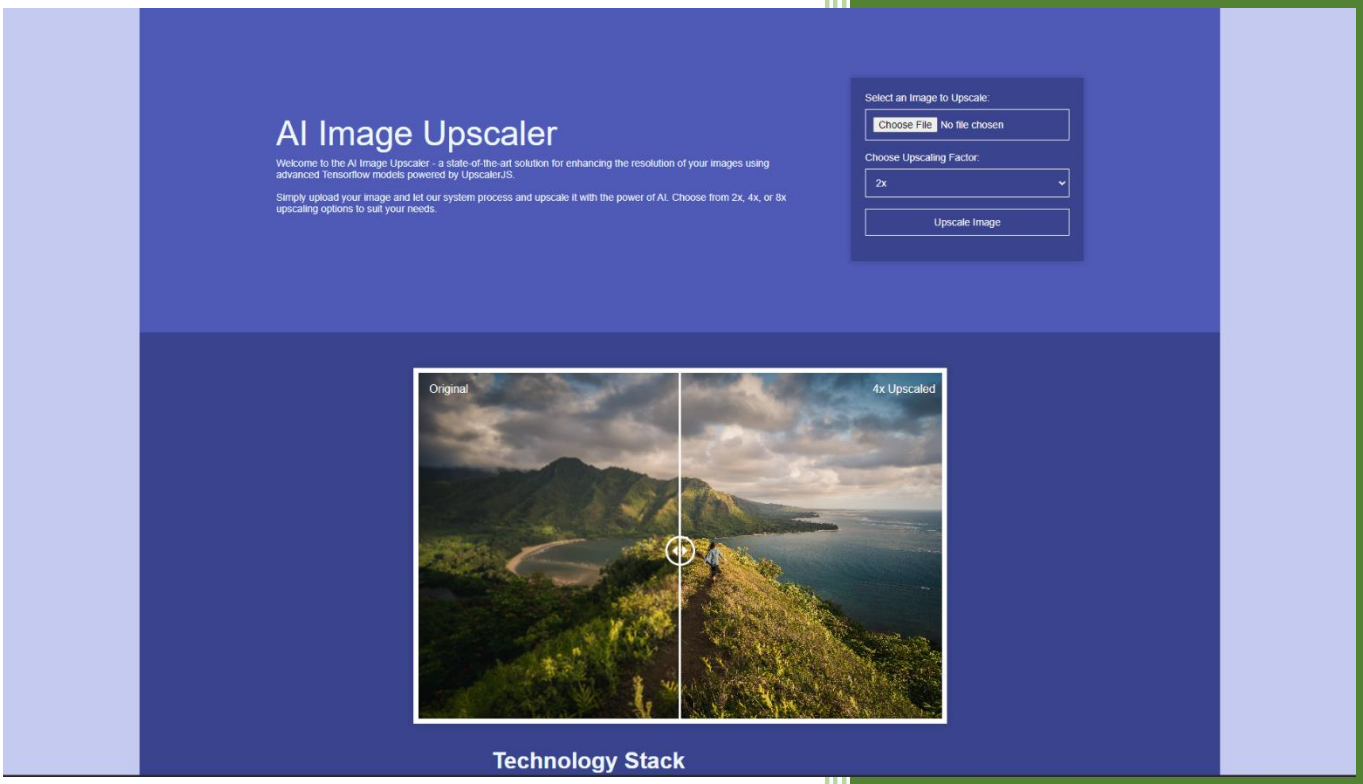
2023

AI Upscale Cloud Project



CAB432

Assignment 2 Cloud Project

Jacob Fallows 10749527

Rhys Boyd 10993444

11/20/2023

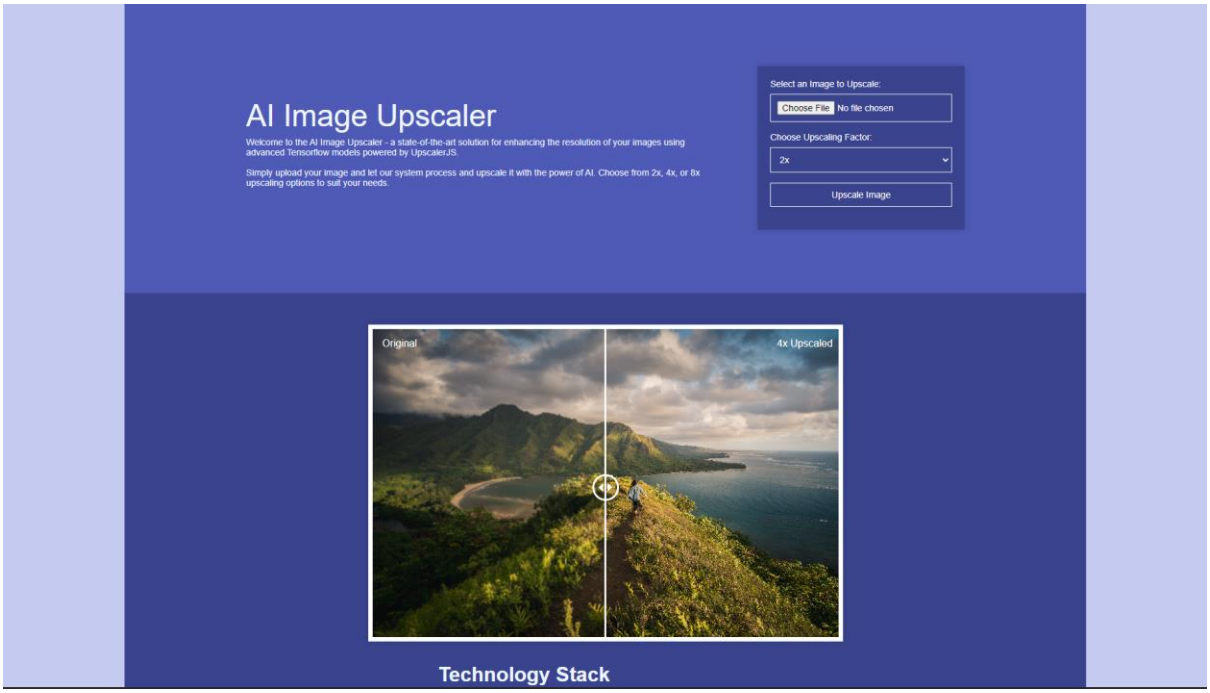
Contents

Introduction 2
Purpose & description 2
Services used 2
UpscalerJS (v 1.0.0-beta.18) 2
Bootstrap (v 5.3) 2
Redis 2
Amazon EC2 Auto Scaling 2
Use cases 3
Technical breakdown 4
Architecture 4
Context diagram 4
Client / server demarcation of responsibilities 4
Response filtering / data object correlation 5
Scaling and Performance 5
Test plan 6
Difficulties / Exclusions / unresolved & persistent errors 6
User guide 7
References 9
Appendices 9
1 - Home Page 9
2 – View Loading Page 9
3 – Result Page 10
4 – API Result 10
5 – Image Fetch (Redis & S3) 10
6 – Information Fetch (Redis & S3) 10
7 - Architecture 11
8 – Stress Test Program (collapsed) 11
9 – Scaling and performance 13

Introduction

Purpose & description

The primary goal of our web application is to simplify the process of image enhancement (upscaling) allowing for users to easily utilize various artificial intelligence models to upscale images. As a result, the application allows users to simply enhance blurry, and or low-resolution photos allowing users to quickly enhance the quality of images.



Services used

UpscalerJS (v 1.0.0-beta.18)

UpscalerJS is a tool for enhancing images in JavaScript using AI. It can run in the browser, Node.js, and in Worker environments. UpscalerJS uses machine learning models like Tensorflow for upscaling.

