Cloud-Deployed Distributed Radiograph Processing System Using AWS

Project Overview

This project presents the design and implementation of a cloud-based distributed application that performs asynchronous processing of medical radiograph images. The system is fully deployed on Amazon Web Services (AWS) and integrates multiple distributed system components, including messaging, persistent storage, and a relational database. The application accepts processing requests through a RESTful API, distributes jobs using a managed messaging service, performs background image processing on worker nodes, and persists both generated artifacts and metadata for later retrieval.

The system is intentionally designed to reflect a real-world healthcare imaging workflow while strictly adhering to project constraints. No serverless functions, containers, Kubernetes, or service mesh technologies are used. All application logic runs on virtual machines, and all interactions occur through explicitly provisioned cloud services.

Real-World Motivation and Relevance

In modern medical imaging environments, radiographs and other imaging studies are frequently processed by downstream systems such as computer-aided detection tools or AI-based diagnostic pipelines. These workflows require asynchronous job handling, reliable message delivery, scalable processing, and durable storage of both images and results. Tight coupling between ingestion and processing can lead to performance bottlenecks and system fragility.

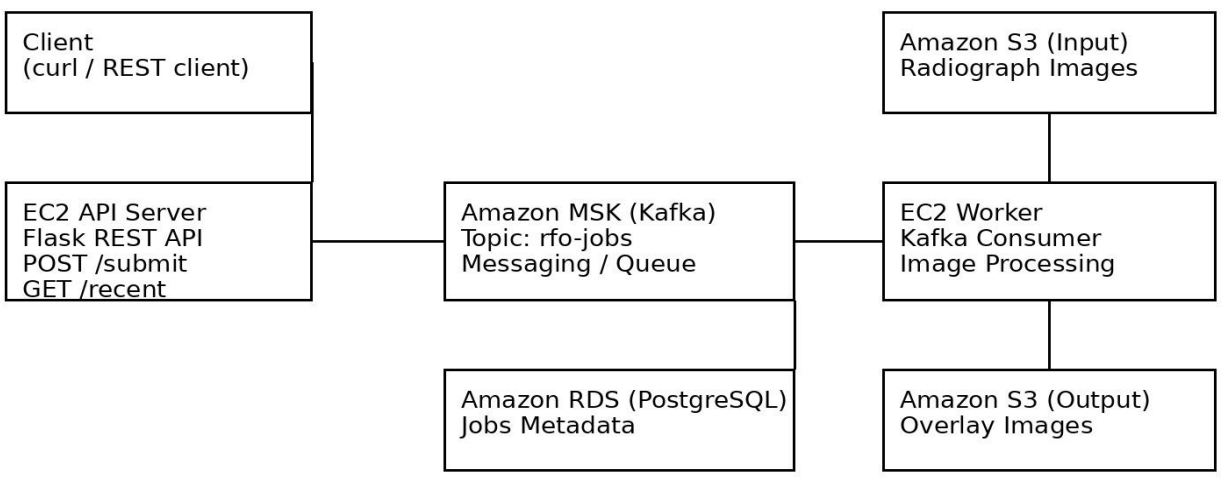
This project models such a workflow by simulating a radiograph post-processing pipeline. While the image processing logic itself is a placeholder that generates a random bounding box overlay, the surrounding infrastructure mirrors real production systems used in healthcare, research, and large-scale data processing environments. The architecture can be directly extended to incorporate real machine learning inference without structural changes.

System Architecture

The application follows a decoupled, event-driven architecture. A lightweight API layer handles client requests and publishes processing jobs to a messaging system. Worker nodes consume these jobs asynchronously, perform image processing, store derived artifacts in object storage, and record metadata in a relational database. Clients can later query the system for recent processing results.

Architecture Description

The API server runs on an Amazon EC2 instance and exposes REST endpoints for job submission and result retrieval. When a client submits a request, the API publishes a message containing the image identifier to an Apache Kafka topic hosted on Amazon MSK. This messaging layer decouples request handling from processing, enabling asynchronous execution and buffering of workload spikes.



A separate EC2 worker instance subscribes to the Kafka topic and processes incoming jobs. For each message, the worker downloads the corresponding radiograph image from an Amazon S3 input bucket, applies a placeholder image processing step that generates a bounding box overlay, and uploads the resulting image to an S3 output bucket. The worker then records job metadata, including timestamps, image locations, processing status, and bounding box coordinates, in an Amazon RDS PostgreSQL database.

The API server also connects to the database and exposes a retrieval endpoint that allows clients to query recent processing results. This design ensures that clients can retrieve results independently of the processing lifecycle.

