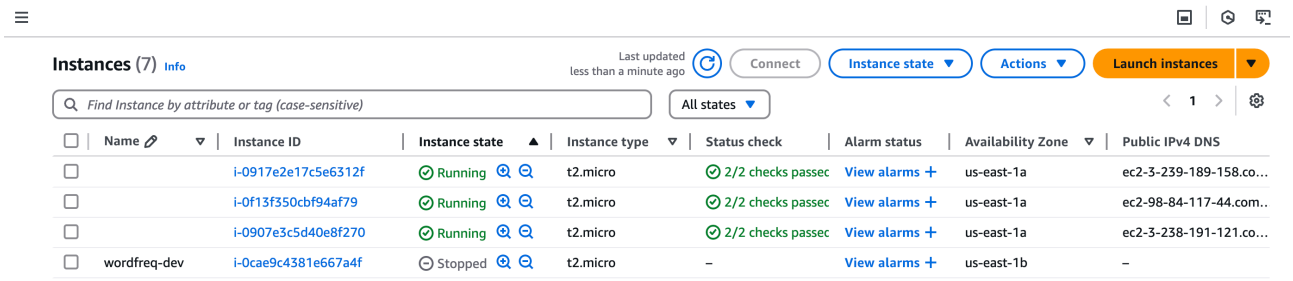
After testing with the initial values and achieving a processing time of 12 minutes and 43 seconds, I proceeded to optimise the processing time by adjusting the ASG instance metrics, scaling policies, and instance type.

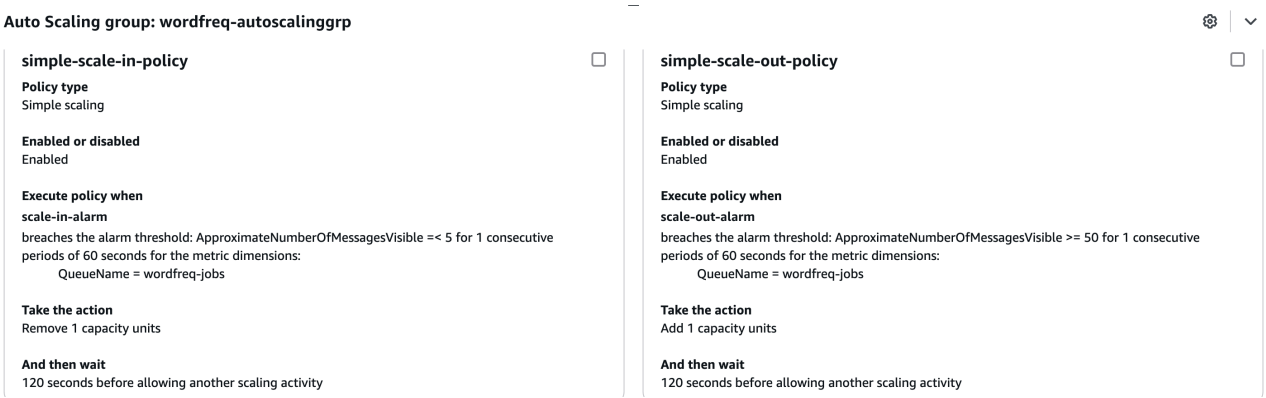
Though before we start another process, it's important to purge the queues and empty the processing bucket.

Field	Value/ Description
Instance Type	t2.micro
Scaling Policy	Simple scaling policy
Scaling Metrics	SQS ApproximateNumberOfMessagesVisible
Desired Capacity	4
Min. Capacity	1
Max. Capacity	4
Wait Time	120
Capacity Units	Scale Out: +1, Scale In: -1
Processing Time	7m 9s
Processing Time for Individual Files	3s 500ms