Docs: https://upscalerjs.com/documentation/

Bootstrap (v 5.3)

Bootstrap is a powerful tool with extensive front-end CSS and JavaScript libraries.

Docs: https://getbootstrap.com/docs/5.3

Redis

Redis is an open-source key-value database designed to operate distributed and in-memory making it a good candidate for caching.

Docs: https://redis.io/docs/

Amazon EC2 Auto Scaling

Amazon EC2 Auto Scaling allows for the rapid scaling of EC2 instances (virtual machines) to facilitate for the dynamic handling of load based on real-time demand.

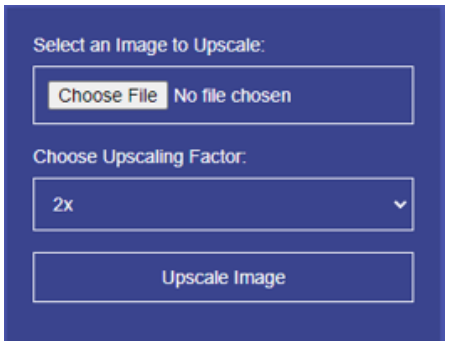
Docs: https://aws.amazon.com/ec2/autoscaling/

Use cases

1. As a **user** I want the system to **allow for me to upload images to be queued to be upscaled in accordance with my selected settings** so that **I can increase the resolution of my images.**
2. As a **user** I want the system to **redirect me to a loading page and automatically redirect me to the image once it has finished upscaling** so that **I don't need to manually refresh the page.**
3. As a **user** I want the system to **display upscaled images once they are upscaled utilizing the pre-generated URL** so that **I can view the image once it has finished processing.**

With the current design, the application has a monolithic design with the application being scaled utilizing images of EC2 instances utilizing Instance Auto Scaling for Amazon EC2. The application allows for the user to upload an image from their device which is sent to the application for processing. Once received, the application will utilize a worker_thread to upscale the image without interrupting the performance of the application. Depending on the number of users and the number of images sent the application will scale out to fit demand. The images and associated information are stored on Amazon S3 for persistence alongside cached on individual Redis instances on each EC2 instance.

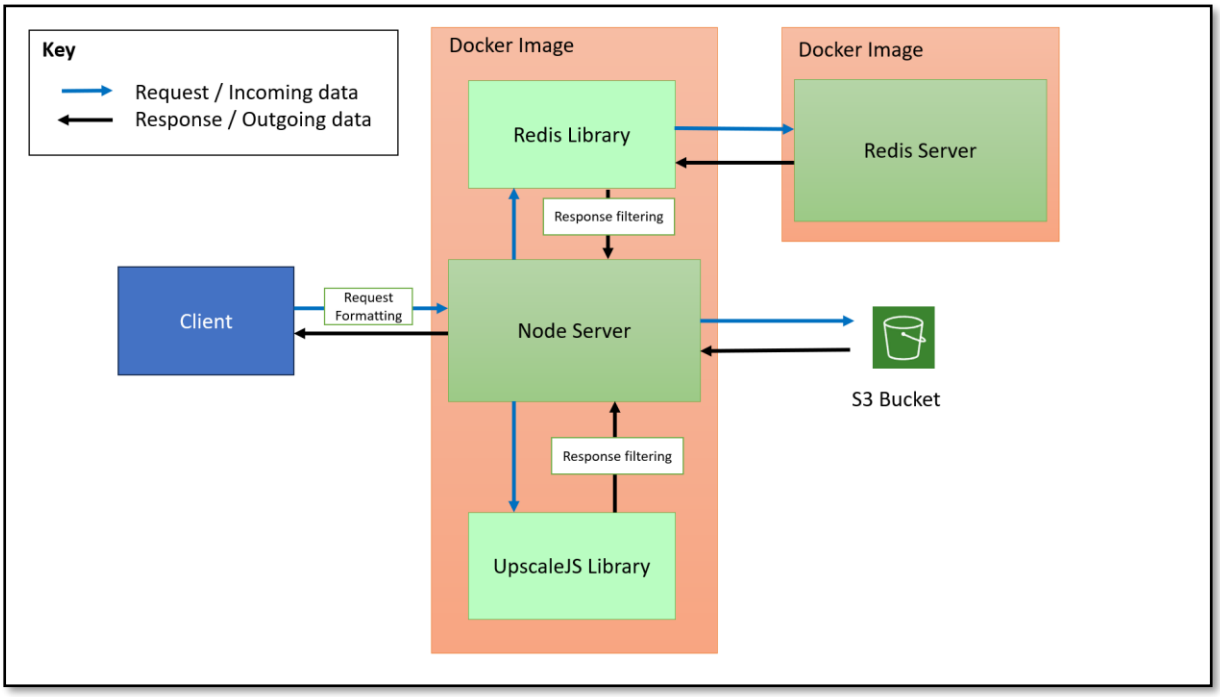
The upscaling of an image requires significant resources such as CPU and RAM in our current situation. Some of our tests showed for example 8 GB of RAM can handle a 2x upscale on a 1920x1080 pixel photo although the results are dependent on your upscaling factor, image resolution, alongside the utilized AI model.



Technical breakdown

Architecture

Context diagram



Client / server demarcation of responsibilities

The workflow starts at the client's request to the load balancer. The load balancer then redirects the request towards one of the available and healthy EC2 instances within the Auto Scaling group running an Express Node server. GET is utilized for the home page. With post being utilized to request for an image to be upscaled with the request containing the image alongside other required information. The initial request uses an HTML form from the home page to prompt the user to make a request. Once one of the EC2 servers receives the image, it checks if the image file is already in the S3 bucket or Redis Cache. If not, it sends it to the cache and creates a worker thread to carry out the upscaling. Upscaling is facilitated via the UpscaleJS library which provides multiple models and options. While the server is processing the image, the end user is redirected to a results page. If the image is still processing, the page will display loading with a background task checking if the image has loaded. If the image has been processed, it will show the upscaled image to the end-user. In the appendices, there is the source code for data going between the S3 bucket and the local Redis server. Depending on the amount of client requests the AWS load balancer will create additional EC2 instances to respond to current demand. The application is persistent and exhibits statelessness; If an EC2 instance becomes unhealthy or the user logs on from another machine their data will be preserved with the Instance Auto Scaling automatically assessing the health of instances regularly. Unexpected behaviour could potentially be experienced if the local Redis instance of the EC2 instance you are hitting goes offline, and or the S3 bucket utilized for persistent storage goes offline.

