Table of Contents

[**Analysis** 3](#_Toc176766751)

[**Identify Bottlenecks** 3](#_Toc176766752)

[**Monitoring Setup** 5](#_Toc176766753)

[**Implementing CloudWatch to continuously track key performance metrics (CPU usage, memory usage, network latency)** 5](#_Toc176766754)

[**Logs these metrics in a structured format (e.g., JSON, CSV).** 10](#_Toc176766755)

[**Triggers an alert (e.g., email, SMS) when certain thresholds are exceeded.** 14](#_Toc176766756)

[**Optimization** 16](#_Toc176766757)

[**Resource Scaling** 16](#_Toc176766758)

[**Optional Enhancements** 20](#_Toc176766759)

[**Log Rotation** 20](#_Toc176766760)

[**System Simulation** 22](#_Toc176766761)

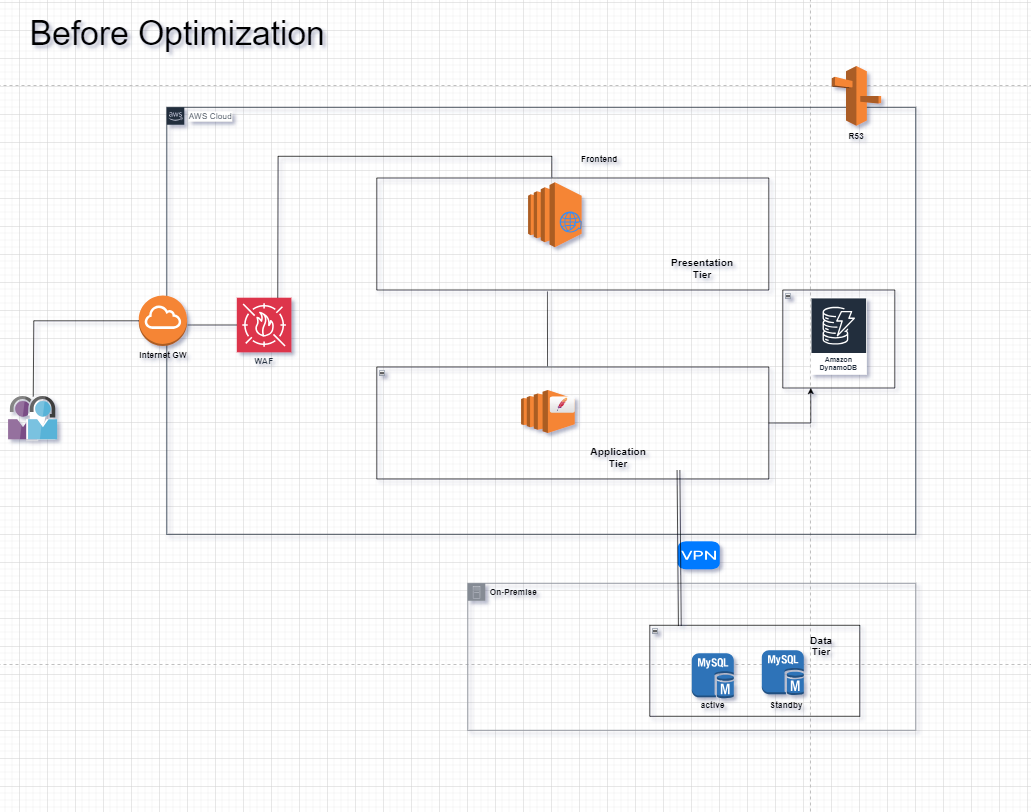
[Write a script to simulate peak load conditions and measure the system’s performance, allowing you to test the effectiveness of your optimizations. 22](#_Toc176766762)

[**Automated Deployment** 23](#_Toc176766763)

**Task 1**

Assumptions:-

Assuming e-commerce application with three tiers:



Presentation Tier: A web server (Apache HTTP Server) running on cloud instance (AWS EC2), which does the heavy lifting of taking care of user requests and showing the website.