End-to-End Execution Flow

Radiograph images are first uploaded to an S3 input bucket. A client submits a processing request by calling the API with the image key. The API immediately returns a success response after publishing the job to Kafka, without waiting for processing to complete. The worker asynchronously consumes the job, processes the image, uploads the generated overlay to S3, and inserts a record into the database. Clients can subsequently query the API to retrieve recent job results, including links to the processed images.

This demonstrates a fully operational end-to-end distributed workflow, from request ingestion through asynchronous processing to persistent storage and retrieval.

Scalability and Design Considerations

The architecture supports horizontal scalability by design. Multiple worker instances can be added to consume messages from the Kafka topic in parallel, increasing throughput without changes to the API layer. The API server is stateless, enabling additional instances to be deployed behind a load balancer if required. The messaging layer provides durability and back-pressure handling, while managed storage and database services ensure persistence and reliability.

The use of managed cloud services reduces operational overhead while preserving explicit architectural control, making the system both scalable and maintainable.

Source Code Organization

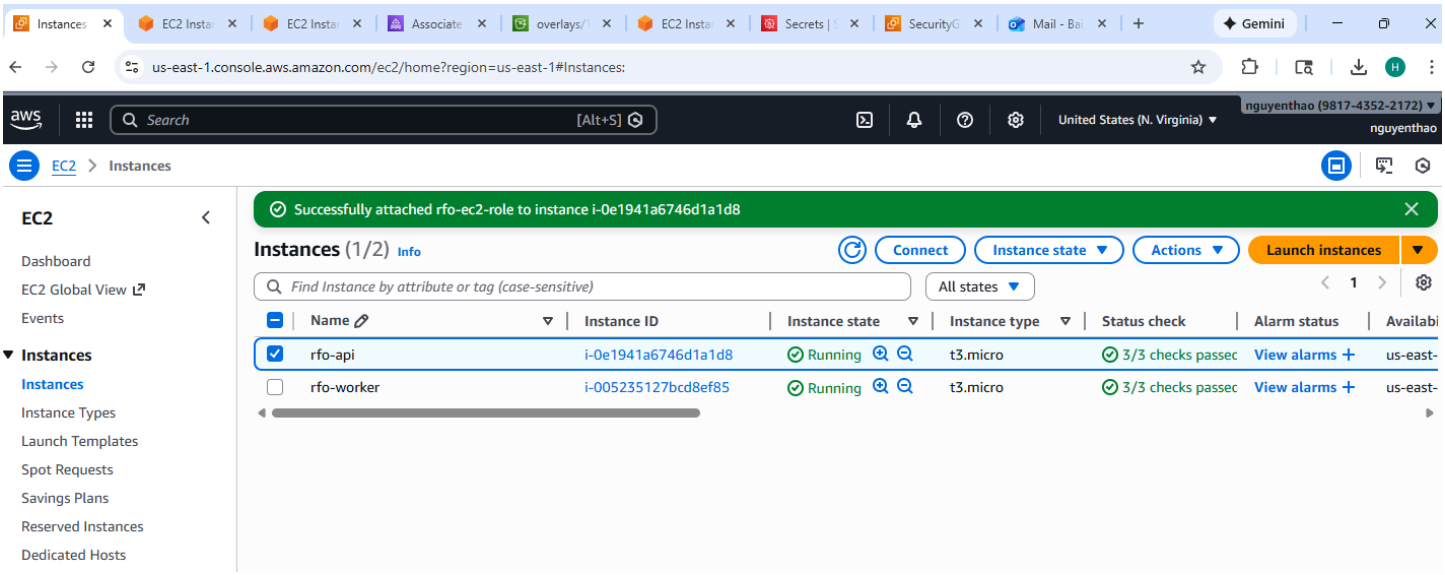
The project consists of two primary Python applications. The API application implements REST endpoints and Kafka producer logic, while the worker application implements Kafka consumer logic, image processing, S3 interaction, and database insertion. Configuration is handled entirely through environment variables, and no credentials are hard-coded. The code is structured for readability and extensibility, following best practices for distributed applications.