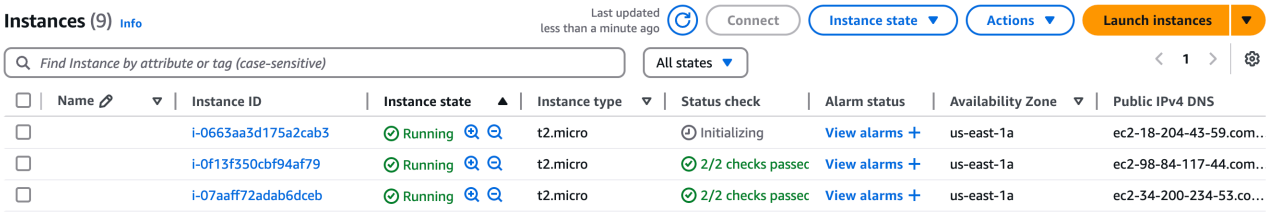


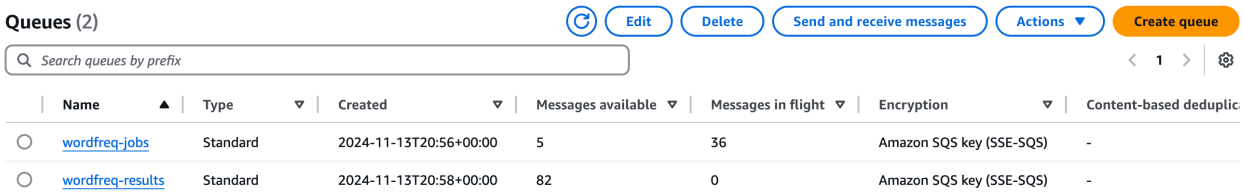






Fluctuations in the ASG Instances:



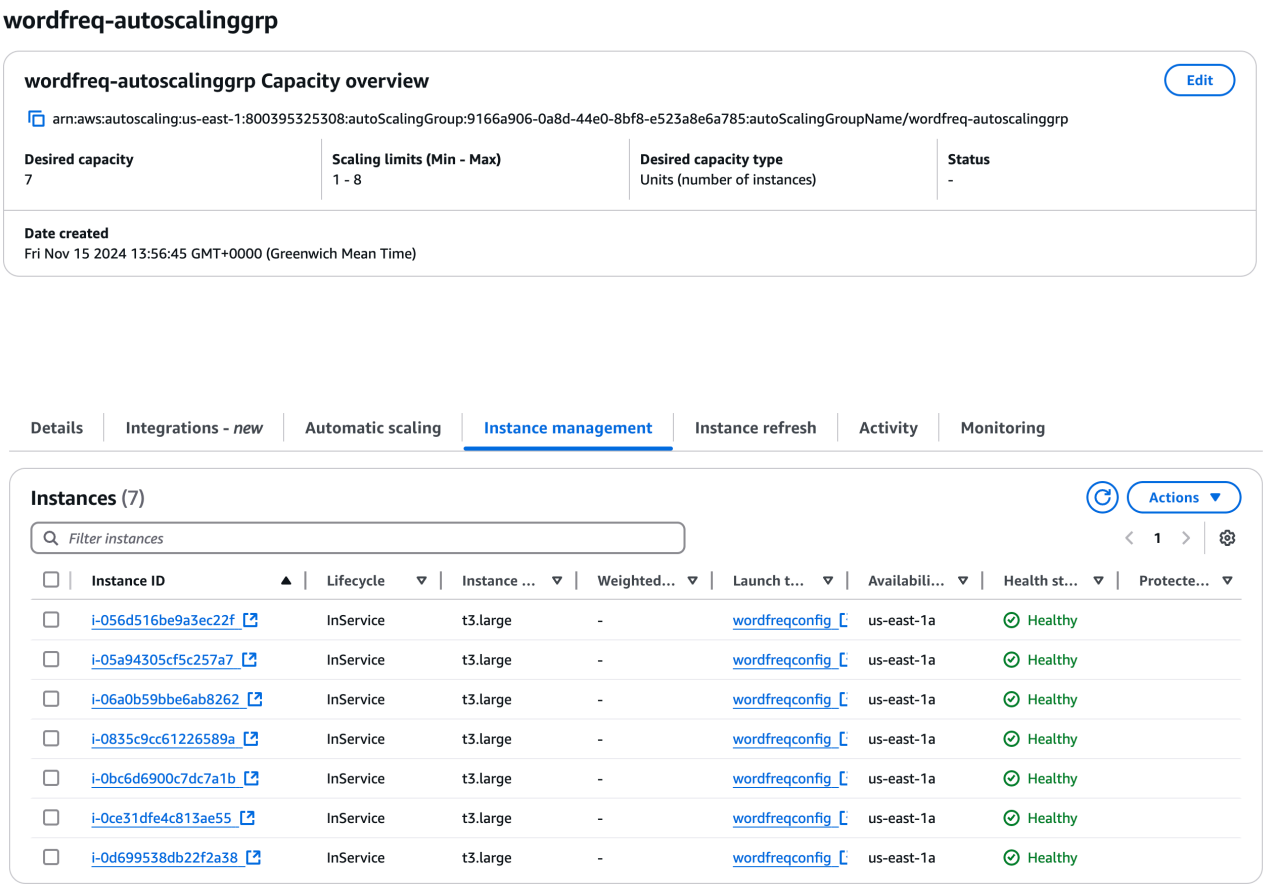


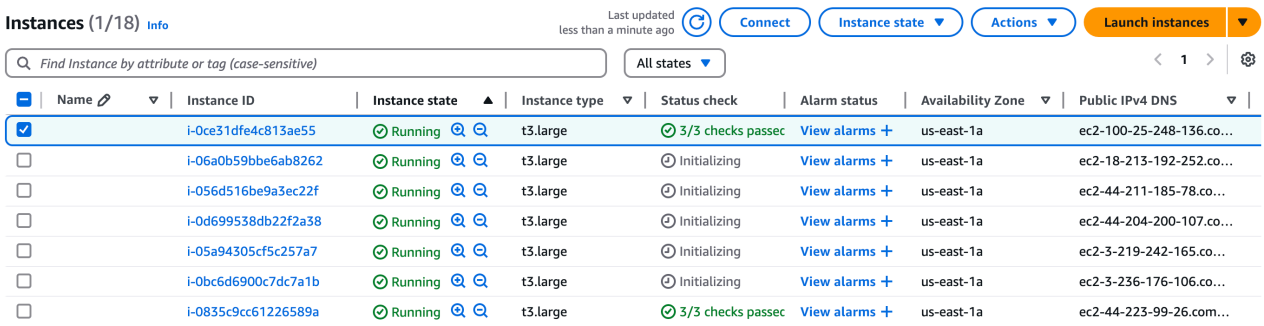
The fluctuations in the running EC2 instances are driven by the scaling policies linked with the Auto Scaling Group (ASG) and the state of the SQS wordfreq-jobs and wordfreq-results queues. The ASG uses CloudWatch alarms, **scale-in-alarm** and **scale-out-alarm**, to dynamically adjust the number of instances based on the queue message load. The scale-out-alarm triggers when the number of visible messages in the queue exceeds 50 for a 1-minute period, indicating increased demand and prompting the ASG to add more EC2 instances to handle the processing load.