Application Tier: A cloud instance (AWS EC2) running a Java-based application server (Apache Tomcat), which processing business logic and communicating with database.

Data Tier: A cloud-based relational database (AWS RDS) running MySQL, responsible for storing and retrieving data.

# **Analysis**

## **Identify Bottlenecks**

**Analyze the potential bottlenecks in a typical multi-tier application (e.g., CPU, memory, network latency, database contention) and explain how they could affect performance.**   
  
A multi-tier application can become a performance bottleneck at any layer, due to the CPU memory, network latency. disco I/O or contention issues.  
Potential bottlenecks can occur in any tier (Presentation, Application or Data) within a multi-tier application architecture and it would affect the overall system performance particularly during peak load timings.

**CPU Bottlenecks**

Handling a large number of concurrent user requests, performing resource-intensive tasks, such as image processing or video transcoding, inefficient server-side rendering or caching mechanisms or serving dynamic content may cause this issue.

**Impact on Performance:** High CPU usage on the web server can lead to slow page loads, timeouts, and a poor user experience. When the server struggles to handle incoming user requests, it can result in delayed responses, reduced throughput, and increased latency. This is especially critical in e-commerce applications where user experience directly affects revenue.  
Also the application server tier can slow down the processing of business logic and lead to request timeouts. This affects the overall response time of the application and could cause the server to crash if the CPU is consistently maxed out, impacting application reliability and availability.  
Database tier can lead to slow query performance, increased latency in data retrieval, and overall degradation of the application's responsiveness. In severe cases, this can lead to database server crashes, affecting the availability of data and causing downtime.  
  
CPU bottlenecks in any tier of a multi-tier application can cause significant performance issues, such as slow response times, timeouts, and application crashes. Identifying the root cause of high CPU usage—whether it is due to handling numerous concurrent requests, running resource-intensive operations, or inefficient processing—is essential for optimizing system performance and ensuring reliability.

**Memory Bottlenecks**

Inefficient caching or session management, Leaks or excessive object creation, Inadequate heap size or inefficient memory allocation can lead to Memory bottlenecks.  
Eg:-  
Insufficient memory (RAM) on the web server could cause slow responses or crashes. When memory is exhausted, the server may start swapping, which can significantly degrade performance. If the Apache server handles too many concurrent connections without adequate memory, it can lead to high response times or failures. Also if the database server does not have enough memory to cache frequently accessed data or to handle the working set, it could result in frequent disk reads, causing latency. Insufficient buffer size and cache could lead to excessive swapping, impacting query performance.

**Impact on Performance:**

Insufficient memory can lead to

Frequent page reloads or slow response times.  
Inadequate caching or session management.  
Frequent disk reads, causing latency.  
Due to the excessive swapping, impacting query performance.

**Network Latency Bottlenecks**

Distance between cloud instances or data centers / geographical distance between users and the server, high traffic, Insufficient network bandwidth (bandwidth limitations) or packet loss, slow third-party APIs or services, High latency in database queries or storage access can lead to Network Latency Bottlenecks.

**Impact on Performance:**

Between the web server and clients could result in slow page load times  
between the application server and other tiers (e.g., database or web server) could cause delays in processing requests.  
Between the application server and the database can cause delays in query execution and data retrieval.

**Database Contention Bottlenecks**

High concurrency or locking issues, Slow queries or inadequate query optimization (Complex or unoptimized SQL queries), Inefficient database schema or indexing (leading to full table scans) , Insufficient database connection pooling (High transaction rates) or caching can leads to a Database contention bottlenecks.

***Impact on Performance:***

Slow query performance  
increased latency in data retrieval and overall degradation of the application's responsiveness.  
Lead to database server crashes, affecting the availability of data and causing downtime

**Storage Bottlenecks**

Insufficient storage capacity or high disk usage, Inefficient data retrieval or caching, Slow disk I/O or inadequate storage configuration can leads to a Storage bottlenecks.