Deployment Evidence and System Validation

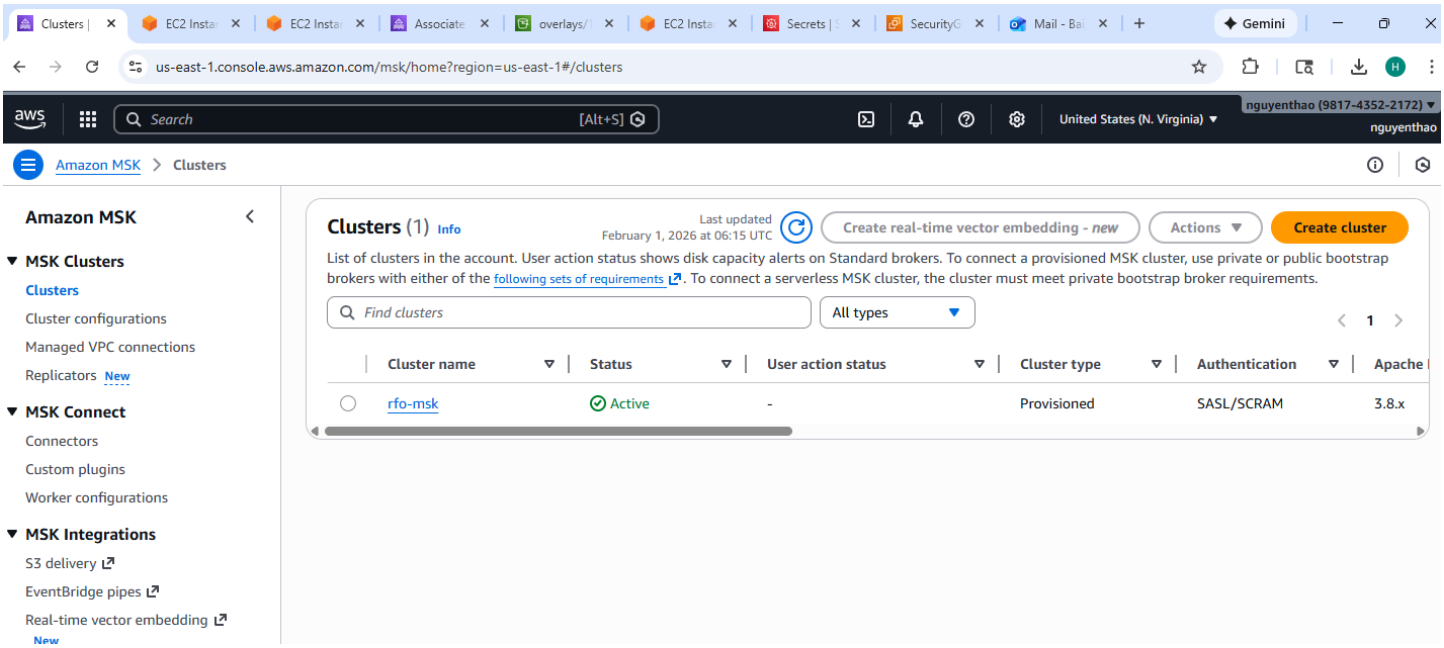
This section provides concrete deployment evidence demonstrating that the distributed application is fully operational in the cloud and that all system components interact correctly from input ingestion through asynchronous processing to result persistence and retrieval. The evidence includes infrastructure screenshots, API execution results, database records, and visual examples of both input and output images.

Cloud Infrastructure Deployment

The application is deployed entirely on Amazon Web Services using multiple managed and compute services. Two Amazon EC2 instances are used: one hosting the REST API server and one hosting the background worker responsible for asynchronous image processing. An Amazon MSK cluster provides the messaging and queuing backbone of the system, while Amazon S3 and Amazon RDS provide durable storage for images and metadata, respectively.