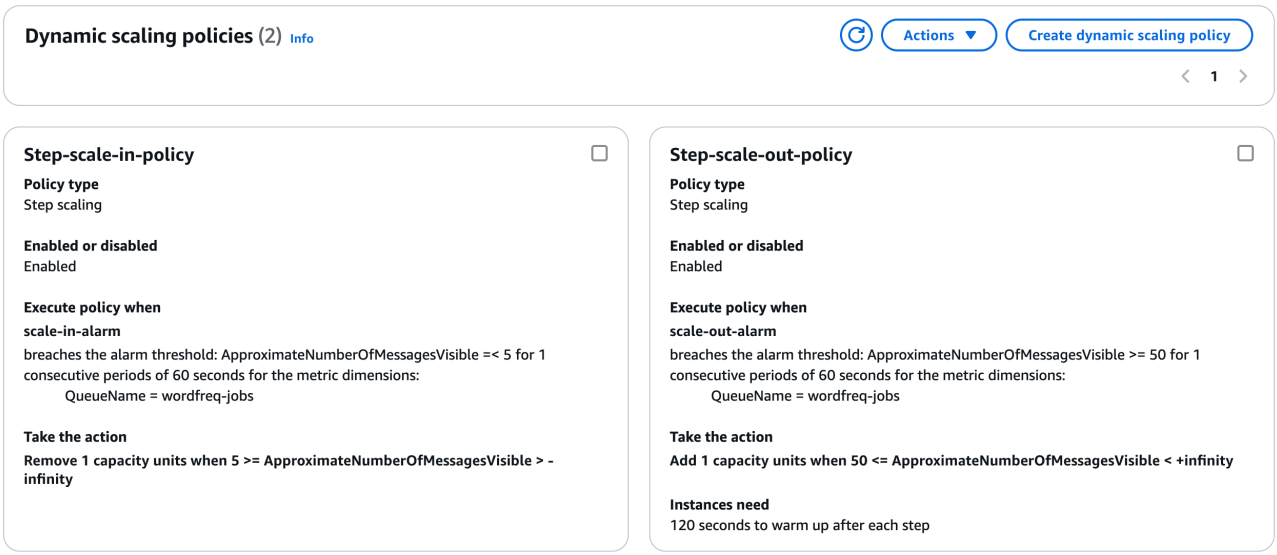
Similarly, the scale-in-alarm triggers when the number of visible messages drops below 5 for 1 datapoint in 1 minute, indicating reduced demand, which causes the ASG to scale down and terminate instances. These alarms ensure that the system adjusts to workload fluctuations in real-time, optimising resource usage by scaling up when there are more messages to process and scaling down when the queue load decreases, thus preventing over launching or under launching of instances.

Processing the Application with t3.large Instance and Step Scaling Policy

Field	Value/ Description
Instance Type	t3.large
Scaling Policy	Step scaling policy
Scaling Metrics	SQS ApproximateNumberOfMessagesVisible
Desired Capacity	7
Min. Capacity	1
Max. Capacity	8
Wait Time	2m i.e. 120 seconds
Capacity Units	Scale Out: +1, Scale In: -1
Processing Time	5m 16s
Processing Time for Individual Files	2s 630ms







So, now keeping the scale-in and scale-out conditions constant for all three processes.

1. Scale-In: ApproximateNumberOfMessagesVisible <= 5 for 1 datapoint within 1 minute
2. Scale-Out: ApproximateNumberOfMessagesVisible >= 50 for 1 datapoint within 1 minute

In conclusion:

Field	t2.micro (1st Process)	t2.micro (2nd Process)	t3.large (3rd Process)
Instance Type	t2.micro (1 vCPU, 1 GB RAM)	t2.micro (1 vCPU, 1 GB RAM)	t3.large (2 vCPUs, 8 GB RAM)
Scaling Policy	Simple Scaling Policy	Simple Scaling Policy	Step Scaling Policy
Scaling Metrics	SQS ApproximateNumberOfMessagesVisible	SQS ApproximateNumberOfMessagesVisible	SQS ApproximateNumberOfMessagesVisible
Desired Capacity	4	4	7
Min. Capacity	1	1	1
Max. Capacity	4	4	8
Wait Time	2m 30s	2m	2m
Capacity Units	Scale Out: +1, Scale In: -1	Scale Out: +1, Scale In: -1	Scale Out: +1, Scale In: -1
Processing Time	12m 43s	7m 9s	5m 16s
Processing Time for Individual Files	6s 350ms	3s 500ms	2s 630ms

The significant change in processing time from **12 minutes 43 seconds** to **5 minutes 16 seconds** between the first and third processes can be due to several factors, primarily involving the **scaling policies**, **instance types**, and **configuration settings** that were used in each process.

