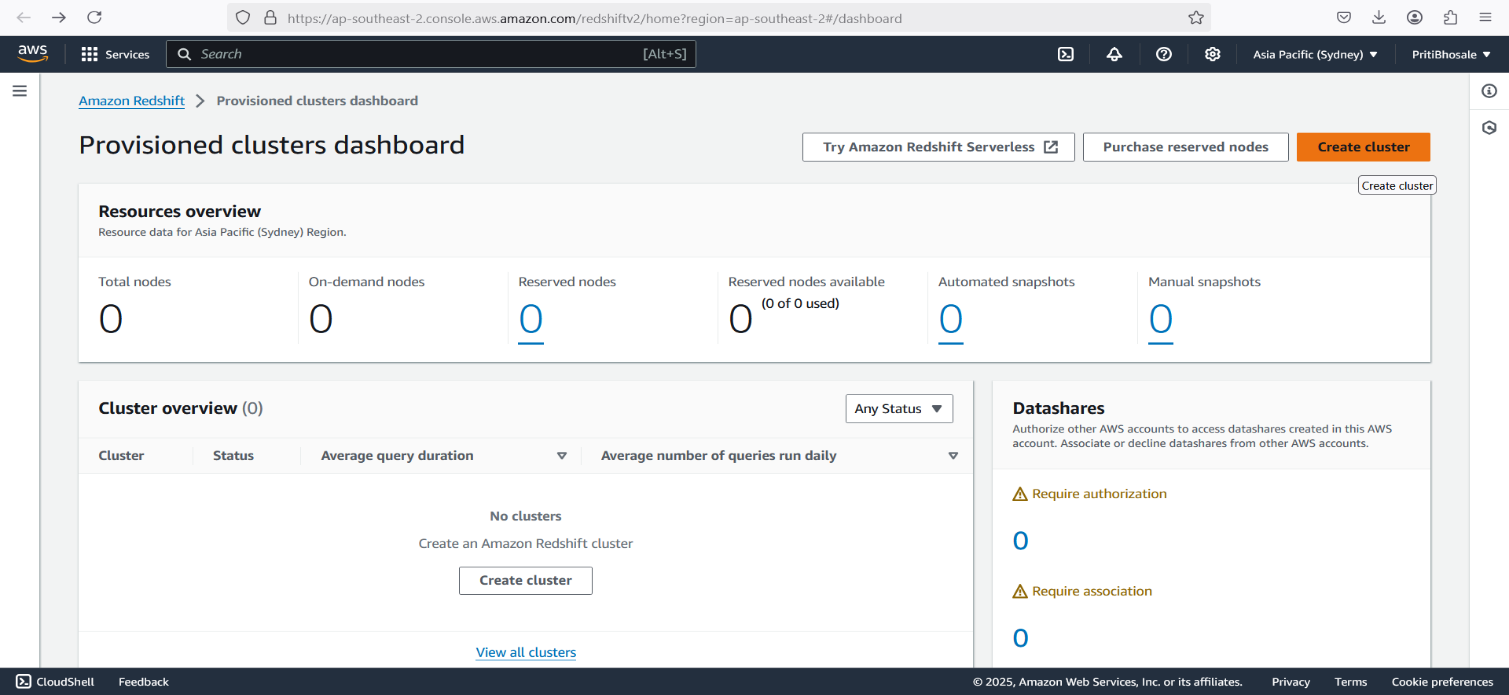
**Task 1: Create Own Database in Amazon Redshift and Fetch Data from SQL Editor**

Step 1: Create a Redshift Cluster

1. Sign in to AWS Management Console and navigate to Amazon Redshift.
2. Click Create Cluster and provide the following details:

* Cluster Identifier: my-redshift-cluster
* Node Type: dc2.large (for small workloads)
* Number of Nodes: 1 (single-node cluster for testing)
* Database Name: mydatabase
* Master Username: admin
* Master Password: yourpassword
* Enable Public Access (optional) and select a VPC.
* Click Create Cluster.

**Step 2: Connect to Redshift Using SQL Editor**

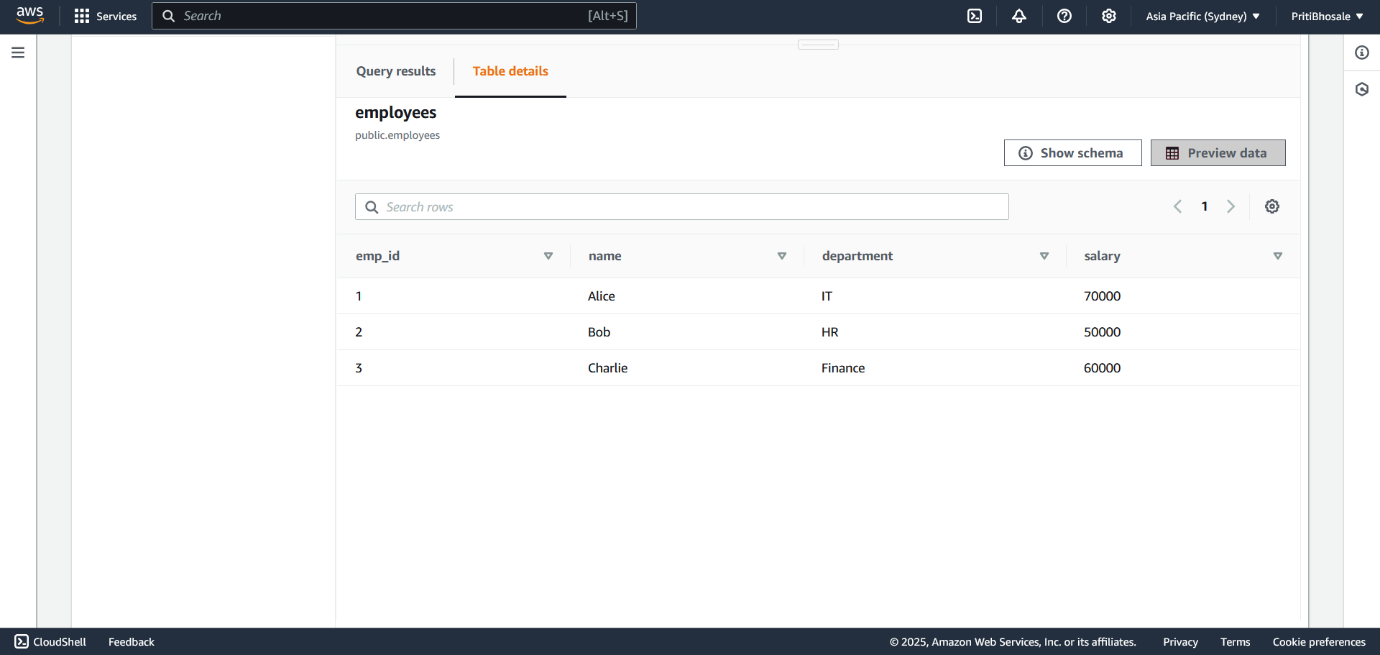
1. Navigate to Query Editor inside the Redshift console.
2. Select your database and connect using the credentials.
3. Run the following SQL commands:

A screenshot of a computer

AI-generated content may be incorrect.

**Conclusion**

You have successfully created a new database in **Amazon Redshift**, connected to it using **SQL Editor**, created a table, inserted data, and fetched records using SQL queries.



**Task 2: Deploy WordPress on AWS Using Elastic Beanstalk**

**Step 1: Prepare Your AWS Environment**

1. Log in to AWS Console

* Go to [AWS Management Console](https://aws.amazon.com/console/).
* Navigate to Elastic Beanstalk.
* Create an IAM Role for Elastic Beanstalk
* Go to IAM Service.
* Click on Roles > Create Role.
* Select AWS Service and choose Elastic Beanstalk.
* Attach policies:
  + - * AWSElasticBeanstalkFullAccess
      * AmazonRDSFullAccess (if using RDS for the database)
* Click Create Role.

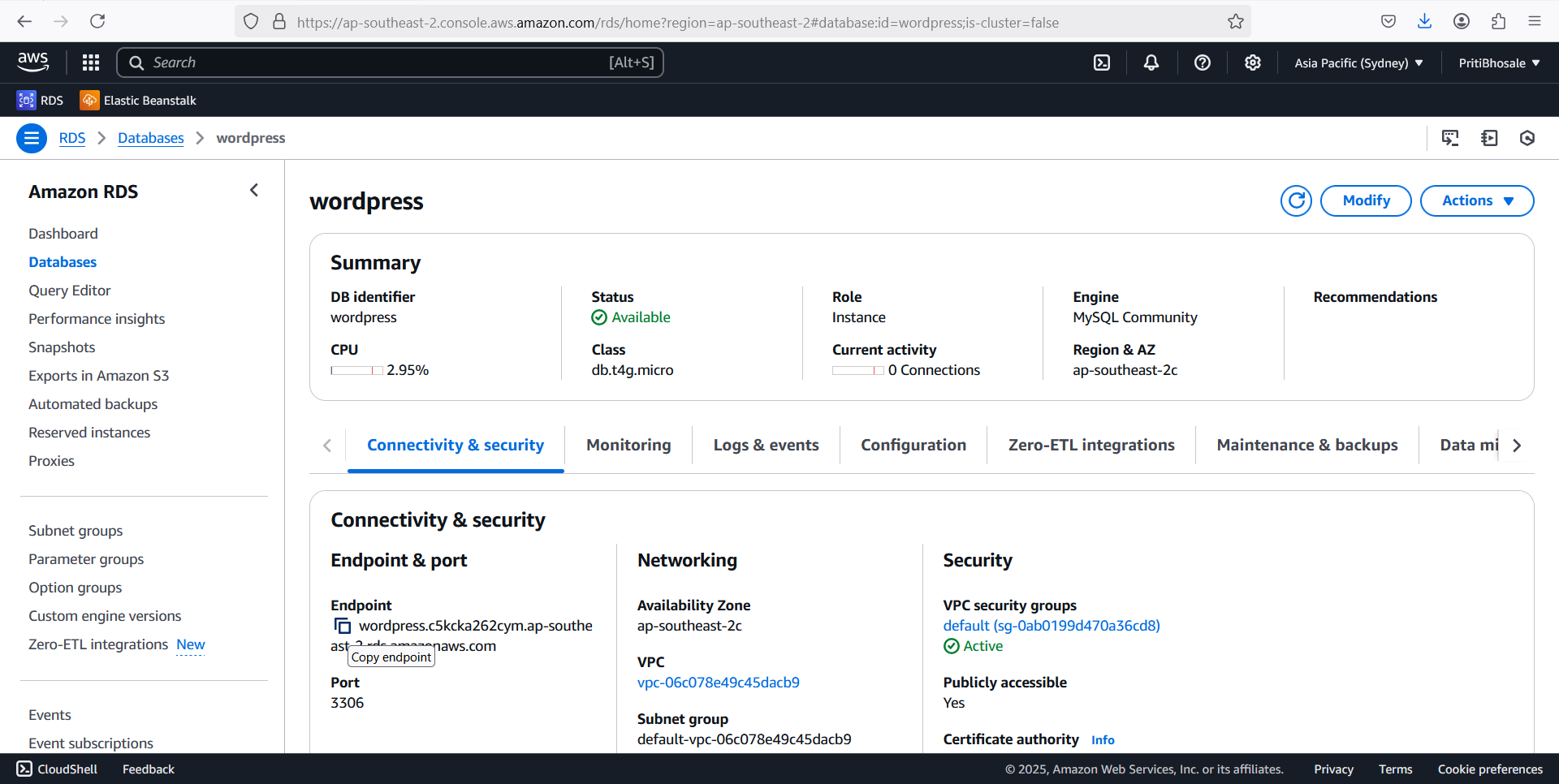
**Step 2: Create an RDS Database for WordPress**