**Impact on Performance:**  
- Insufficient storage capacity or high disk usage can cause slow query response times, leading to delayed page loads and frustrated users.  
- Slow disk I/O can increase the time it takes to retrieve data, resulting in slower application performance.  
- Due to the additional latency in the system, affecting the overall responsiveness of the application. High latency can lead to slower page loads, increased bounce rates, and a poor user experience.  
- reduce the throughput of the application, limiting the number of requests that can be processed simultaneously, **it** can result in slower performance, even during non-peak usage times.  
- increased error rates, such as timeouts or failed queries, which can cause application errors and affect user experience  
- resource contention, where multiple components of the application compete for limited storage resources. Components may need to wait for resources to become available, leading to slower response times.

## **Monitoring Setup**

### **Implementing CloudWatch to continuously track key performance metrics (CPU usage, memory usage, network latency)**

Setup custom CloudWatch custom metrics in AWS EC2

Steps: -

1. Install the CloudWatch Agent in EC2 instance.

*sudo yum install amazon-cloudwatch-agent*

2. Configure the CloudWatch Agent.

Create a configuration file manually at /opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.json

*amazon-cloudwatch-agent.json*

{

"metrics": {

"namespace": "CustomNamespace",

"metrics\_collected": {

"cpu": {

"measurement": [

"cpu\_usage\_idle",

"cpu\_usage\_iowait",

"cpu\_usage\_user",

"cpu\_usage\_system"

],

"metrics\_collection\_interval": 60,

"totalcpu": true

},

"mem": {

"measurement": [

"mem\_used\_percent",

"mem\_available\_percent"

],

"metrics\_collection\_interval": 60

},

"netstat": {

"measurement": [

"tcp\_established",

"tcp\_time\_wait",

"tcp\_close\_wait"

],

"metrics\_collection\_interval": 60

}

}

}

}

Novo energy

Explanation of the Configuration:  
  
Collects CPU usage metrics :   
cpu\_usage\_idle: Percentage of time the CPU is idle.

cpu\_usage\_iowait: Percentage of time the CPU is waiting for I/O operations to complete.

cpu\_usage\_user: Percentage of CPU time spent on user-level processes.

cpu\_usage\_system: Percentage of CPU time spent on system-level processes.

