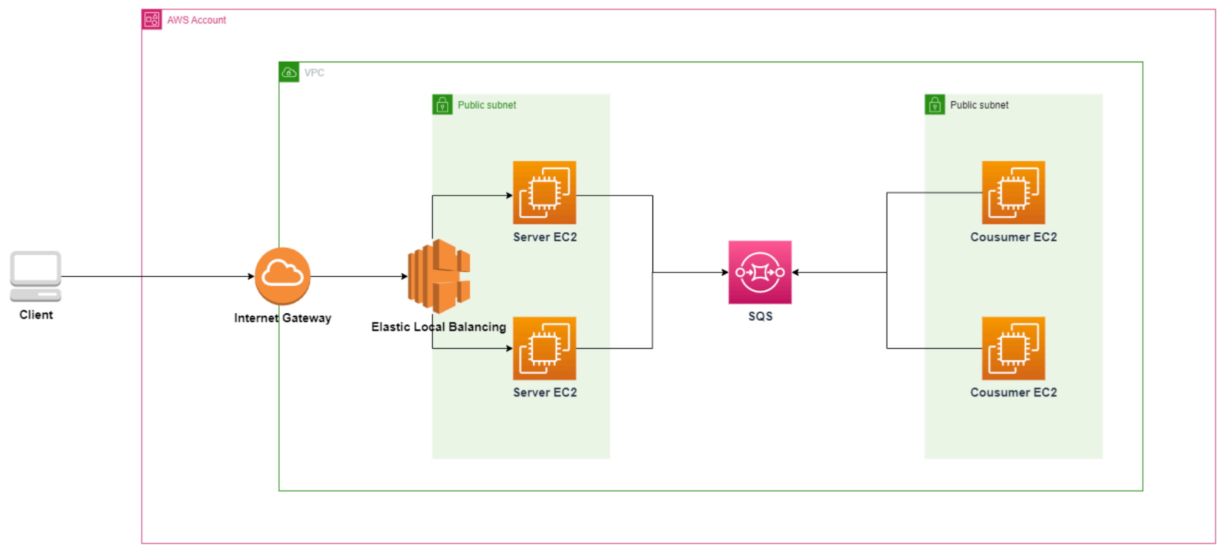
Git Repository URL

https://github.com/Orange-135/cs6650/tree/main/hw2

Server Design Description

High Level

Architecture Diagram



The following is an architecture diagram of the server configured with load balancing:

- The client first sends data to AWS Elastic Load Balancing (ELB).
- ELB then distributes the requests to the server.
- The server receives the requests and performs parameter validation.
- If there are no issues, it sends the data to an AWS SQS queue.
- The consumer retrieves and processes data from the SQS queue.

We use Spring Boot as the web framework.

Low Level

Detailed Request Sequence

Client

- The client sends requests using Spring's synchronous HTTP client.
- First Call: Processes 32,000 requests, with each thread sending 1,000 requests, totaling 32 threads.
- Second Call: Processes 165,000 requests, with each thread sending 5,160 requests, totaling 32 threads.
- This results in a total of 200,000 requests being sent.

After the requests are completed, the client saves each request's start timestamp, request method, request latency, and return status code to a CSV file. It also calculates and outputs:

Number of successful requests
Number of failed requests
Total running time (ms)
Average response time (ms)
Median response time (ms)
P99 response time (ms)
Minimum response time (ms)
Maximum response time (ms)
Throughput (requests per second)

Server

- The server uses @RestController to expose endpoints and specifies the interface path using @RequestMapping("/skiers").
- It utilizes the Jakarta Bean Validation API to validate parameters passed from the client, commonly used in Spring Boot 3 and newer versions.
- Upon receiving a message, it performs parameter validation to determine if the client's message is valid.
  - If valid, the message is saved to the SQS message queue.
  - If invalid, it returns the corresponding error.

Consumer

- The consumer continuously polls the queue.
- The thread pool is configured with 100 threads by default.
- As long as there is data in the queue, it consumes it immediately to ensure stable queue load and prevent excessive load.