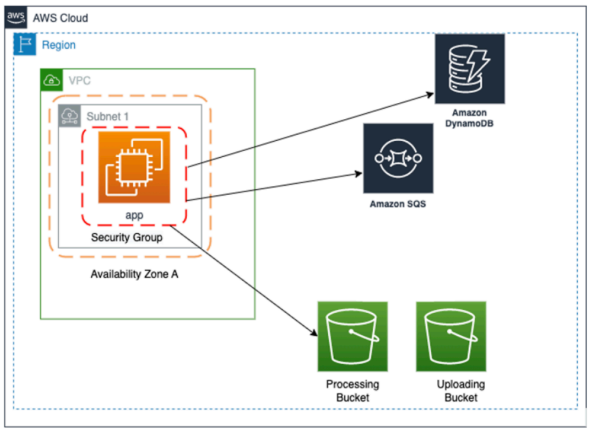
1. Instance Type: The most notable change is the instance type from t2.micro to t3.large. The first process uses a t2.micro instance, which has 1 vCPU and 1 GB RAM. This instance type is a general-purpose, low-performance machine with limited processing power. On the other hand, for the third process, I upgraded to a t3.large instance, which has 2 vCPUs and 8 GB RAM. This upgrade provides more computational resources which enables faster processing. The increased number of CPUs allows the application to handle parallel tasks, improving the overall performance and reducing the time it takes to process files.

2. Scaling Policy: Another important difference is in the scaling policies used. The first and second processes rely on a simple scaling policy, which works by scaling out or in (I.e. up or down) based on a straightforward threshold. On the other hand, the third process uses a step scaling policy, which is more refined and adaptable. This policy lets the system adjust its capacity gradually, step by step, instead of making big changes all at once. This makes it more responsive to shifts in demand or workload, in this case, the number of messages waiting in the SQS queue.

3. Wait Time (Scaling Metrics): In all three processes, the scaling metric is based on SQS ApproximateNumberOfMessagesVisible, which measures the number of messages waiting to be processed in the queue. However, the wait time differs. Basically, shorter wait time allows quicker scaling and better instance scaling, further improving processing efficiency.