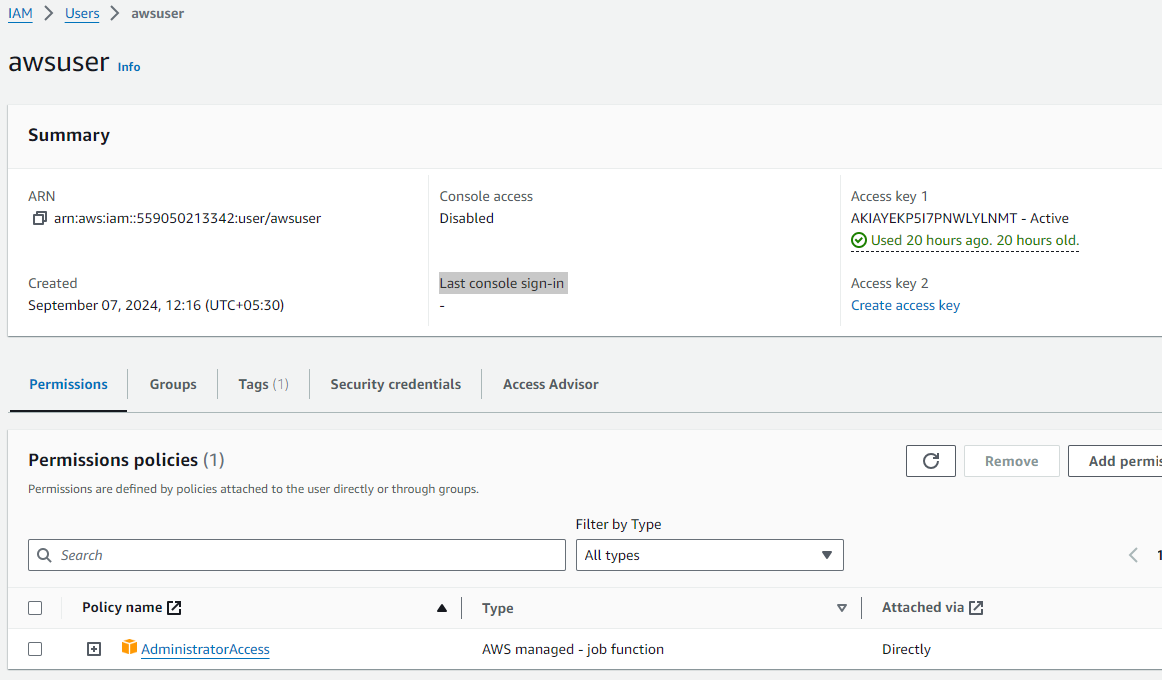
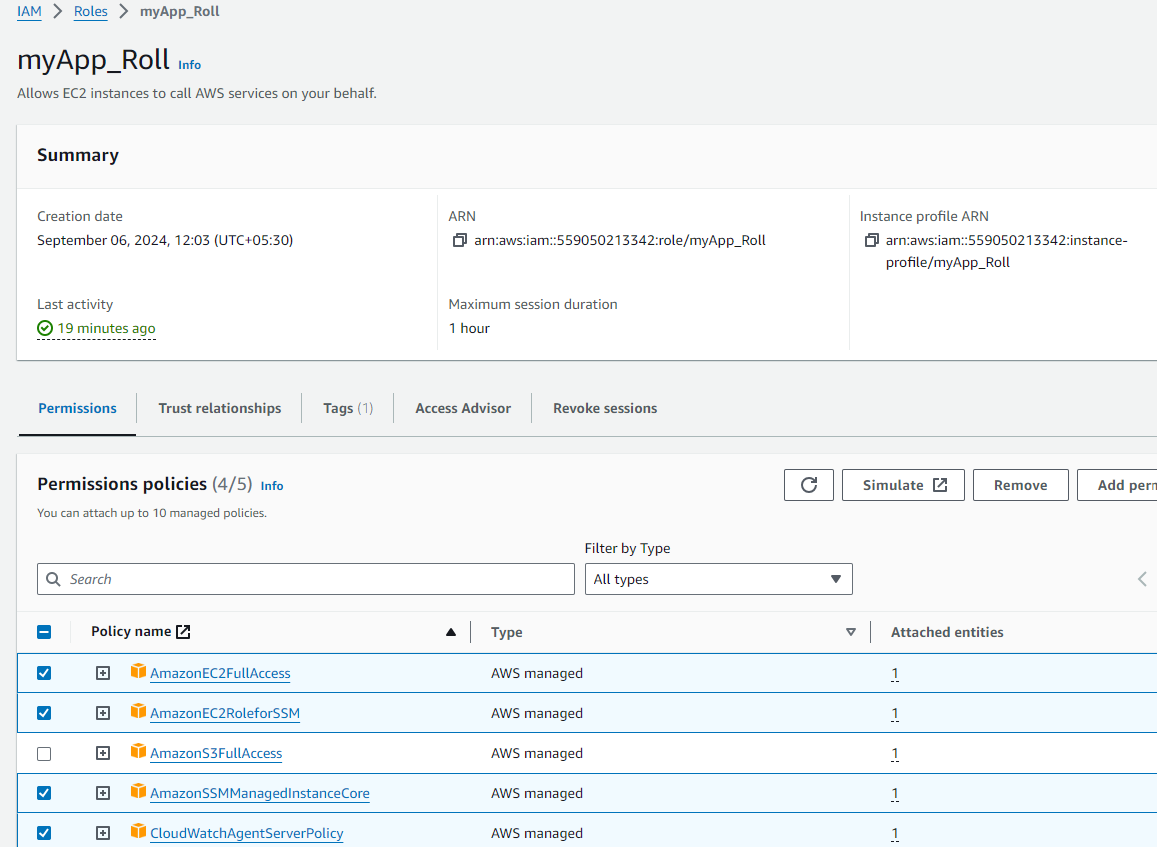
Collects memory usage metrics :  
mem\_used\_percent: Percentage of used memory.