Structure

Client Structure

```
com.example.client
├── ClientApplication.java
│    - Initialize and coordinate components
│    - Manage request statistics and logs
│    - Run tests and generate reports
```

```
│
├── config/
│   └── AppConfig.java
│       - Configure RestTemplate instance
│       - Handle HTTP request settings
│
├── generator/
│   └── RequestDataGenerator.java
│       - Generate random skier request data
│       - Fill request queue
│
├── pojo/
│   └── SkiersRequest.java
│       - Define request data structure
│       - Contains skier, resort, lift info
│
└── request/
    └── RequestSender.java
        - Send HTTP requests to server
        - Handle responses and retries
        - Record response times and status
```

Server Structure

```
com.example.server
├── ServerApplication.java  # Server main class
│   - Start Spring Boot application
│
├── controller/
│   └── SkiersController.java  # REST controller
│       - Handle /skiers endpoints
│       - Perform parameter validation
│       - Simulate processing delay
│
├── pojo/
│   └── SkiersRequest.java     # Data model class
│       - Define request data structure
│       - Contains validation annotations
│
└── exception/
    └── GlobalExceptionHandler.java  # Global exception handler
        - Handle parameter validation exceptions
        - Standardize error responses
```

Consumer Structure

```
com.example.consumer
├── ConsumerApplication.java  # Consumer main class
│    - Start Spring Boot application
│    - Enable scheduling support
│
├── config/
│    └── AwsSqsConfig.java    # AWS SQS configuration
│        - Configure SQS client
│        - Set AWS region
│
└── SqsMessageConsumer.java  # Message consumer service
     - Poll SQS queue periodically
     - Process messages concurrently
     - Delete processed messages
```

Data Flow

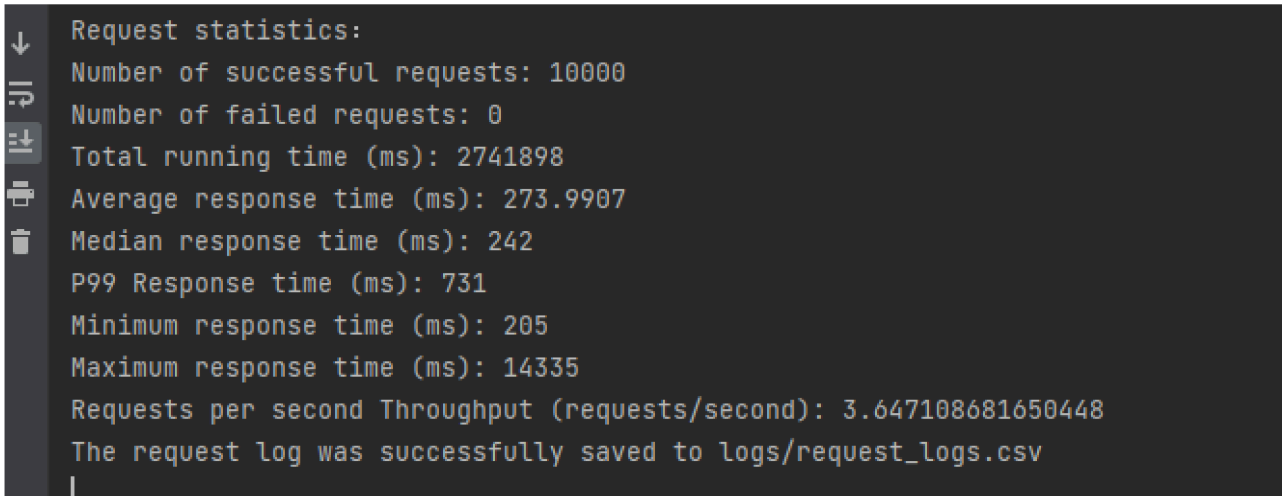
```
[Client] → HTTP Request → [Server] → SQS Message → [Consumer]
```

Results

Single-Threaded Test Results

The client uses a single thread to send 10,000 requests to the server.
CSV Link: https://github.com/Orange-135/cs6650/blob/main/hw2/logs/request_logs_10k.csv

Output:

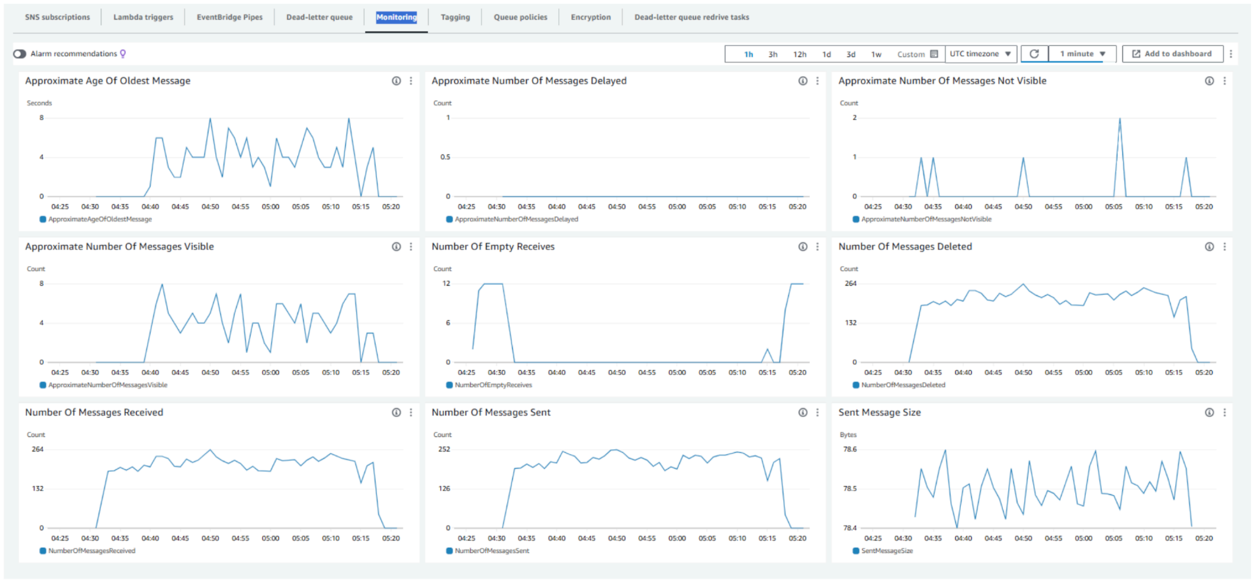


```
Request statistics:
Number of successful requests: 10000
Number of failed requests: 0
Total running time (ms): 2741898
Average response time (ms): 273.9907
Median response time (ms): 242
P99 Response time (ms): 731
Minimum response time (ms): 205
Maximum response time (ms): 14335
Requests per second Throughput (requests/second): 3.647108681650448
The request log was successfully saved to logs/request_logs.csv
```

All 10,000 requests were successfully sent. The minimum latency was 205 ms, and the maximum latency was 14,335 ms.

From the SQS charts below, we can see that in the 'Number Of Messages Deleted' and 'Approximate Age Of Oldest Message' graphs, the consumer can delete messages from the SQS queue very quickly. There are fewer than 10 messages present in SQS at any given time.

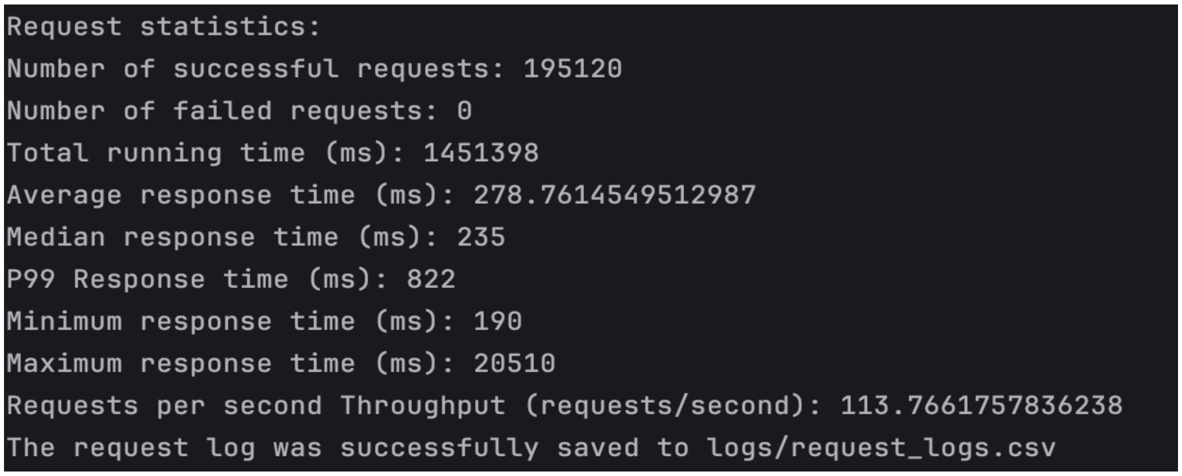
AWS SQS Console Monitoring Screenshot:



Load Balancing Test Results

The client sends 200,000 requests to the server.
CSV Link: https://github.com/Orange-135/cs6650/blob/main/hw2/logs/request_logs_200k.csv

Output:



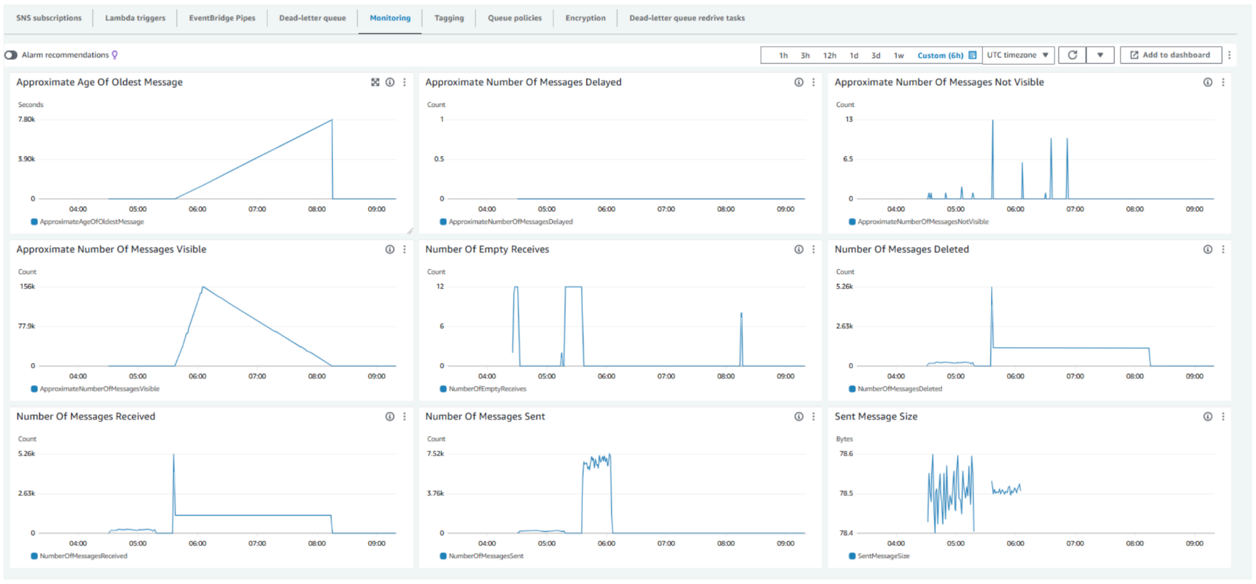
```
Request statistics:
Number of successful requests: 195120
Number of failed requests: 0
Total running time (ms): 1451398
Average response time (ms): 278.7614549512987
Median response time (ms): 235
P99 Response time (ms): 822
Minimum response time (ms): 190
Maximum response time (ms): 20510
Requests per second Throughput (requests/second): 113.7661757836238
The request log was successfully saved to logs/request_logs.csv
```

Minimum latency was 190 ms, and maximum latency was 20,510 ms.

This shows abnormal response times for a small number of requests. Such long durations are due to instantaneous high loads or delays from external dependencies, indicating that the system may encounter performance bottlenecks under extreme conditions.

The system's average response time is low, and most requests respond quickly; however, under very high traffic volumes, significant delays can occur.

AWS SQS Console Monitoring Screenshot:



System Optimization

Run the Client on the Server's Internal Network

- Running the client within the server's internal network can reduce latency and speed up request processing.
- When deploying the client and server, choose a network configuration with high bandwidth to further accelerate request transmission.
- Ensure that the internal network's DNS resolution speed is fast to avoid increasing resolution time when communicating between different servers.

Increase the Number of Servers for Load Balancing

- Adding more servers and distributing traffic through load balancing can reduce the pressure on a single server and improve overall system throughput.
- Use auto-scaling strategies (such as AWS Auto Scaling) to dynamically increase or decrease the number of servers based on traffic, ensuring enough servers are available to handle requests during peak times.

- Set up health checks to ensure the load balancer can identify and bypass unhealthy instances, reducing request latency.
- If requests come from different regions, deploy servers in the nearest region to the user to reduce network transmission latency.
- Utilize SQS's support for batch sending mechanisms to reduce the number of times requests reach SQS, thereby improving overall system response efficiency.

Increase vCPU on Consumer Instances to Speed Up Consumption

- With a higher number of vCPUs, the consumer can use more threads to consume messages concurrently, achieving efficient consumption.
- Leverage SQS's batch consumption feature to retrieve multiple messages at once and process them in parallel across multiple threads, speeding up overall consumption.
- While increasing vCPUs, ensure there is sufficient memory to support high-concurrency operations to avoid throughput degradation due to insufficient memory.

Note1:

If you still have questions, please visit my AWS Academy account to check specific information, Thanks. The account information is as follows (open in Google guest mode).
Link: https://awsacademy.instructure.com/courses/93410/modules/items/8609034
Email: cheng.yul@northeastern.edu
Password: Wodeshengri0829!
Region: us-west-2

Note2:

I used JAR packaging (which shouldn't be much different from war), and I packaged the server, client, and consumer all as JAR files. However, Assignment 1 requires the server to be packaged as a WAR, so I re-packaged the server as a war and you can see two packaging files for the server on GitHub.