1. Create a MySQL or MariaDB database in RDS.

A screenshot of a computer

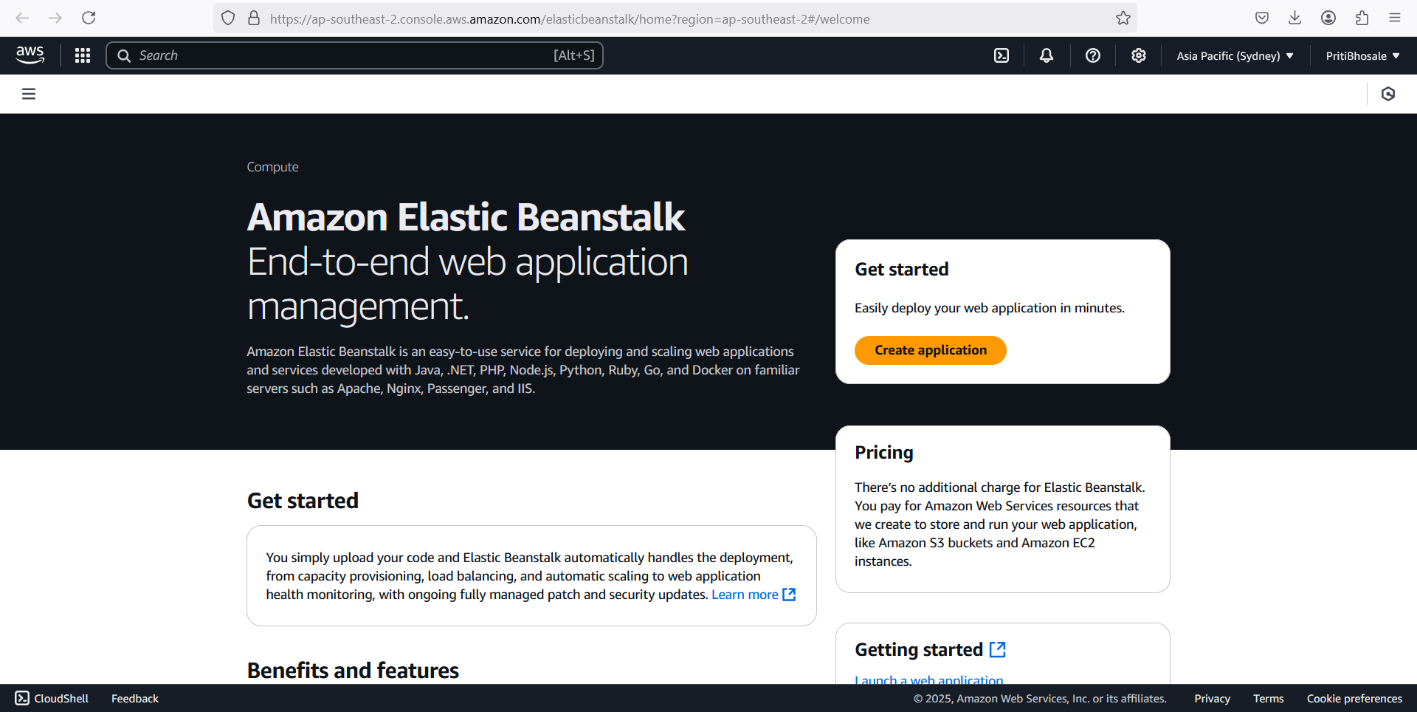
AI-generated content may be incorrect.

1. Add port no 3306 and 22 in security group of RDS
2. Configure the database and note the endpoint.



**Step 3: Create an Elastic Beanstalk Environment**

1. Navigate to AWS Elastic Beanstalk.



1. Click Create New Application, provide a name (wordpress-app).
2. Choose Platform: PHP.
3. Select **Create Environment** → Choose Web Server Environment.
4. Upload a pre-built WordPress ZIP file.

* Go to the official **WordPress website**: <https://wordpress.org/download/>
* Click on the **Download WordPress** button to get the latest .zip file.
* Extract and Prepare the Files
* You will see a folder named wordpress containing all the necessary files.
* Modify the **wp-config-sample.php** file (if needed) to include database credentials:

define('DB\_NAME', 'your-database-name');

define('DB\_USER', 'your-username');

define('DB\_PASSWORD', 'your-password');

define('DB\_HOST', 'your-rds-endpoint');

1. Rename the file to wp-config.php.
2. Compress the Files into a ZIP

* Select all extracted **WordPress** files (not the parent folder).
* Right-click and choose **Compress to ZIP**
* Ensure that wordpress.zip contains the **files directly** (not inside an extra folder).

**Step 5: Finalize and Access WordPress**

1. Check the Application Health

* In Elastic Beanstalk, ensure the environment status is Healthy.

1. Access WordPress

* Copy the Elastic Beanstalk URL from the console.
* Open it in a browser: <http://your-environment.elasticbeanstalk.com>.