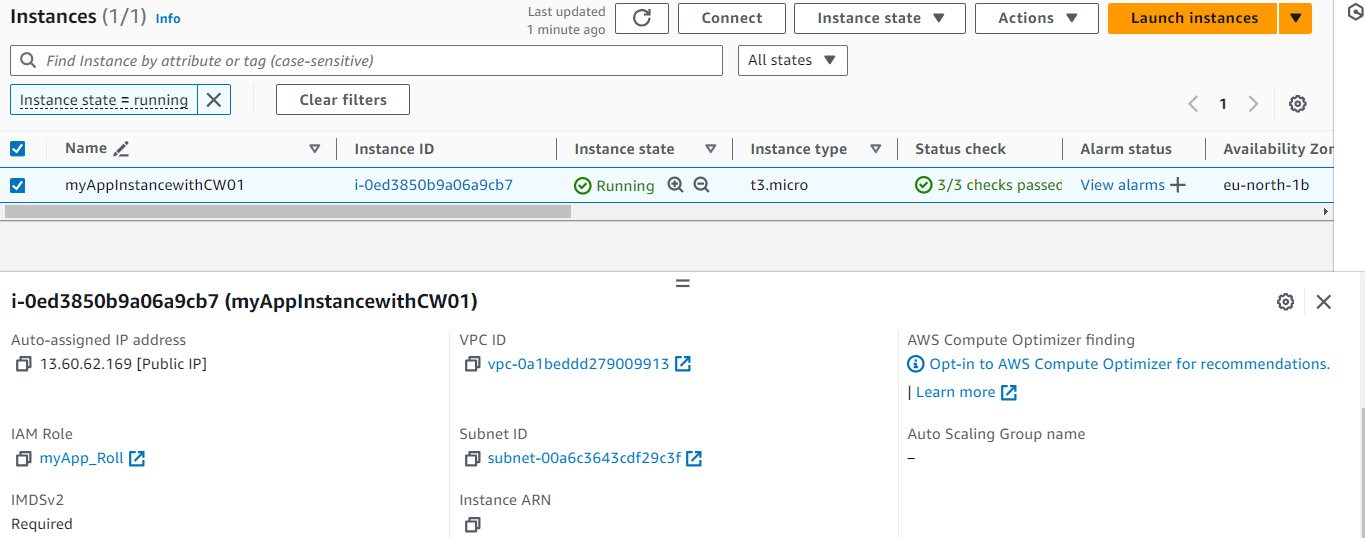
mem\_available\_percent: Percentage of available memory.  
  
Collects network-related TCP metrics :  
tcp\_established: Number of TCP connections that are currently established.