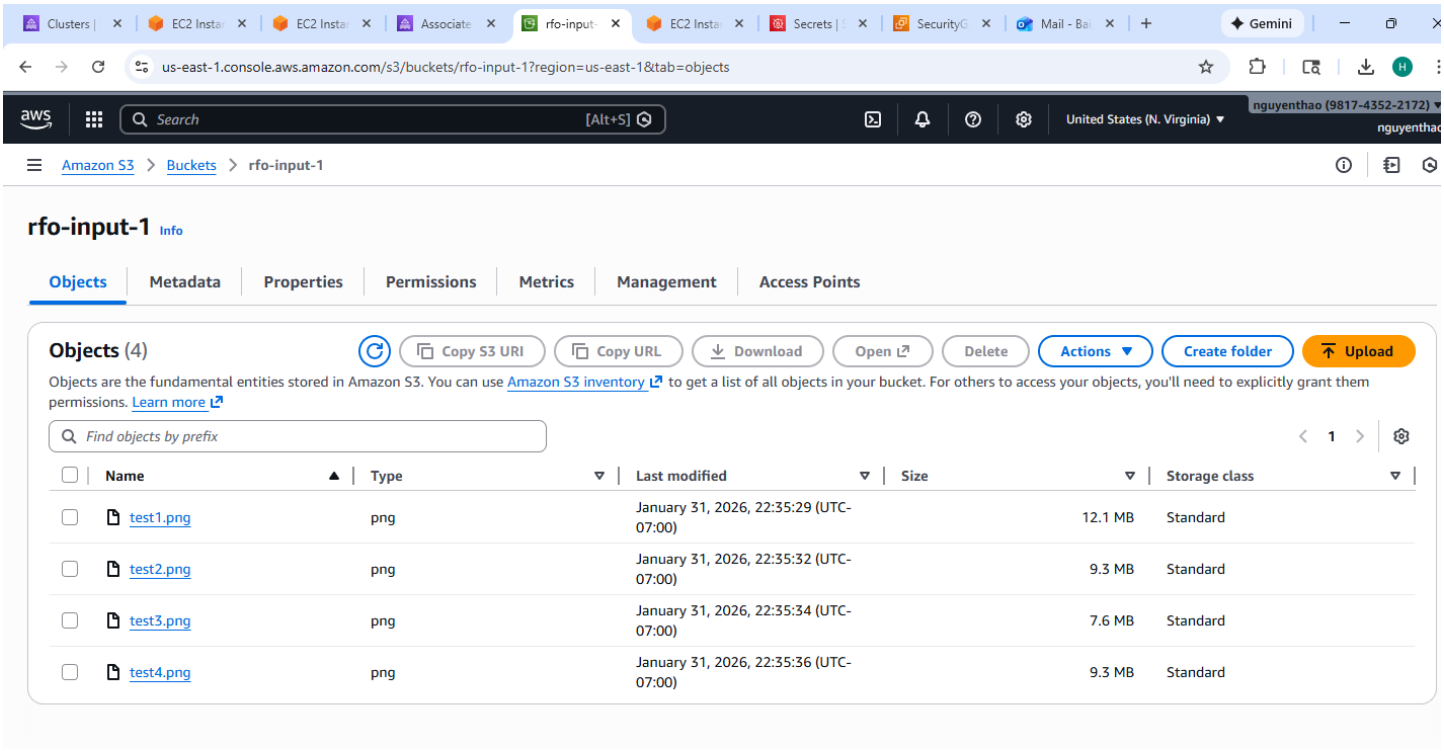
Screenshot: AWS EC2 console showing both the API instance (rfo-api) and the worker instance (rfo-worker) in a running state.



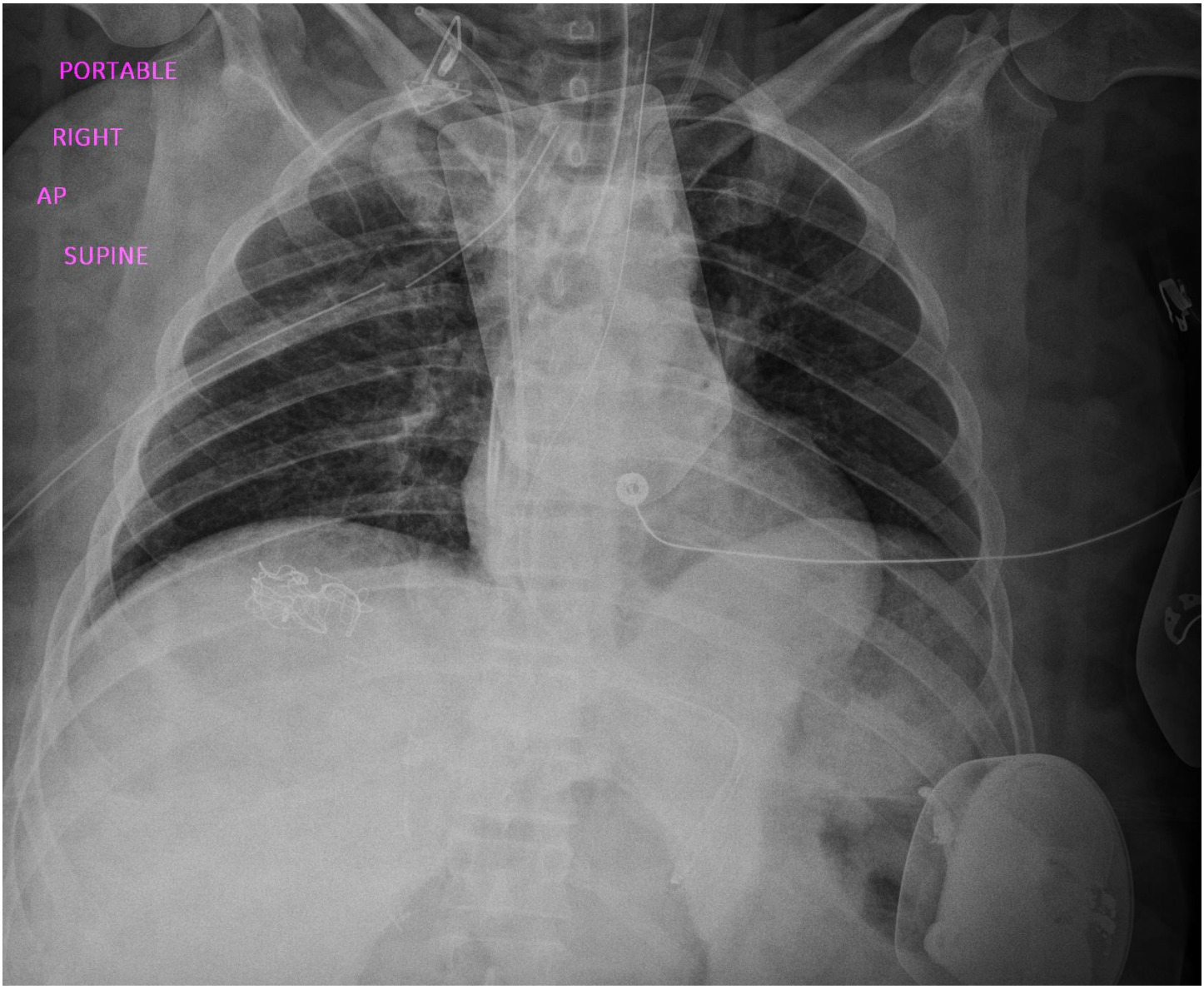
Screenshot: Amazon MSK cluster details page showing cluster status as Active and the Kafka topic rfo-jobs.