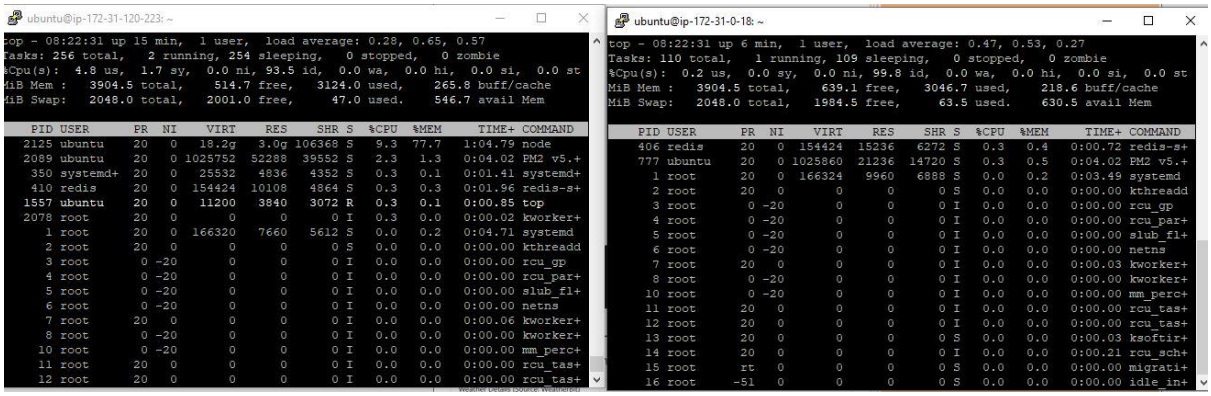
Response filtering / data object correlation

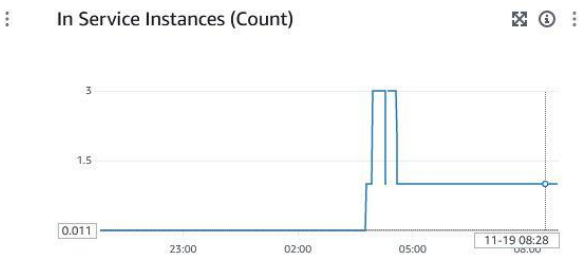
From the S3 bucket, the JSON data is filtered by getting the body, converting it to a string, and formatting it with UTF-8. Since this system does not use a typical API there can be no filtering. The same idea is used for the Redis cache, and the Upscaler is just an internal node library.

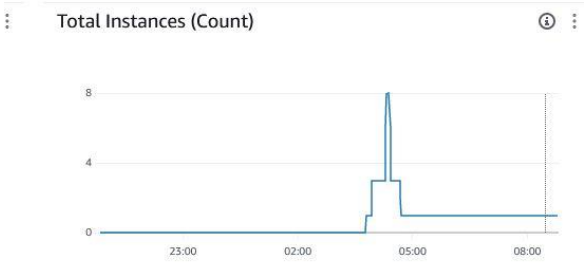
```
try {
    const data = await this.s3.getObject(params).promise();
    return data.Body.toString("utf-8");
```

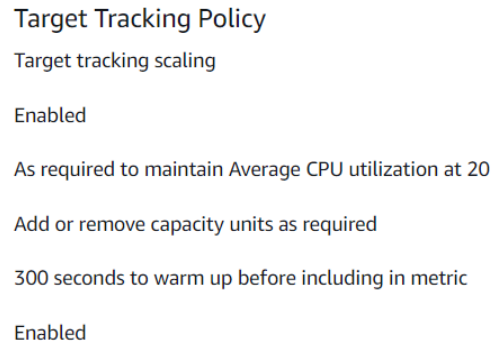
Scaling and Performance

During the development of the application, a script attached in the the appendix, was created to automate the process of stress testing the application. The application allows for configurable demand allowing for us to gauge how the application responds to different loads.









When running the stress tester, it can be seen that the Automatic Scaling in the Auto Scaling Group automatically creates new instances to account for the load. Currently, the auto-scaling group is set to 20% CPU utilization. The load balancer will create up to 3 instances depending on the load. In the appendix are additional screenshots of scaling and performance metrics.

Test plan

Test	Expected Outcome	Result	Appendix
View Homepage	Homepage displayed, option to upload image for upscaling available.	PASS	1
View Loading Page	Load page is displayed while waiting for an image to get upscaled.	PASS	2
View Result	The upscaled image displayed.	PASS	3
View upscale result	View JSON information regarding information on an upscale task.	PASS	4
Result image Redis fetch	Result images are fetched from Redis if available.	PASS	5
Result information Redis fetch	Result information is fetched from Redis if available.	PASS	6

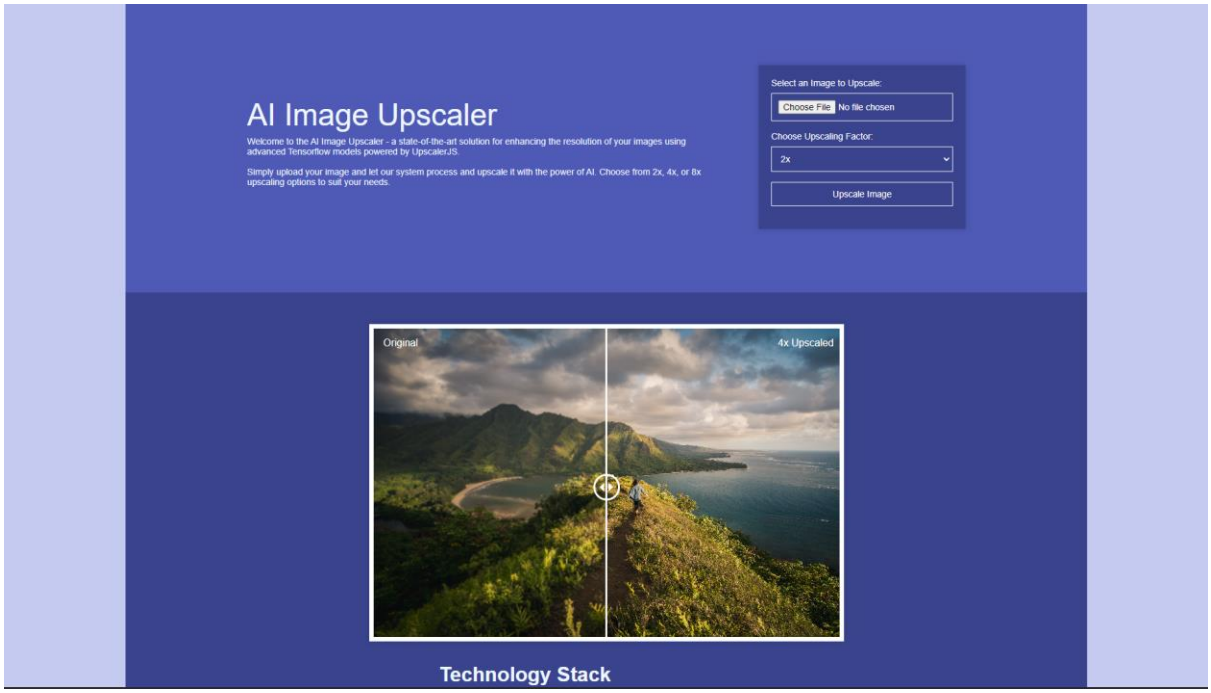
Difficulties / Exclusions / unresolved & persistent errors /

During the development of the application, many issues were encountered:

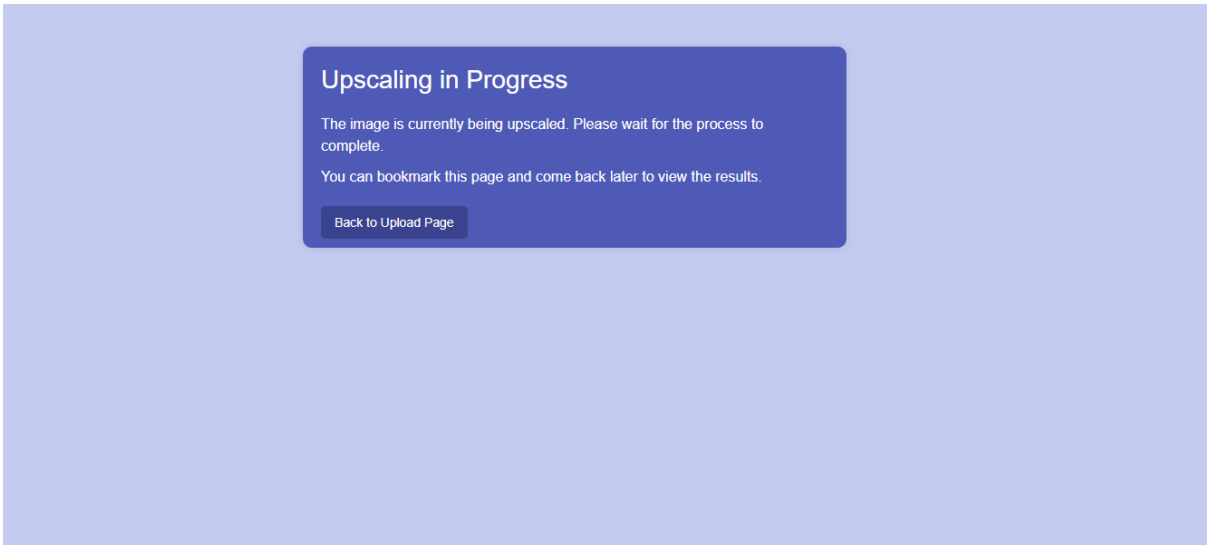
1. UpscalerJS library throwing unexpected errors with certain images:
    a. The Problem: Early testing of the UpscalerJS library showed that attempts to upscale certain types of PNG resulted in errors from the Tensorflow node when attempting to execute the model. We suspect the issue was a result of PNG images having a fourth column for opaqueness.
    b. The Solution: Use a library called Jimp to convert all images to JPEG automatically to make images uploaded compatible with the UpscalerJS library.
2. Installation of Redis on Windows during development:
    a. The Problem: I was unable to get Ubuntu WSL to install from the Windows Store due to various issues with initializing the WSL subsystem.
    b. The Solution: After carrying out research, I found out that I had to change a BIOS setting on my computer to allow for WSL to start. After changing the setting, uninstalling and reinstalling WSL allowed it to successfully work. As a result, I was able to get Redis to function correctly in a Windows environment.
3. Getting IAM roles functioning to allow automatic authentication of the AWS-SDK node module:
    a. The Problem: Getting the AWS-SDK to automatically authenticate without manually providing AWS credentials that quickly expire.
    b. The Solution: Adding the pre-existing ec2-service-role provided the permissions required for the EC2 machines to access and administrate S3 storage utilizing the AWS-SDK.
4. Memory constraints of provided EC2 instances limiting the AI models we could utilize:
    a. The Problem: Due to the high memory requirements of AI Upscaling of images, the amount of RAM provided on a t2.micro EC2 instance was insufficient.
    b. The Solution: As a result, we upgraded to t2.medium instances for our workload. Once we had done this, we still had issues with high-resolution images under certain circumstances utilizing too much RAM. As a result, for demonstration purposes, we changed the AI model from esrgan-thick to esrgan-slim which is primarily designed for in-brower clientside upscaling for the demonstration on AWS. In the real world, EC2 instances with GPU resources would be utilized to resolve this issue.

5. Issues installing the @tensorflow/tjfs-node library on Windows.
    a. The Problem: When attempting to install the library, NPM would error out with various errors. The first error was due to no prebuilt binaries being available for the library. When NPM would then attempt to build the library, it would fail due to multiple reasons including my PC having Python 3 instead of 2.
    b. The Solution: Researching on the internet showed this was a common issue. Another user said they had luck with the library after downgrading to Node 18.16.1 and manually installing a specific version of Python. Doing this appeared to resolve the issue.
    c. Additional Information: This issue only seemed to occur on my laptop running Windows 11, however, didn't appear to occur on my Windows 10 desktop. Additionally, we had no issues installing the library on Linux.

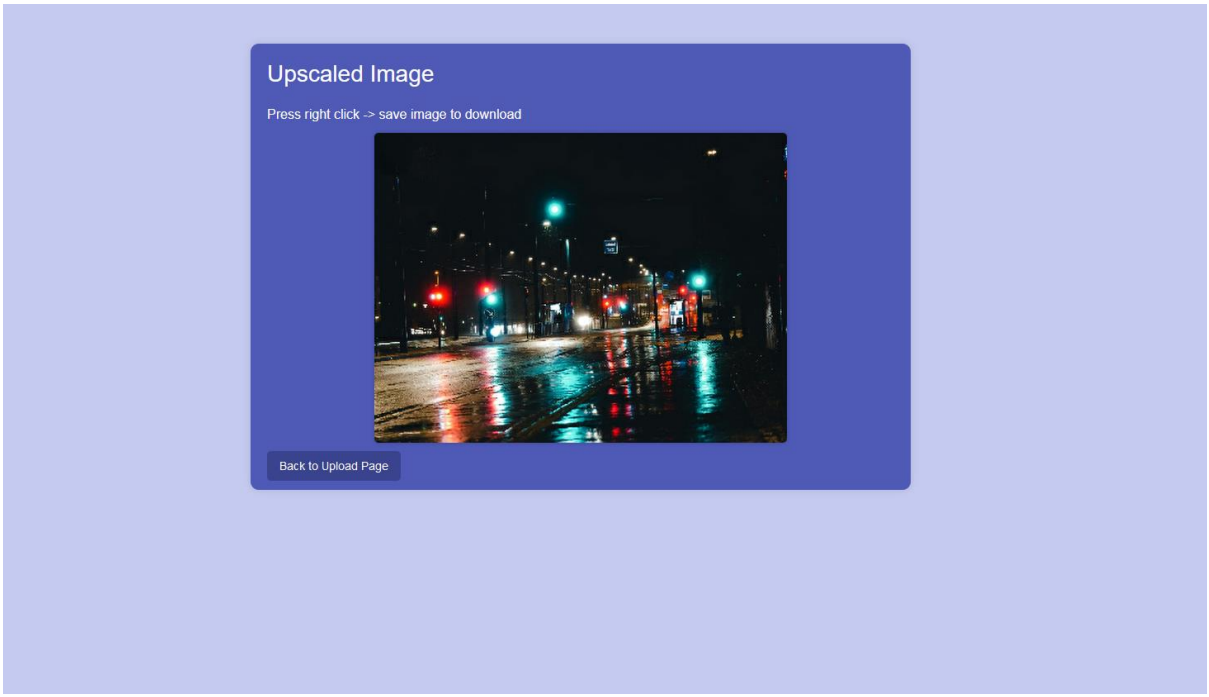
User guide



1. Home page: Press choose file to select an image for upscaling. You can select an upscaling factor but currently, only 2x upscaling is supported. Move the slider below to see how the upscaling looks.
2. You will be prompted with a loading page while the image is being upscaled. This should only take a couple of seconds and you will be redirected to see the result.