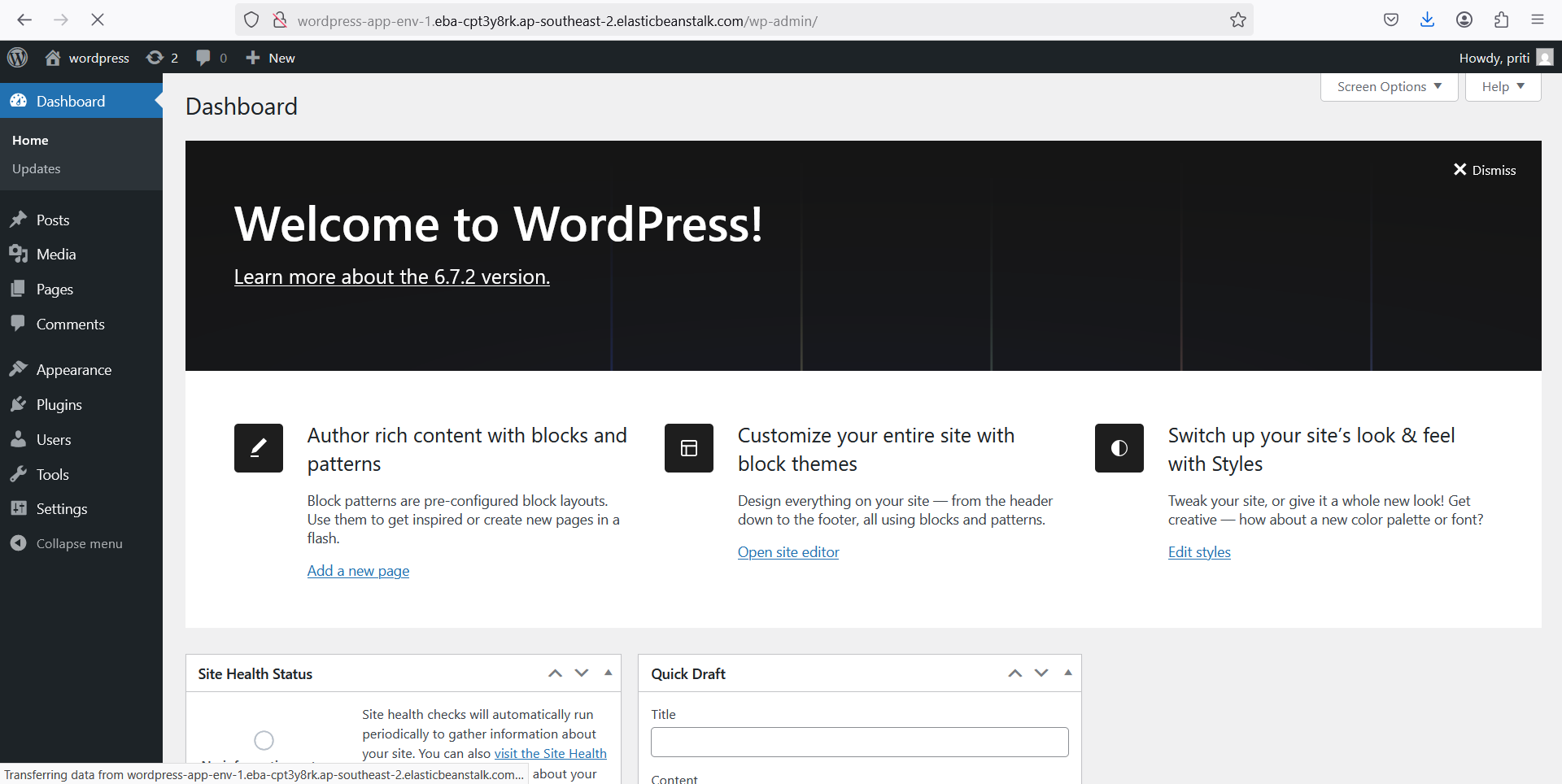
A screenshot of a computer

AI-generated content may be incorrect.

* Follow the WordPress installation steps.

**Conclusion**

You have successfully deployed WordPress on **AWS Elastic Beanstalk** with an optional **RDS database** for storage. You can now customize WordPress, install themes/plugins, and start your website!



**Task 3: Use Nginx as a Proxy Server**

**Step 1: Launch an EC2 Instance**

* 1. Go to AWS Console > EC2 > Instances → Click Launch Instance

A screenshot of a computer

AI-generated content may be incorrect.