Input Data Storage

Radiograph images are stored in an Amazon S3 input bucket prior to processing. These images represent the raw data consumed by the distributed system. The use of S3 ensures high durability and availability of input data and reflects common storage practices in medical imaging pipelines.



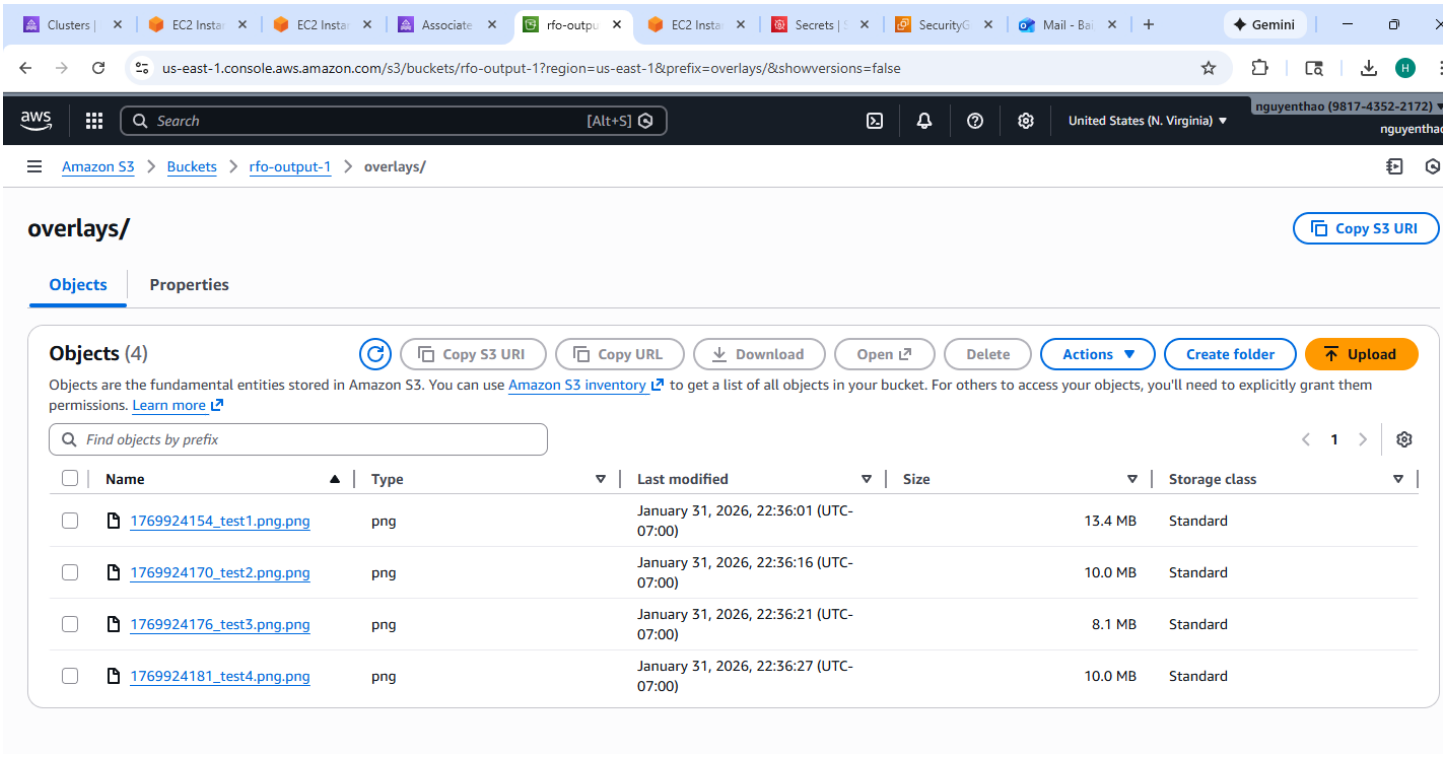
Screenshot: Amazon S3 input bucket listing radiograph image files.