3. Press right click -> save image to download your result!



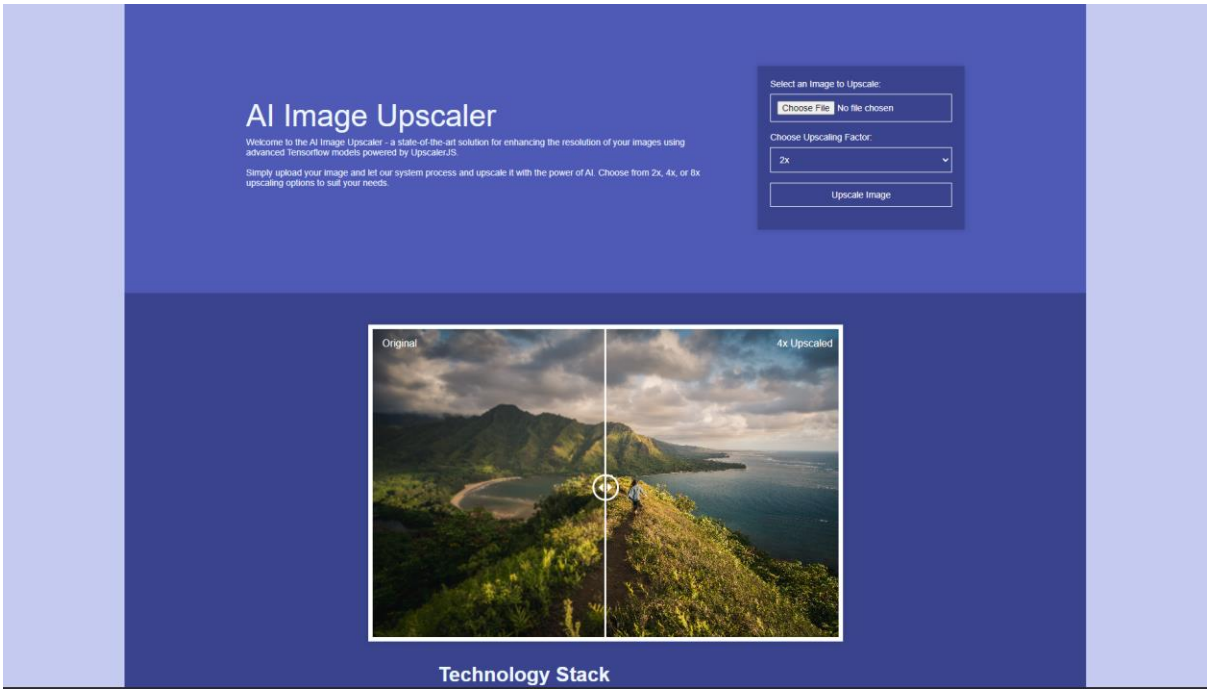
References

Amazon Web Services, Inc. (2023). *AWS SDK for JavaScript*. Retrieved from Amazon Documents: https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/welcome.html

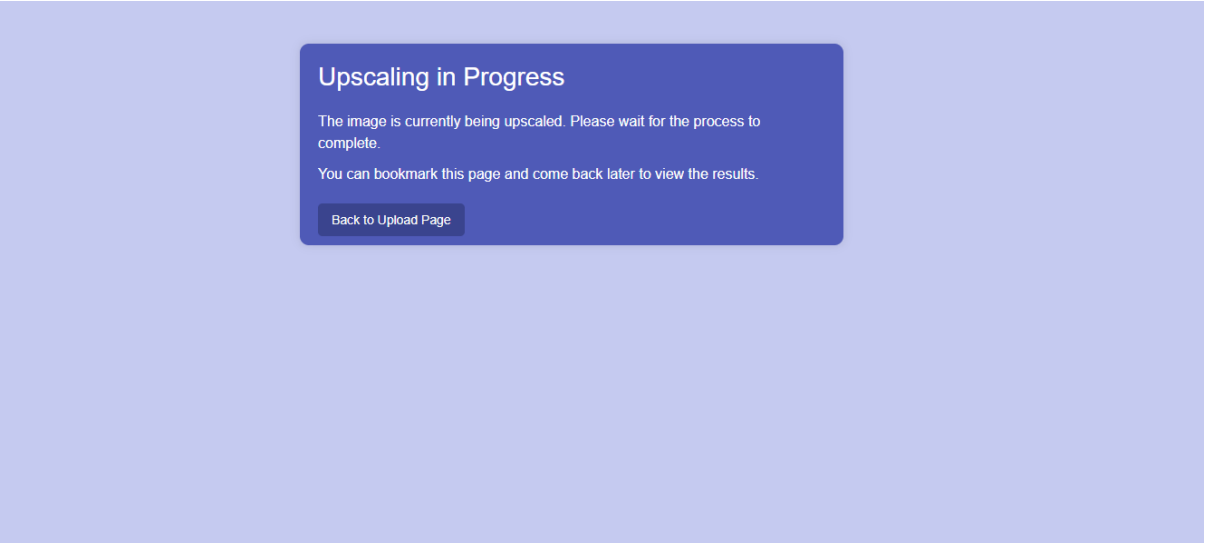
Scott, K. (2023). *Docs - Introduction*. Retrieved from UpscalerJS: https://upscalerjs.com/documentation

Appendices

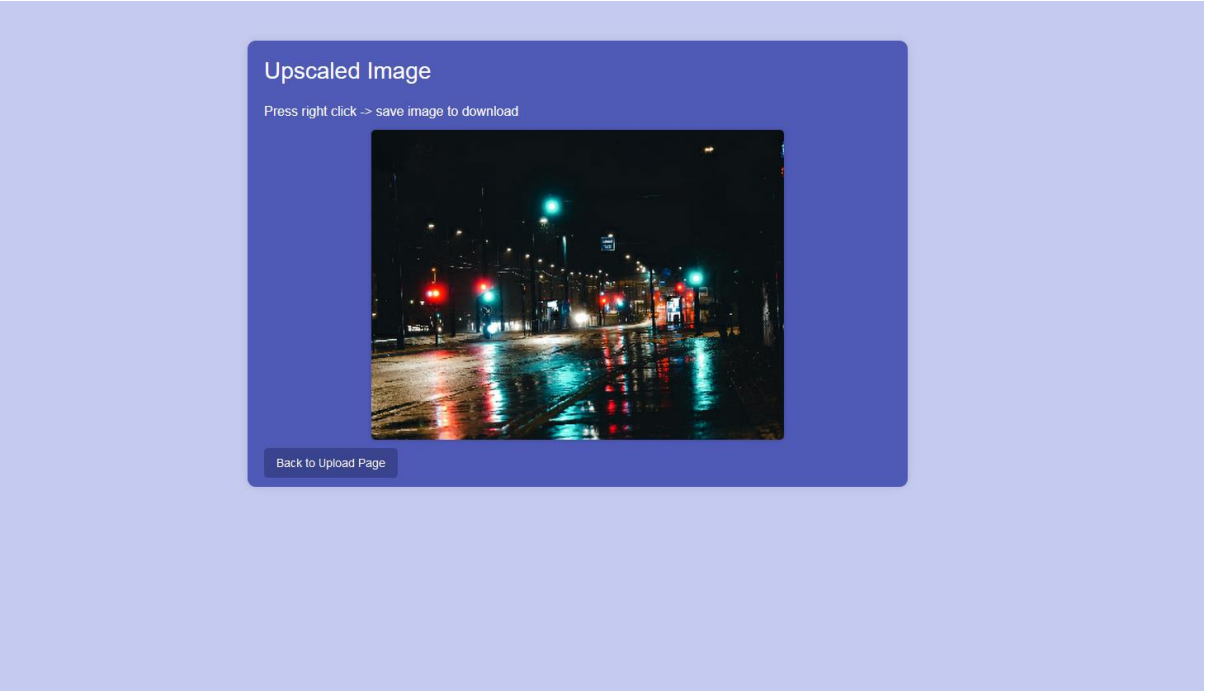
1 - Home Page



2 – View Loading Page



3 – Result Page



4 – API Result

5 – Image Fetch (Redis & S3)

```
a476869330b5552b6ec45742090a3a55bcadc8b2916c1150aefc763e43c79431 found in S3, uploaded to Redis and displayed
GET /image/original/a476869330b5552b6ec45742090a3a55bcadc8b2916c1150aefc763e43c79431 200 214.026 ms - 119676
GET /favicon.ico - - ms - -
a476869330b5552b6ec45742090a3a55bcadc8b2916c1150aefc763e43c79431 found in Redis and displayed
GET /image/original/a476869330b5552b6ec45742090a3a55bcadc8b2916c1150aefc763e43c79431 200 3.466 ms - 119676
```