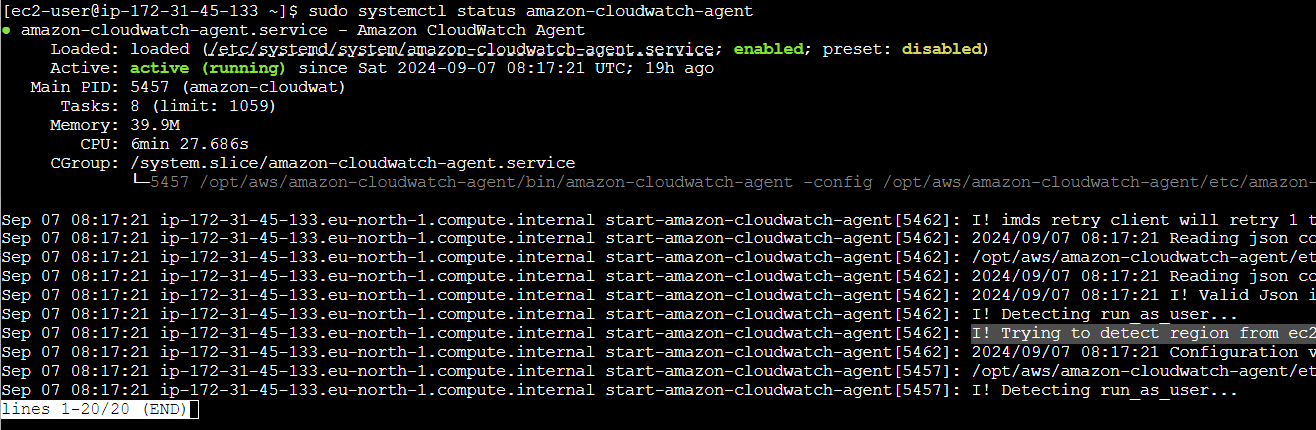
tcp\_time\_wait: Number of TCP connections that are in the TIME-WAIT state.

tcp\_close\_wait: Number of TCP connections that are in the CLOSE-WAIT state.

3. In this case I have created IAM, user (awsuser) and Role with appropriate policies (attached to the instance).  
  
AmazonEC2FullAccess  
AmazonEC2RoleforSSM  
AmazonSSMManagedInstanceCore  
CloudWatchAgentServerPolicy



4. Restart the CloudWatch Agent to apply/reload the new configuration:  
  
# sudo systemctl status amazon-cloudwatch-agent  
# sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a fetch-config -m ec2 -c file:/opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.json -s  
  


**5. Verify the Metrics**

- Navigate to the CloudWatch console in the AWS Management Console.  
- Check the custom namespace (CustomNamespace) to verify that the metrics for CPU usage, memory **usage, and network latency are being collected.**

**Note:   
This configuration will ensure that only the desired metrics are collected and displayed in Amazon CloudWatch under custom namespace.**

### **Logs these metrics in a structured format (e.g., JSON, CSV).**

1. Update the “amazon-cloudwatch-agent.json” file in /opt/aws/amazon-cloudwatch-agent/etc/ with the following content

{

"metrics": {

"namespace": "CustomNamespace",

"metrics\_collected": {

"cpu": {

"measurement": [

"cpu\_usage\_idle",

"cpu\_usage\_iowait",

"cpu\_usage\_user",

"cpu\_usage\_system"

],

"metrics\_collection\_interval": 60,

"totalcpu": true

},

"mem": {

"measurement": [

"mem\_used\_percent",

"mem\_available\_percent"

],

"metrics\_collection\_interval": 60

},

"netstat": {

"measurement": [

"tcp\_established",

"tcp\_time\_wait",

"tcp\_close\_wait"

],

"metrics\_collection\_interval": 60

}

}

},

"logs": {

"logs\_collected": {

"files": {

"collect\_list": [

{

"file\_path": "/var/log/cpu\_metrics.log",

"log\_group\_name": "CustomNamespace/CPU",

"log\_stream\_name": "{instance\_id}",

"timestamp\_format": "%Y-%m-%d %H:%M:%S",

"timezone": "UTC"

},

{

"file\_path": "/var/log/memory\_metrics.log",

"log\_group\_name": "CustomNamespace/Memory",

"log\_stream\_name": "{instance\_id}",

"timestamp\_format": "%Y-%m-%d %H:%M:%S",

"timezone": "UTC"

},

{

"file\_path": "/var/log/network\_metrics.log",

"log\_group\_name": "CustomNamespace/Network",

"log\_stream\_name": "{instance\_id}",

"timestamp\_format": "%Y-%m-%d %H:%M:%S",

"timezone": "UTC"

}

]

}

},

"force\_flush\_interval": 15

}

}

Explanation of the Configuration:

Logs Collection:

* collect\_list includes three log files:  
    
  CPU Metrics Logs: Collected from /var/log/cpu\_metrics.log.