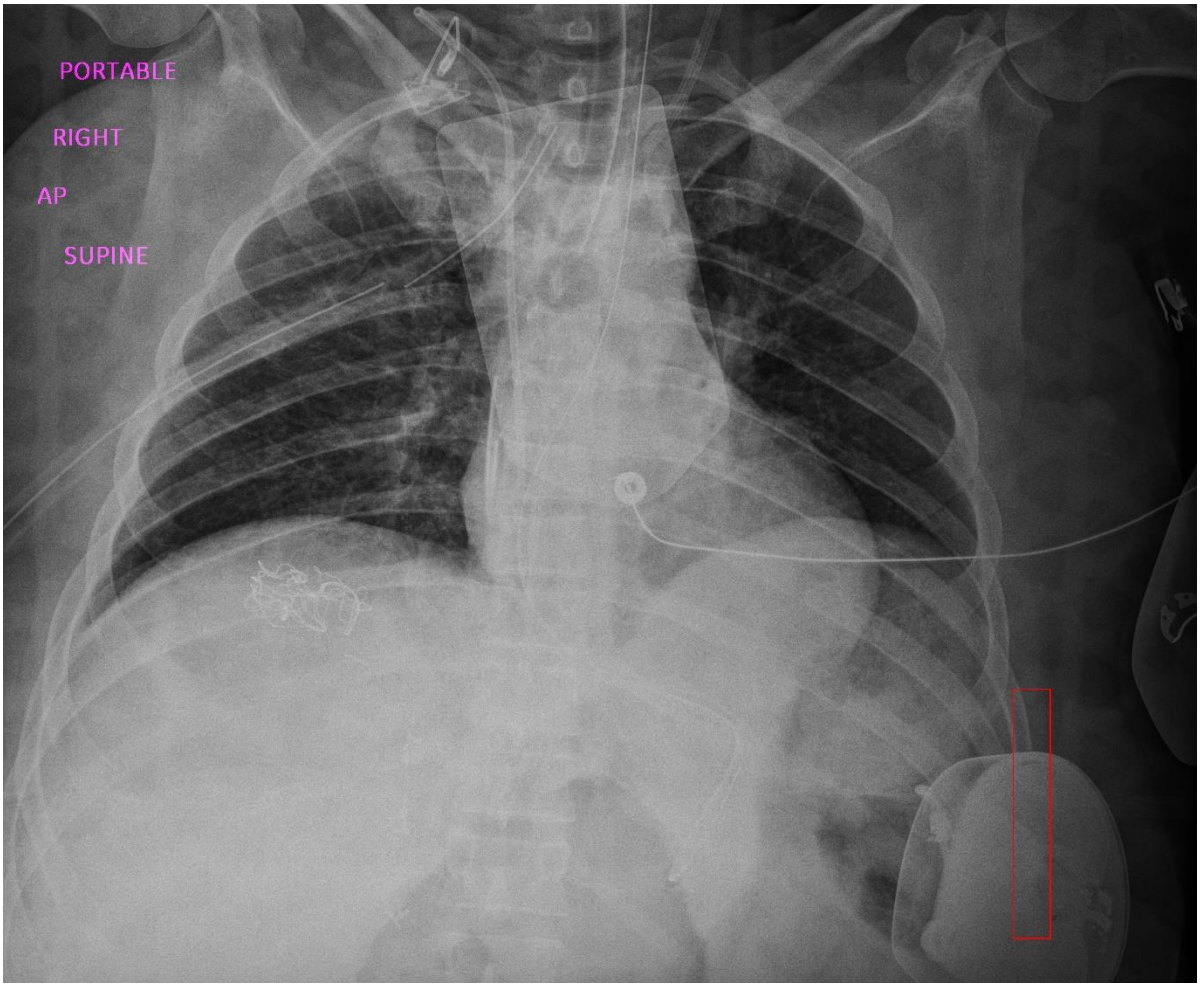
Sample input radiograph image stored in the Amazon S3 input bucket prior to processing.

Asynchronous Processing and Output Generation

Upon job submission, the API publishes a message to the Kafka topic. The worker instance asynchronously consumes the message, downloads the corresponding input image from S3, applies a placeholder image processing operation that generates a random bounding box overlay, and uploads the processed image to an Amazon S3 output bucket. This confirms successful message delivery, background execution, and artifact generation.