6 – Information Fetch (Redis & S3)

```
Information for image a476869330b5552b6ec45742090a3a55bcadc8b2916c1150aefc763e43c79431 found in S3 and uploaded to Redis
GET /api/result/a476869330b5552b6ec45742090a3a55bcadc8b2916c1150aefc763e43c79431 200 261.528 ms - 239
Information for image a476869330b5552b6ec45742090a3a55bcadc8b2916c1150aefc763e43c79431 found in Redis
GET /api/result/a476869330b5552b6ec45742090a3a55bcadc8b2916c1150aefc763e43c79431 200 2.864 ms - 242
```

7 - Architecture

S3 Incoming Data

```
//Upload image
if (!await this.uploadFile(this.bucket, imageKey, image, "image/png")) {
    return false;
}

//Upload JSON
if (!await this.uploadFile(this.bucket, jsonKey, jsonInformation, "application/json")) {
    return false;
}
```

S3 Outgoing Data

```
const data = await this.s3.getObject(params).promise();
return data.Body.toString("utf-8");
```

Redis Incoming Data

```
await this.redisClient.set(key, value);
console.log(`Data set for key: ${key}`);
return true;
```

Redis Outgoing Data

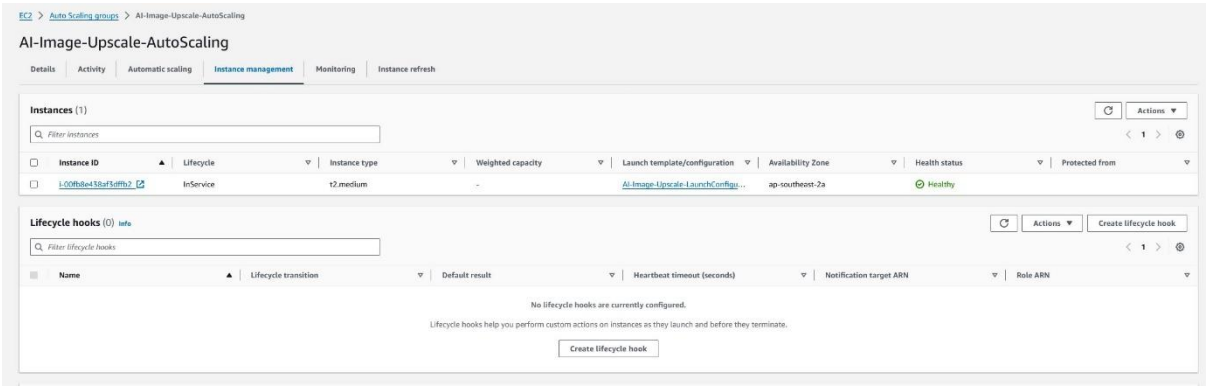
```
const data = await this.redisClient.get(key);
return data;
```

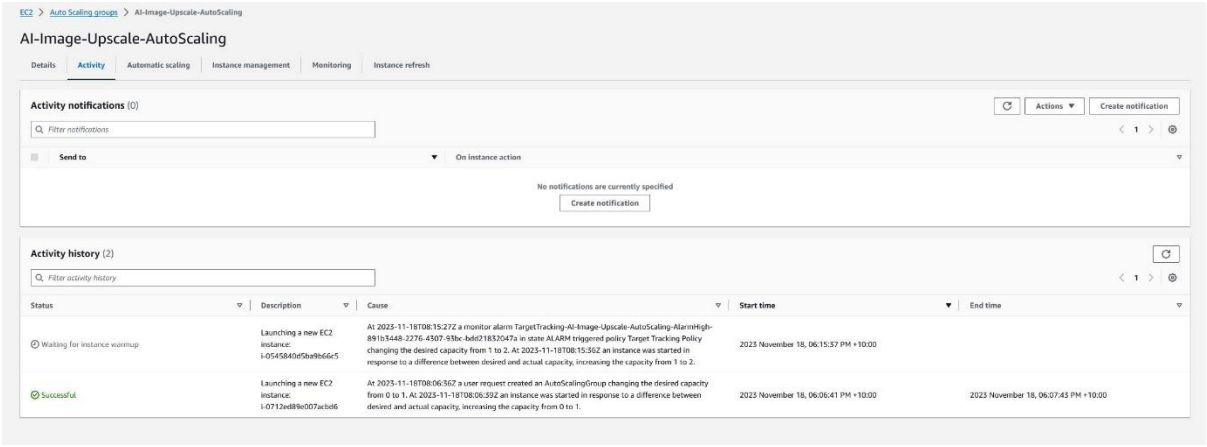
8 – Stress Test Program (collapsed)

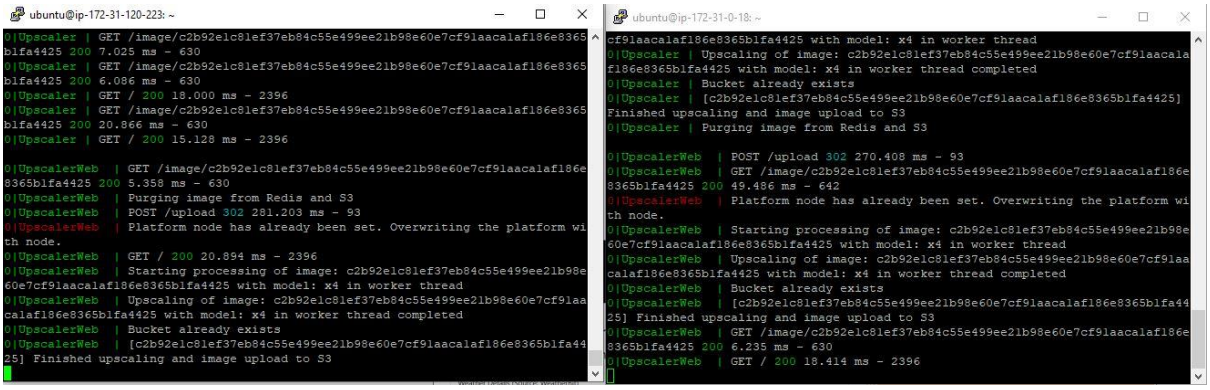
```
const axios = require('axios');
const fs = require('fs');
const FormData = require('form-data');

const TARGET_URL = 'http://13.211.165.92:3000/upload';
const REQUESTS_PER_BATCH = 3
const TOTAL_BATCHES = 500;
const BATCHES_FOR_AVERAGE = 1;

const makeRequest = async () => {
  //Simulate a file upload request from the browser
  try {
    const form = new FormData();
    form.append('upscalingOption', 'x4');
    form.append('ignoreCache', 'on');
    const file = fs.readFileSync('test.png');
    form.append('image', file, 'test.png');

    const response = await axios.post(TARGET_URL, form, {
      headers: form.getHeaders()
    });
```