2. Choose Amazon Linux 2 (or Ubuntu 22.04 for latest Node.js support)

3. Instance Type: Select t2.micro (Free Tier eligible)

4. Security Group:

* Allow SSH (port 22) → Your IP
* Allow HTTP (port 80) → Anywhere

5. Key Pair: Select or create a new key

6. Launch Instance and Connect via SSH

**Step 2: Install Node.js & Nginx on EC2**

🔹 For Amazon Linux 2

sudo yum update -y

curl -fsSL https://rpm.nodesource.com/setup\_18.x | sudo bash -

sudo yum install -y nodejs

🔹 For Ubuntu

sudo apt update -y

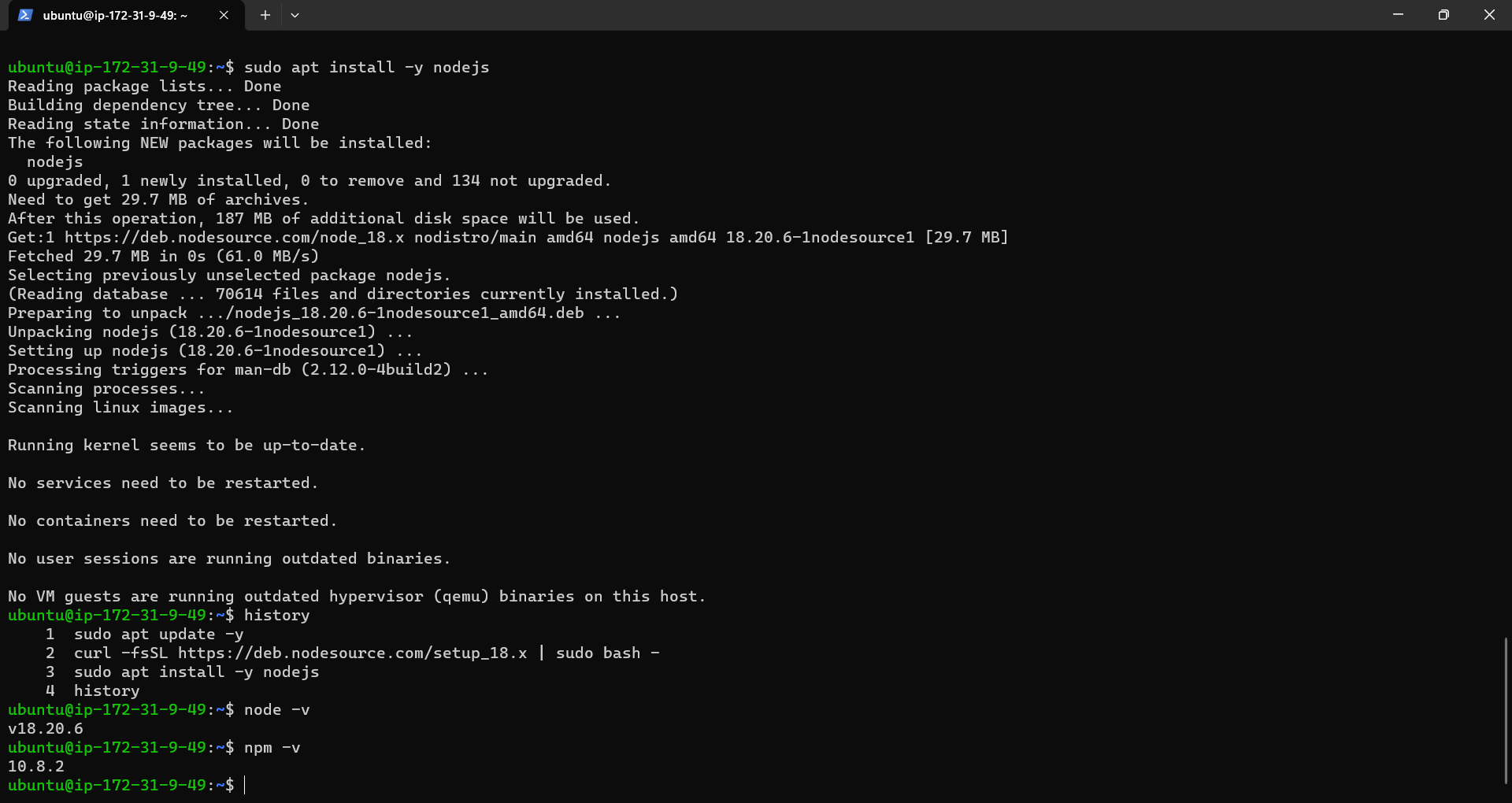
curl -fsSL https://deb.nodesource.com/setup\_18.x | sudo bash -

sudo apt install -y nodejs

🔹 Verify Installation

node -v

npm -v



**Step 3: Deploy a Sample Node.js App**

* + 1. Create a directory for your app

mkdir ~/node-app && cd ~/node-app

* + 1. Initialize Node.js App

npm init -y

* + 1. Install Express

npm install express

1. Create server.js

nano server.js

1. Add the following code:

const express = require('express');

const app = express();

const PORT = 5000;

app.get('/', (req, res) => {

res.send('Hello from Node.js running behind Nginx on AWS!');

});