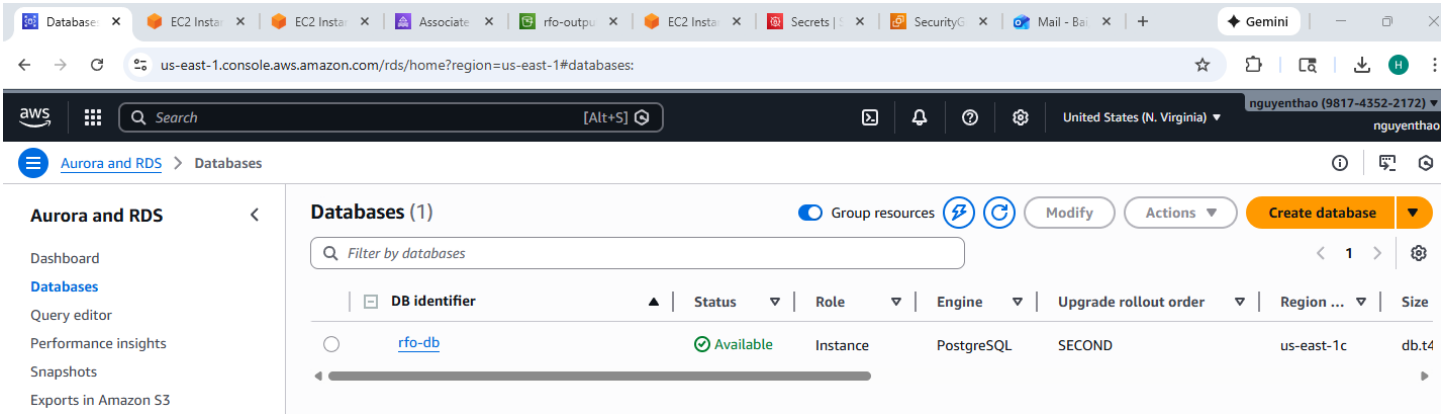


Screenshot: Amazon S3 output bucket showing generated overlay images.



Metadata Persistence in Database

For each processed image, the worker inserts a corresponding record into a PostgreSQL database hosted on Amazon RDS. The database stores the input image key, output overlay key, processing status, bounding box metadata, and timestamps. This enables reliable tracking and querying of processing results independent of the worker lifecycle.



Screenshot: Amazon RDS PostgreSQL instance details page.

```
t":"2026-02-01T05:36:00.497732+00:00","image_key":"test1.png","overlay_key":"overlays/1769924154_test1.png.png","status":"done"}]
[ec2-user@ip-172-31-24-98 ~]$ psql -h rfo-db.c2ji6kskup56.us-east-1.rds.amazonaws.com \
     -U rfo_app \
     -d rfoapp
Password for user rfo_app:
psql (15.15, server 17.6)
WARNING: psql major version 15, server major version 17.
         Some psql features might not work.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

rfoapp=> SELECT created_at, image_key, overlay_key, status
FROM jobs
ORDER BY created_at DESC
LIMIT 5;
         created_at          | image_key |           overlay_key            | status
-----------------------------+-----------+----------------------------------+--------
 2026-02-01 05:36:26.470485+00 | test4.png | overlays/1769924181_test4.png.png | done
 2026-02-01 05:36:20.749704+00 | test3.png | overlays/1769924176_test3.png.png | done
 2026-02-01 05:36:15.609263+00 | test2.png | overlays/1769924170_test2.png.png | done
 2026-02-01 05:36:00.497732+00 | test1.png | overlays/1769924154_test1.png.png | done
(4 rows)

rfoapp=>
```

Screenshot: SQL query output showing rows in the jobs table containing recent processing records.

API Interaction and Result Retrieval

The REST API provides endpoints for both job submission and result retrieval. The successful execution of these endpoints demonstrates that the API layer, messaging system, and database layer are correctly integrated