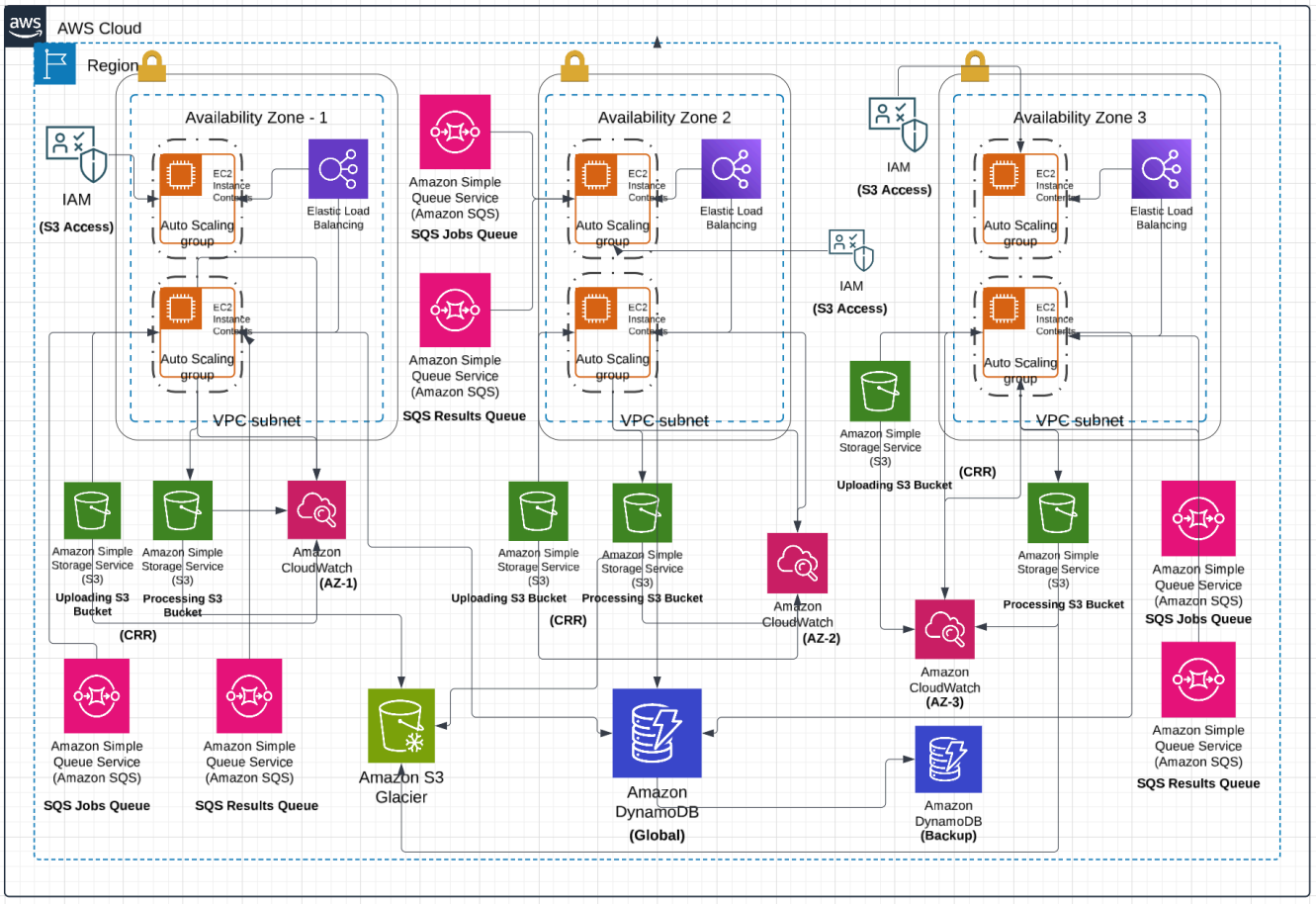
Task D: Re-designing WordFreq for Enhanced Resilience, Backup, Cost-Efficiency, and Security

The current architecture of the WordFreq application depends on a single EC2 instance to handle all tasks, which exposes the system to several vulnerabilities and inefficiencies.



This architecture is highly susceptible to downtime during component failure, lacks proper mechanisms for backing up critical data, is not optimised for cost efficiency, and features limited security measures.

To address these challenges, a redesigned architecture is proposed, utilising AWS services and features explored during the Cloud Foundations course.



The new solution is designed to improve the application's resilience and availability by utilising AWS's scalability and fault tolerance features. It also ensures reliable data backups through services like Amazon S3 Glacier and DynamoDB, while optimising costs with auto-scaling. Additionally, it strengthens security while keeping the application's functionality and code intact.

To increase resilience and availability, we will upgrade the application architecture from a single EC2 instance to a more distributed and fault-tolerant setup. By implementing an Auto Scaling Group (ASG) with instances spread across multiple Availability Zones (AZs), the system will ensure high availability and reduce the risk of downtime due to AZ-level failures. Elastic Load Balancing (ELB) will distribute incoming requests evenly across instances, further improving the reliability of the system under varying traffic loads. DynamoDB Global Tables will be implemented to deliver low-latency access to data and ensure high availability across regions, ensuring that even in the event of regional failures, the application continues to function without any hiccups.

For example, consider a scenario where a user in the US East region (AZ 1) uploads a .txt file for word frequency analysis. The file's metadata and processing status are replicated in real-time to the US West region (AZ 2) using DynamoDB Global Tables. If an unexpected failure occurs in the US East region, the application can seamlessly access the uploaded file and its associated data from the US West region. Even if the EC2 instances in US East go down, the application can continue processing jobs using the replicated data from US West, ensuring no interruption in service. This upgraded architecture makes the WordFreq application more resilient and ensures high availability, allowing users to submit files and receive processed results regardless of regional failures.