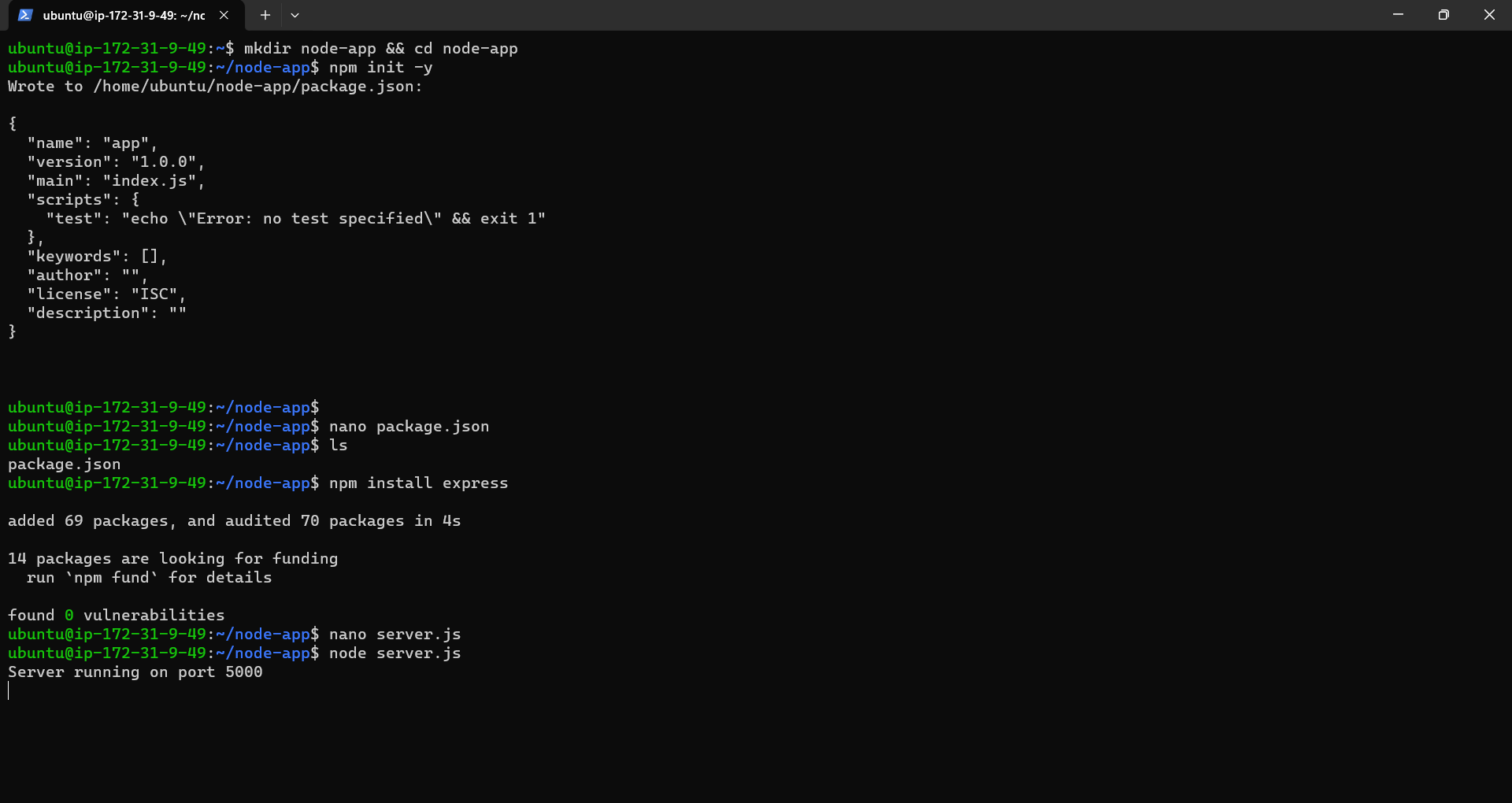
app.listen(PORT, () => {

console.log(`Server running on port ${PORT}`);

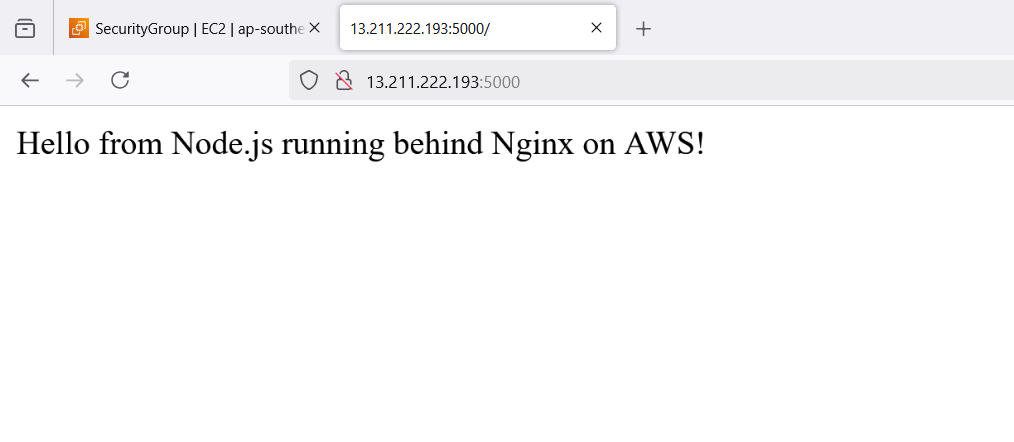
});

1. Save & Exit (Ctrl + X, then Y, then Enter)
2. Run the Node.js App

node server.js



* Visit http://EC2-PUBLIC-IP:5000 → You should see "Hello from Node.js..."



* If you want to run it in the background, install pm2:

npm install -g pm2

pm2 start server.js

pm2 save

pm2 startup

**Step 4: Configure Nginx as a Reverse Proxy**

* 1. Install Nginx

sudo yum install nginx -y # Amazon Linux

sudo apt install nginx -y # Ubuntu

* 1. Edit Nginx Configuration

sudo nano /etc/nginx/sites-available/default

* 1. Replace or add this inside the http {} block:

server {

listen 80;

server\_name \_; # Replace with your domain if available

location / {

proxy\_pass http://127.0.0.1:5000;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header X-Forwarded-Proto $scheme;

}

}

* 1. Save & Exit (Ctrl + X, then Y, then Enter)
  2. Restart Nginx

sudo systemctl restart nginx

sudo systemctl enable nginx

**Step 5: Allow Traffic in Security Groups**

* + 1. Go to AWS EC2 Dashboard > Security Groups
    2. Edit Inbound Rules and allow:

HTTP (80) → Anywhere (0.0.0.0/0)

Custom TCP (5000) → Your Private IP (if needed)

**Step 6: Test the Setup**

* + - 1. Open http://EC2-PUBLIC-IP in your browser

You should see:

**Hello from Node.js running behind Nginx on AWS!**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Conclusion**

In this task, we successfully set up **Nginx as a reverse proxy** for a **Node.js application** on an **AWS EC2 instance**. We:

* Deployed a **Node.js Express app** running on port **5000**
* Installed & configured **Nginx** to forward requests to Node.js
* Opened necessary **firewall & security group rules**
* Verified the setup by accessing **EC2’s public IP**

**Task 4: Short Note on S3 Features**

**Amazon S3 Features**

Amazon Simple Storage Service (S3) is a highly scalable, durable, and secure object storage service designed for various use cases such as backup, archiving, big data analytics, and hosting static websites.