and functioning as a cohesive distributed application.

```
      _/m/'
Last login: Sun Feb  1 03:44:34 2026 from 18.206.107.29
[ec2-user@ip-172-31-24-98 ~]$ curl -X POST http://localhost:8000/submit -H "Content-Type: application/json" -d '{"image_key":"test.png"}'
{"image_key":"test.png","queued":true,"topic":"rfo-jobs"}
[ec2-user@ip-172-31-24-98 ~]$ curl "http://localhost:8000/recent?n=5"
[]
[ec2-user@ip-172-31-24-98 ~]$ curl -X POST http://localhost:8000/submit \
  -H "Content-Type: application/json" \
  -d '{"image_key":"test.png"}'
{"image_key":"test.png","queued":true,"topic":"rfo-jobs"}
[ec2-user@ip-172-31-24-98 ~]$ aws s3 ls s3://rfo-input-1/
2026-02-01 04:11:16   12700825 Image 2_patient 1.png
2026-02-01 04:11:22    9778249 Image 3_patient 1.png
2026-02-01 04:11:18    7940914 image 2_patient 2.png
2026-02-01 04:11:20    9778249 image 3_patient 2.png
[ec2-user@ip-172-31-24-98 ~]$ curl -X POST http://localhost:8000/submit \
  -H "Content-Type: application/json" \
  -d '{"image_key":"image2_patient 1.png"}'
{"image_key":"image2_patient 1.png","queued":true,"topic":"rfo-jobs"}
[ec2-user@ip-172-31-24-98 ~]$ curl -X POST http://localhost:8000/submit   -H "Content-Type: application/json"   -d '{"image_key":"test1.png"}'
{"image_key":"test1.png","queued":true,"topic":"rfo-jobs"}
[ec2-user@ip-172-31-24-98 ~]$ curl -X POST http://localhost:8000/submit   -H "Content-Type: application/json"   -d '{"image_key":"test2.png"}'
{"image_key":"test2.png","queued":true,"topic":"rfo-jobs"}
[ec2-user@ip-172-31-24-98 ~]$ curl -X POST http://localhost:8000/submit   -H "Content-Type: application/json"   -d '{"image_key":"test3.png"}'
{"image_key":"test3.png","queued":true,"topic":"rfo-jobs"}
[ec2-user@ip-172-31-24-98 ~]$ curl -X POST http://localhost:8000/submit   -H "Content-Type: application/json"   -d '{"image_key":"test4.png"}'
{"image_key":"test4.png","queued":true,"topic":"rfo-jobs"}
[ec2-user@ip-172-31-24-98 ~]$
```

Screenshot: Terminal output showing successful POST /submit request returning a queued job response.

```
[ec2-user@ip-172-31-24-98 ~]$ curl -X POST http://localhost:8000/submit   -H "Content-Type: application/json"   -d '{"image_key":"test4.png"}'
{"image_key":"test4.png","queued":true,"topic":"rfo-jobs"}
[ec2-user@ip-172-31-24-98 ~]$ curl "http://localhost:8000/recent?n=5"
[{"box":{"x1":1068,"x2":2221,"y1":3006,"y2":3336},"created_at":"2026-02-01T05:36:26.470485+00:00","image_key":"test4.png","overlay_key":"overlays/1769924181
_test4.png.png","status":"done"},{"box":{"x1":3331,"x2":3445,"y1":3047,"y2":3401},"created_at":"2026-02-01T05:36:20.749704+00:00","image_key":"test3.png","o
verlay_key":"overlays/1769924176_test3.png.png","status":"done"},{"box":{"x1":4128,"x2":4163,"y1":2647,"y2":3357},"created_at":"2026-02-01T05:36:15.609263+0
0:00","image_key":"test2.png","overlay_key":"overlays/1769924170_test2.png.png","status":"done"},{"box":{"x1":3581,"x2":3715,"y1":2427,"y2":3313},"created_a
t":"2026-02-01T05:36:00.497732+00:00","image_key":"test1.png","overlay_key":"overlays/1769924154_test1.png.png","status":"done"}]
[ec2-user@ip-172-31-24-98 ~]$
```

Screenshot: Terminal output showing successful GET /recent request returning processed job records from the database.

Worker Execution Evidence

The worker instance logs each successfully processed job, including the input image key, generated output key, and bounding box metadata. These logs provide direct evidence that messages are consumed from Kafka and processed end-to-end.

```
[ec2-user@ip-172-31-30-135 ~]$ python3 work.py
/home/ec2-user/.local/lib/python3.9/site-packages/boto3/compat.py:89: PythonDeprecationWarning: Boto3 will no longer support Python 3.9 starting April 29, 2026. To continue receiving service updates, bug fixes,
 and security updates please upgrade to Python 3.10 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)
Worker started. Waiting for messages...
Processed test1.png -> overlays/1769924154_test1.png.png box={'x1': 3581, 'y1': 2427, 'x2': 3715, 'y2': 3313}
Processed test2.png -> overlays/1769924170_test2.png.png box={'x1': 4128, 'y1': 2647, 'x2': 4163, 'y2': 3357}
Processed test3.png -> overlays/1769924176_test3.png.png box={'x1': 3331, 'y1': 3047, 'x2': 3445, 'y2': 3401}
Processed test4.png -> overlays/1769924181_test4.png.png box={'x1': 1068, 'y1': 3006, 'x2': 2221, 'y2': 3336}
```

Screenshot: Worker terminal output showing successful image processing messages.

Summary of Deployment Evidence

Together, the screenshots and images presented in this section confirm that the distributed system is fully deployed and operational in the cloud. The evidence demonstrates correct interaction between the API server, messaging queue, worker nodes, storage services, and database, as well as successful execution of the complete business workflow from input image ingestion to output generation and result retrieval.

Python code files are attached