Now to ensure long-term data protection, the architecture includes automated backup mechanisms to safeguard crucial information. S3 Glacier will be used for the cost-effective storage of archived files, offering durability for long-term retention. Additionally, DynamoDB backups will be automated, enabling point-in-time recovery of all tables, which is crucial for disaster recovery. With cross-region replication in S3, critical data will be duplicated across geographically separate locations, enhancing resilience.

For instance, consider a scenario where the WordFreq application processes thousands of user-uploaded .txt files for word frequency analysis. In the event of a system failure or data corruption in one region, S3 Glacier’s long-term storage will ensure that archived files are preserved and can be retrieved without loss. Similarly, if a critical table in DynamoDB is accidentally deleted, automated backups will allow the system to restore the table to its previous state, ensuring continuity. Cross-region replication in S3 ensures that all processed files are automatically duplicated in a separate AWS region, so if the primary region goes down, users can still access their data from the replicated location, minimising the risk of disruption. This setup guarantees that even in the face of disasters, the WordFreq application can recover quickly and maintain data integrity.

Cost efficiency is achieved by carefully selecting instance types and configuring the architecture to scale dynamically. For background processing tasks with minimal resource requirements, T3 instances, known for their cost-effectiveness, will be used. During peak traffic periods, the Auto Scaling Group (ASG) will dynamically adjust the number of instances, ensuring resources are added or removed based on demand. By using a weighted ASG configuration, instances with different resource capacities can be prioritised based on workload. For example, T3 instances might handle standard tasks, while larger instances, such as M5, are allocated higher weights to process

demanding workloads during spikes. This setup ensures that the system optimises performance while keeping costs under control, as resources are provisioned only when needed.

To draw a clearer understanding, during high-traffic periods when multiple users upload large .txt files simultaneously for processing, the ASG will automatically scale out by adding more T3 instances to handle routine tasks, like basic word frequency analysis. However, if certain files are particularly large or complex and require more compute power, the ASG will launch instances with higher weights, such as M5, to handle these demanding jobs.

The application will be hosted within a private Virtual Private Cloud (VPC), ensuring secure communication between resources and isolation from external networks. IAM roles are set up to follow the principle of least privilege, granting each service only the permissions it needs. In the proposed architecture, when a user uploads a .txt file to the WordFreq application for processing. The file is stored in an S3 bucket. With the VPC setup, the communication between the EC2 instances and S3 remains secure, ensuring that the file is not exposed to external threats. IAM roles are configured so that only the EC2 instances responsible for processing the file have the necessary permissions to access the S3 bucket. Additionally, bucket policies restrict access to the S3 bucket to only authorised users or services, preventing any unauthorised entity from retrieving or modifying the file. This setup ensures that sensitive data, like the user-uploaded file, is protected at every stage.

In conclusion, the following table summarises the proposed changes to the WordFreq application architecture, along with their impacts and the AWS services used.

Field Name	Current Feature	Proposed Feature	AWS Features/ Services Used	Impact (WordFreq App)
Hosting Environment	Single EC2 instance	Hosted within a private VPC	VPC (Virtual Private Cloud)	Ensures secure, isolated communication between resources, protecting user-uploaded files and sensitive data from unauthorized access in the WordFreq app.
IAM Roles	Limited or no role-based access control	IAM roles configured following the principle of least privilege	IAM (Identity and Access Management)	Limits service permissions to only what is necessary, reducing the risk of unauthorized access to WordFreq data, such as file uploads and results.
S3 Bucket Access	Potential unrestricted access	Access to S3 buckets restricted with bucket policies and permissions	S3 (Simple Storage Service), Bucket Policies	Ensures only authorized users and services can access WordFreq's processed data and uploaded files, improving security.
Security	No network isolation	Resources isolated within a private VPC, with secure communication between services	VPC, IAM Roles, Security Groups	Protects WordFreq's sensitive user data and processing tasks by isolating them from external threats and unauthorized access.
Data Protection	No clear backup or data replication strategy	Automated backups to S3 Glacier, cross-region replication for disaster recovery	S3 Glacier, DynamoDB, Cross-Region Replication	Enhances WordFreq's data protection by ensuring file backups and user data are preserved, even in case of regional AWS failures or application disruptions.
Auto Scaling	Static EC2 instances, manual scaling	Auto Scaling Group (ASG) to scale based on traffic load	EC2 Auto Scaling Group (ASG)	Automatically scales resources based on user upload volume and traffic, ensuring WordFreq handles peaks in file uploads efficiently without over-provisioning.