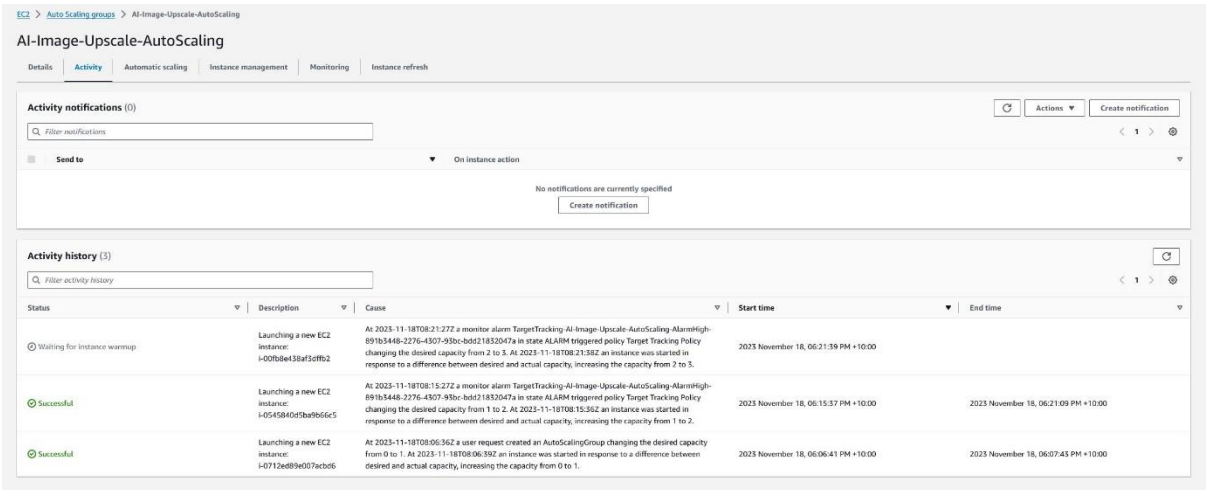
```
      if (response.status !== 200) {
        console.log(response.status);
      }
    } catch (error) {
      console.error(`Error: ${error.response ? error.response.status : error.message}`);
    }
};

const spamApi = async () => {
    let totalElapsedTime = 0;

    for (let batch = 0; batch < TOTAL_BATCHES; batch++) {
      console.log(`Sending batch #${batch + 1}`);

      const startTime = Date.now();

      let promises = [];
      for (let i = 0; i < REQUESTS_PER_BATCH; i++) {
        promises.push(makeRequest());
      }

      await Promise.all(promises);

      const endTime = Date.now();
      const elapsedTime = (endTime - startTime) / 1000; // in seconds
      totalElapsedTime += elapsedTime;

      //Sleep for 5 seconds
      await new Promise(resolve => setTimeout(resolve, 3000));

      console.log(`Finished batch #${batch + 1}`);

      if ((batch + 1) % BATCHES_FOR_AVERAGE === 0) {
        const averageRate = (REQUESTS_PER_BATCH * BATCHES_FOR_AVERAGE * 60) /
totalElapsedTime; // requests per minute
        totalElapsedTime = 0; // reset elapsed time counter
      }
    }
};

spamApi();
```

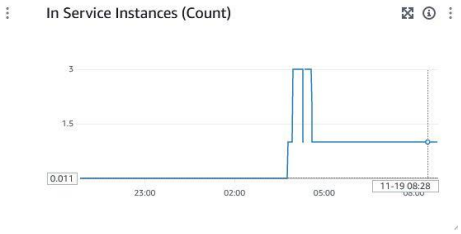
9 – Scaling and performance

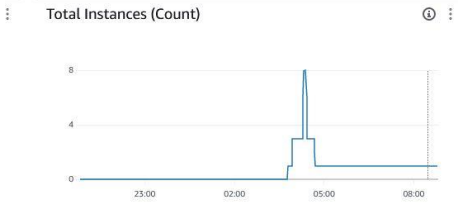












EC2 > Auto Scaling groups > AI-Image-Upscale-AutoScaling

AI-Image-Upscale-AutoScaling

Details | Activity | Automatic scaling | Instance management | Monitoring | Instance refresh

Activity notifications (0)

Filter notifications

Actions | Create notification

Send to	On instance action

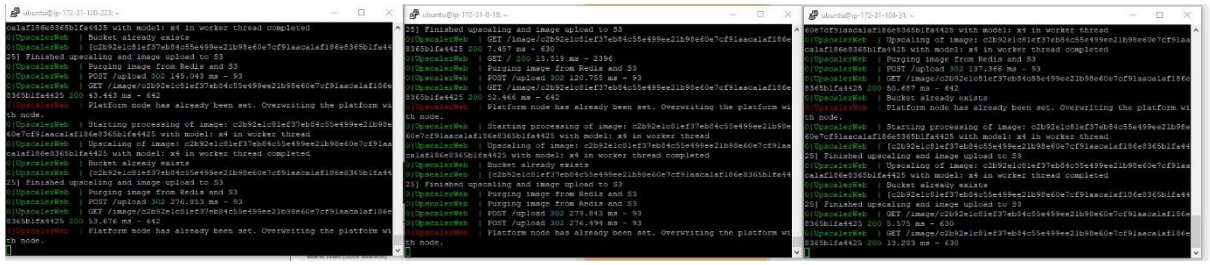
No notifications are currently specified

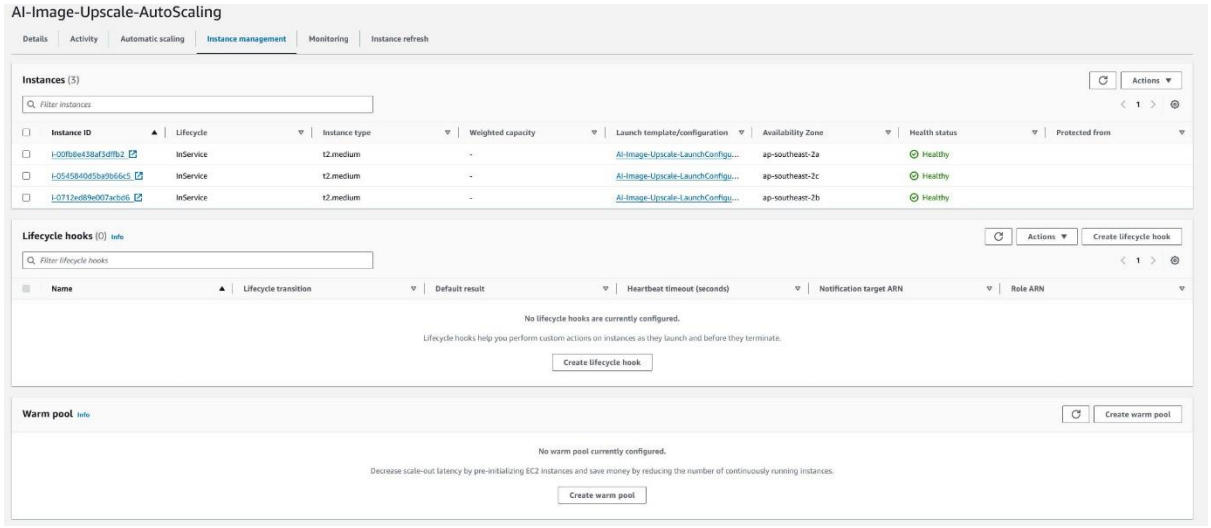
Create notification

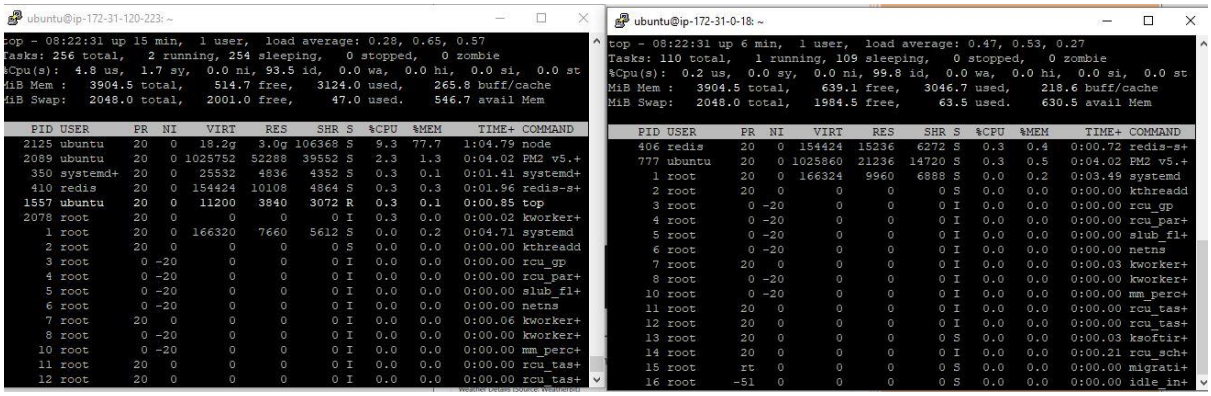
Activity history (5)