**1. Storage Classe**

S3 Standard – High durability & availability for frequently accessed data.

S3 Intelligent-Tiering – Automatically moves data between cost-effective tiers.

S3 Standard-IA & One Zone-IA – Cost-effective for infrequent access.

S3 Glacier & Glacier Deep Archive – Low-cost archival storage.

**2. Security & Compliance**

Encryption – Supports server-side (SSE-S3, SSE-KMS, SSE-C) and client-side encryption.

IAM Policies & Bucket Policies – Fine-grained access control.

Block Public Access – Prevents unintended public exposure.

Object Lock – Enforces WORM (Write Once Read Many) compliance.

**3. Data Management & Performance**

Versioning – Maintains multiple versions of an object to prevent accidental deletions.

Lifecycle Policies – Automates data movement between storage classes.

Replication – Supports Cross-Region Replication (CRR) & Same-Region Replication (SSR).

Event Notifications – Triggers AWS Lambda, SNS, or SQS on object changes

**4. Cost Optimization & Monitoring**

S3 Storage Lens – Provides visibility into storage usage and activity.

Requester Pays – Shifts data transfer costs to requesters.

Access Logs & CloudTrail Integration – Tracks access and activity for security audits.

* 1. **Performance & Scalability**

High Durability (99.999999999% - 11 9’s) – Ensures data reliability.

Parallel Processing – Optimized for high throughput workloads.

Multi-Part Uploads – Speeds up large object uploads.

**6. Static Website Hosting & Edge Integration**

Static Website Hosting – Hosts websites with public access.

S3 Transfer Acceleration – Enhances upload speed using AWS Edge locations.

Integration with CloudFront – Distributes content globally with low latency.

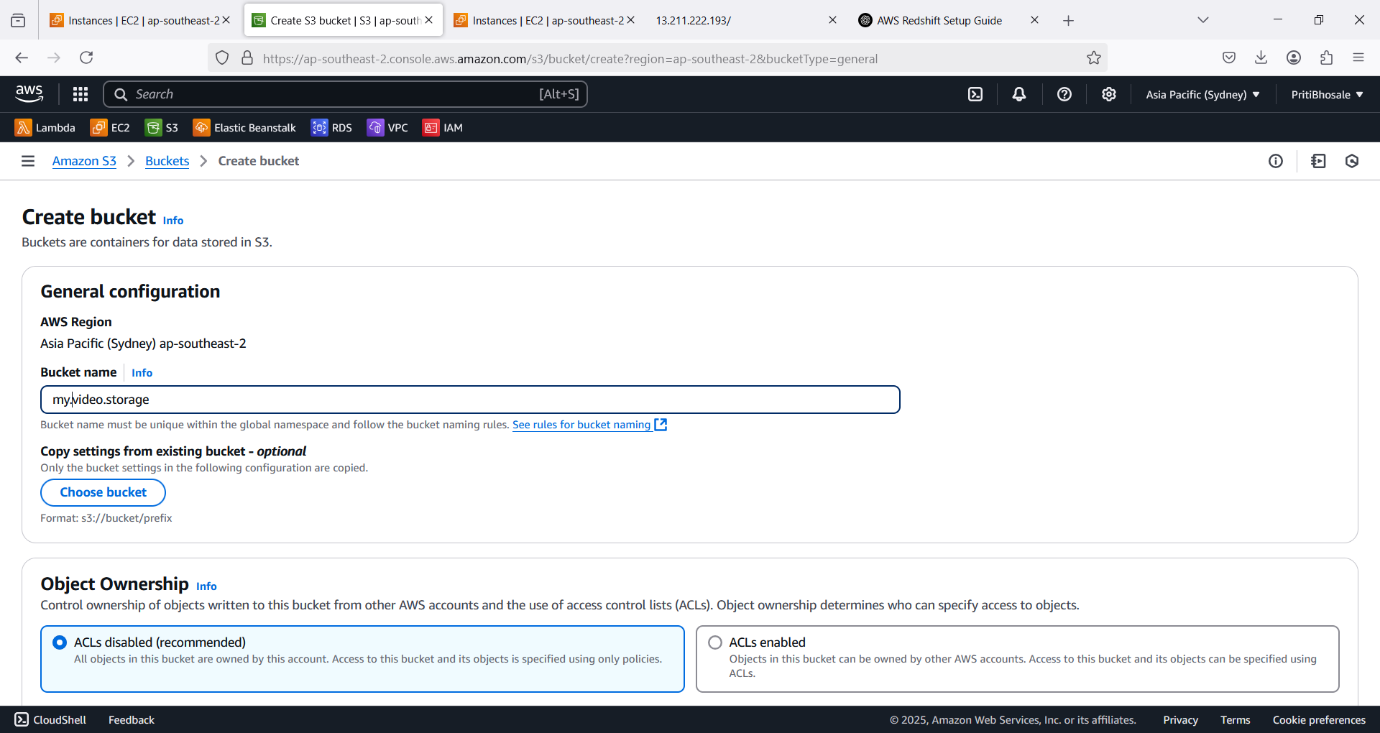
**Conclusion**

Amazon S3 provides secure, scalable, and cost-efficient object storage, making it an essential service for modern cloud architectures.