Task E: Alternative Data Processing Services for Enhanced Performance and Robustness

To further improve the performance of the WordFreq application and make it more robust, AWS Lambda and AWS Batch can be used as alternative data processing services.

AWS Lambda can significantly enhance the WordFreq application by transforming it into an event-driven system. For example, when multiple users upload large .txt files for word frequency analysis, each upload triggers a Lambda function to process the file immediately. This eliminates the need for continuously running EC2 instances, which were previously responsible for handling file processing tasks. Instead, AWS Lambda automatically scales based on the volume of file uploads. During peak periods, such as when many users upload files simultaneously, Lambda will spin up the necessary compute resources to handle the increased load, ensuring that no files are delayed or left unprocessed.

By eliminating the need for running EC2 instances, Auto-Scaling groups and complex scaling configurations Lambda optimises the infrastructure leading to reduced operational complexity. Once the file is processed, the Lambda function automatically terminates, which means the application only incurs costs during the function's execution. The application is highly cost-efficient even during times of low traffic, due to it’s pay per use model.

In addition, Lambda offers built-in retry mechanisms and error handling, improving the application's fault tolerance and reliability. This ensures that if an error occurs during the file processing, the function can retry automatically without any manual intervention. Therefore, AWS Lambda optimises the infrastructure, reduces operational overhead, and ensures that resources are used efficiently, making it an ideal solution for handling fluctuating demand and maintaining high availability.

Another alternative that can be implemented is AWS Batch, which is designed for large-scale batch processing and is particularly suited for more resource intensive tasks, such as handling large files.

In the context of the WordFreq application, consider a scenario where multiple users upload large .txt files simultaneously for word frequency analysis. With AWS Batch, the system can automatically manage these file processing tasks by placing them in job queues. For instance, files of varying sizes can be prioritised and scheduled based on the available resources and processing times. Large files that require more computational power could be allocated to higher priority queues, which will ensure that they are processed efficiently without overloading the system. AWS Batch would process the files by scaling dynamically based on the job’s requirements and execute these tasks in a controlled and optimised manner. This approach along with it’s built-in retry mechanisms for failed jobs helps the WordFreq application handle large-scale processing without compromising performance or reliability.

In conclusion, I have provided a comparison between AWS Lambda and AWS Batch, highlighting how they can significantly improve the WordFreq application’s performance and robustness.

Feature	AWS Lambda	AWS Batch
Use Case	Event-driven, small to medium tasks	Large-scale batch processing, compute-intensive tasks
Resource Provisioning	Serverless, automatic scaling based on events	Automatic provisioning based on job requirements
Scaling	Scales automatically with incoming traffic	Scales based on job resource requirements
Cost Model	Pay-per-use, costs incurred only when functions execute	Pay-per-use, cost based on resource usage
Compute Resources	Limited to small, short tasks, can scale quickly	Suited for resource-intensive, long-running tasks
Fault Tolerance	Built-in retry mechanisms and error handling	Built-in retry mechanisms for failed jobs
Example Use Case	File upload triggers processing (e.g., WordFreq file analysis)	Processing large .txt files or complex word frequency tasks
Integration Complexity	Easy to integrate with existing services for event-driven processes	Requires more configuration to define job queues and compute resources
Ideal for	Smaller tasks, quick processing like file uploads	Large, compute-heavy tasks like processing large text files
Replacement Impact	Replaces EC2 instances and ASG for event-driven processing, optimises infrastructure	Replaces EC2 instances and ASG for long-running, resource-intensive tasks, improves processing efficiency by automatic scaling