Filter activity history

Status	Description	Cause	Start time	End time
Successful	Terminating EC2 instance: i-0545840d5ba9b66c5	At 2023-11-18T08:45:22Z a monitor alarm TargetTracking-AI-Image-Upscale-AutoScaling-AlarmLow-7a622099-ddcb-4943-ad6e-42636cdf4447 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 1. At 2023-11-18T08:45:27Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 2 to 1. At 2023-11-18T08:45:27Z instance i-0545840d5ba9b66c5 was selected for termination.	2023 November 18, 06:45:27 PM +10:00	2023 November 18, 06:51:18 PM +10:00
Successful	Terminating EC2 instance: i-0712ed89e007acbd6	At 2023-11-18T08:44:22Z a monitor alarm TargetTracking-AI-Image-Upscale-AutoScaling-AlarmLow-7a622099-ddcb-4943-ad6e-42636cdf4447 in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 3 to 2. At 2023-11-18T08:44:26Z an instance was taken out of service in response to a difference between desired and actual capacity, shrinking the capacity from 3 to 2. At 2023-11-18T08:44:26Z instance i-0712ed89e007acbd6 was selected for termination.	2023 November 18, 06:44:26 PM +10:00	2023 November 18, 06:50:11 PM +10:00
Successful	Launching a new EC2 instance: i-00ffb8e438af3cffb2	At 2023-11-18T08:21:27Z a monitor alarm TargetTracking-AI-Image-Upscale-AutoScaling-AlarmHigh-891b5448-2276-4507-93bc-bdd21832047a in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 2 to 3. At 2023-11-18T08:21:38Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 2 to 3.	2023 November 18, 06:21:39 PM +10:00	2023 November 18, 06:27:11 PM +10:00
Successful	Launching a new EC2 instance: i-0545840d5ba9b66c5	At 2023-11-18T08:15:27Z a monitor alarm TargetTracking-AI-Image-Upscale-AutoScaling-AlarmHigh-891b5448-2276-4507-93bc-bdd21832047a in state ALARM triggered policy Target Tracking Policy changing the desired capacity from 1 to 2. At 2023-11-18T08:15:36Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 1 to 2.	2023 November 18, 06:15:37 PM +10:00	2023 November 18, 06:21:09 PM +10:00
Successful	Launching a new EC2 instance: i-0712ed89e007acbd6	At 2023-11-18T08:06:36Z a user request created an AutoScalingGroup changing the desired capacity from 0 to 1. At 2023-11-18T08:06:39Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 0 to 1.	2023 November 18, 06:06:41 PM +10:00	2023 November 18, 06:07:43 PM +10:00




AI-Image-Upscale-AutoScaling
Details
Activity
Automatic scaling
Instance management
Monitoring
Instance refresh
Instances (3)
Actions
Filter instances
Instance ID
Lifecycle
Instance type
Weighted capacity
Launch template/configuration
Availability Zone
Health status
Protected from
i-00fb8e438af3dffb2
InService
t2.medium
AI-Image-Upscale-LaunchConfigu...
ap-southeast-2a
Healthy
i-05458040d5ba9b66c5
InService
t2.medium
AI-Image-Upscale-LaunchConfigu...
ap-southeast-2c
Healthy
i-0712ed89e007acbc6
InService
t2.medium
AI-Image-Upscale-LaunchConfigu...
ap-southeast-2b
Healthy
Lifecycle hooks (0) Info
Actions
Create lifecycle hook
Filter lifecycle hooks
Name
Lifecycle transition
Default result
Heartbeat timeout (seconds)
Notification target ARN
Role ARN
No lifecycle hooks are currently configured.
Lifecycle hooks help you perform custom actions on instances as they launch and before they terminate.
Create lifecycle hook
Warm pool Info
Create warm pool
No warm pool currently configured.
Decrease scale-out latency by pre-initializing EC2 instances and save money by reducing the number of continuously running instances.
Create warm pool


ubuntu@ip-172-31-120-223: ~
top - 08:22:31 up 15 min, 1 user, load average: 0.28, 0.65, 0.57
Tasks: 256 total, 2 running, 254 sleeping, 0 stopped, 0 zombie
%Cpu(s): 4.8 us, 1.7 sy, 0.0 ni, 93.5 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 3904.5 total, 514.7 free, 3124.0 used, 265.8 buff/cache
MiB Swap: 2048.0 total, 2001.0 free, 47.0 used. 546.7 avail Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
2125 ubuntu 20 0 18.2g 3.0g 106368 S 9.3 77.7 1:04.79 node
2089 ubuntu 20 0 1025752 52288 39552 S 2.3 1.3 0:04.02 PM2 v5.+
350 systemd+ 20 0 25532 4836 4352 S 0.3 0.1 0:01.41 systemd+
410 redis 20 0 154424 10108 4864 S 0.3 0.3 0:01.96 redis-s+
1557 ubuntu 20 0 11200 3840 3072 R 0.3 0.1 0:00.85 top
2078 root 20 0 0 0 0 I 0.3 0.0 0:00.02 kworker+
1 root 20 0 166320 7660 5612 S 0.0 0.2 0:04.71 systemd
2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
3 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 rcu_gp
4 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 rcu_par+
5 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 slub_fl+
6 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 netns
7 root 20 0 0 0 0 I 0.0 0.0 0:00.06 kworker+
8 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 kworker+
10 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 mm_perc+
11 root 20 0 0 0 0 I 0.0 0.0 0:00.00 rcu_tas+
12 root 20 0 0 0 0 I 0.0 0.0 0:00.00 rcu_tas+
ubuntu@ip-172-31-0-18: ~
top - 08:22:31 up 6 min, 1 user, load average: 0.47, 0.53, 0.27
Tasks: 110 total, 1 running, 109 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.2 us, 0.0 sy, 0.0 ni, 99.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 3904.5 total, 639.1 free, 3046.7 used, 218.6 buff/cache
MiB Swap: 2048.0 total, 1984.5 free, 63.5 used. 630.5 avail Mem
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
406 redis 20 0 154424 15236 6272 S 0.3 0.4 0:00.72 redis-s+
777 ubuntu 20 0 1025860 21236 14720 S 0.3 0.5 0:04.02 PM2 v5.+
1 root 20 0 166324 9960 6888 S 0.0 0.2 0:03.49 systemd
2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
3 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 rcu_gp
4 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 rcu_par+
5 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 slub_fl+
6 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 netns
7 root 20 0 0 0 0 I 0.0 0.0 0:00.03 kworker+
8 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 kworker+
10 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 mm_perc+
11 root 20 0 0 0 0 I 0.0 0.0 0:00.00 rcu_tas+
12 root 20 0 0 0 0 I 0.0 0.0 0:00.00 rcu_tas+
13 root 20 0 0 0 0 S 0.0 0.0 0:00.03 ksoftir+
14 root 20 0 0 0 0 I 0.0 0.0 0:00.21 rcu_sch+
15 root rt 0 0 0 0 S 0.0 0.0 0:00.00 migrati+
16 root -51 0 0 0 0 S 0.0 0.0 0:00.00 idle_in+