**Task 4: Accessing Multiple Videos in S3 Using a Single URL**

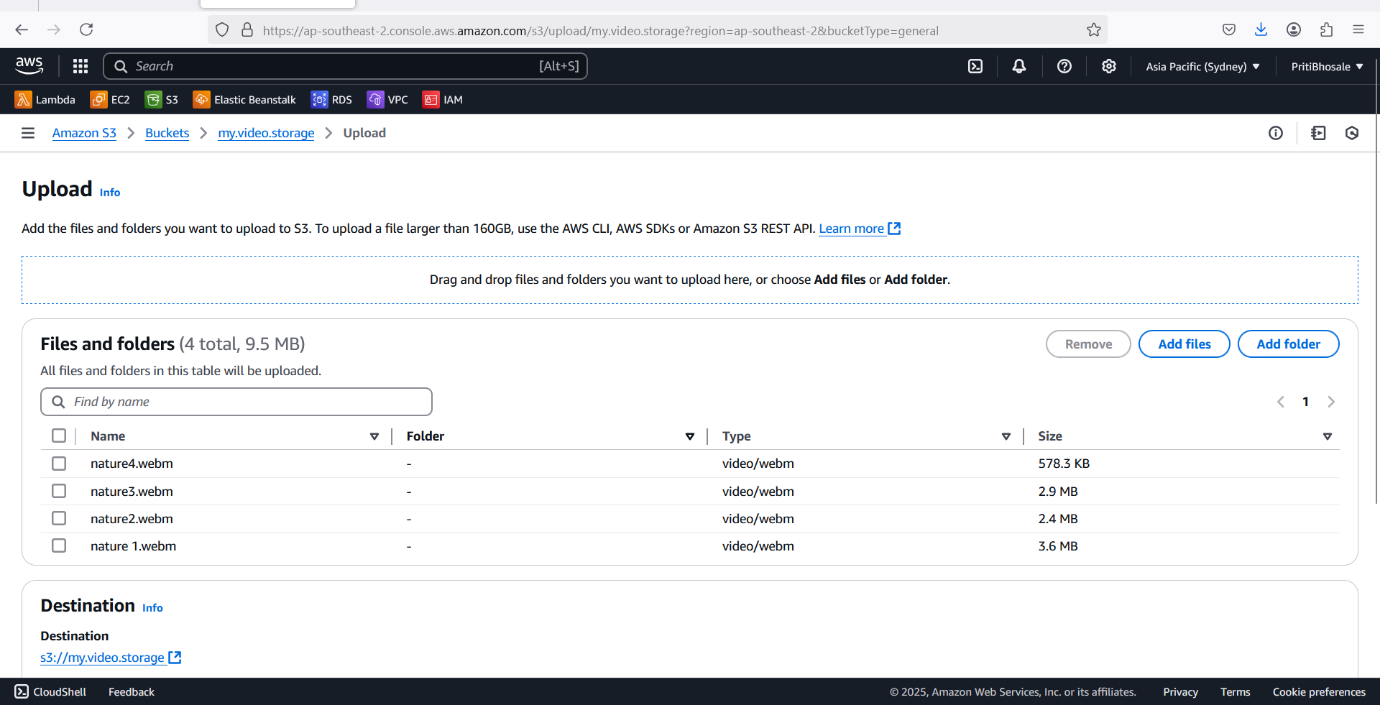
**Step 1: Create an S3 Bucket**

1. Go to AWS Console > S3
2. Click Create bucket
3. Enter a unique bucket name (e.g., my.video.storage)
4. Select your preferred AWS region
5. Uncheck "Block all public access" (if videos are public)
6. Click Create bucket



**Step 2: Upload Multiple Videos to S3**

1. Open the S3 bucket you created
2. Click Upload → Add files
3. Select multiple video files (.mp4, .avi, .mkv, etc.)
4. Click Upload



**Step 3: Configure S3 Permissions (Optional: Public Access)**

If videos should be publicly accessible:

Open S3 bucket → Go to Permissions

Click Bucket Policy

Add this policy (Replace my-video-storage with your bucket name):

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Principal": "\*",

"Action": "s3:GetObject",

"Resource": "arn:aws:s3:::my.video.storage/\*"

}

]

}

Click Save

A screenshot of a computer

AI-generated content may be incorrect.

Now, each video has a public URL:

https://s3.ap-southeast-2.amazonaws.com/my.video.storage/nature2.webm

**Step 4: Create a CloudFront Distribution (Single URL for All Videos)**

1. Go to AWS Console > CloudFront
2. Click Create Distribution
3. Under Origin, choose:

* Origin Domain → Select your S3 bucket
* Origin Access → Choose Origin Access Control (OAC)
* Click Create Control Setting → Allow CloudFront to access the bucket

1. Under Default Behaviour, choose:

* Viewer Protocol Policy → Redirect HTTP to HTTPS
* Cache Policy → Select CachingOptimized

1. Click Create Distribution

A screenshot of a computer

AI-generated content may be incorrect.

* Now, CloudFront provides a single URL like:

<https://dqazx2p8x9j4d.cloudfront.net/nature2.webm>

<https://dqazx2p8x9j4d.cloudfront.net/nature3.webm>

**Step 5: Use the Single URL for Accessing Videos**

Now, all your videos can be accessed using CloudFront with a **single base URL**, like:

[https://d123abc.cloudfront.net/{video-name}.mp4](https://d123abc.cloudfront.net/%7bvideo-name%7d.mp4)

A person in an orange raincoat walking on a grassy hill

AI-generated content may be incorrect.

**Conclusion**

In this task, we uploaded multiple videos to **Amazon S3** and used **AWS CloudFront** to create a **single base URL** for accessing them. This setup:

* Provides **faster video streaming** with CloudFront CDN
* Enables **a consistent URL format** for all videos
* Improves **security & access control** with S3 and CloudFront

This approach is ideal for **web apps, streaming services, and content delivery**.