Memory Metrics Logs: Collected from /var/log/memory\_metrics.log.

Network Metrics Logs: Collected from /var/log/network\_metrics.log.

- Each log entry will be sent to a CloudWatch Logs group under the corresponding log group names (CustomNamespace/CPU, CustomNamespace/Memory, CustomNamespace/Network).

* Logs will be timestamped using the format %Y-%m-%d %H:%M:%S in UTC

1. Install Required Tools

sudo yum install -y sysstat net-tools

sudo yum install cronie  
sudo systemctl start crond

sudo systemctl enable crond

sysstat :- mpstat, vmstat, iostat, etc., for CPU and memory usage.  
net-tools :-For network-related metrics.  
cronie :- Install cron package to setup cronjobs

1. Create log files where the metrics will be logged:

sudo touch /var/log/cpu\_metrics.log

sudo touch /var/log/memory\_metrics.log

sudo touch /var/log/network\_metrics.log

# Set permissions so that they are writable by the 'cwagent' user or the user running the cron jobs

sudo chown cwagent:cwagent /var/log/cpu\_metrics.log

sudo chown cwagent:cwagent /var/log/memory\_metrics.log

sudo chown cwagent:cwagent /var/log/network\_metrics.log

sudo chmod 664 /var/log/cpu\_metrics.log /var/log/memory\_metrics.log /var/log/network\_metrics.log

1. Create three separate scripts for collecting CPU, memory, and network metrics. Each script will log the data in JSON format.

/opt/metric-scripts/cpu\_metrics.sh

#!/bin/bash

# Collect metrics

TIMESTAMP=$(date +'%Y-%m-%d %H:%M:%S')

CPU\_USAGE\_USER=$(mpstat 1 1 | awk '/Average:/ {print 100 - $12}')

CPU\_USAGE\_SYSTEM=$(mpstat 1 1 | awk '/Average:/ {print $4}')

CPU\_USAGE\_IDLE=$(mpstat 1 1 | awk '/Average:/ {print $12}')

CPU\_USAGE\_IOWAIT=$(mpstat 1 1 | awk '/Average:/ {print $6}')

# Format and log metrics in JSON

echo "{\"timestamp\": \"$TIMESTAMP\", \"cpu\_usage\_user\": $CPU\_USAGE\_USER, \"cpu\_usage\_system\": $CPU\_USAGE\_SYSTEM, \"cpu\_usage\_idle\": $CPU\_USAGE\_IDLE, \"cpu\_usage\_iowait\": $CPU\_USAGE\_IOWAIT}" >> /var/log/cpu\_metrics.log

/opt/metric-scripts/memory\_metrics.sh

#!/bin/bash

# Collect metrics

TIMESTAMP=$(date +'%Y-%m-%d %H:%M:%S')

MEM\_USED\_PERCENT=$(free | awk '/Mem:/ {printf "%.2f", $3/$2 \* 100.0}')

MEM\_AVAILABLE\_PERCENT=$(free | awk '/Mem:/ {printf "%.2f", $7/$2 \* 100.0}')

# Format and log metrics in JSON

echo "{\"timestamp\": \"$TIMESTAMP\", \"mem\_used\_percent\": $MEM\_USED\_PERCENT, \"mem\_available\_percent\": $MEM\_AVAILABLE\_PERCENT}" >> /var/log/memory\_metrics.log

/opt/metric-scripts/network\_metrics.sh

#!/bin/bash

# Collect metrics

TIMESTAMP=$(date +'%Y-%m-%d %H:%M:%S')

TCP\_ESTABLISHED=$(netstat -an | grep ESTABLISHED | wc -l)

TCP\_TIME\_WAIT=$(netstat -an | grep TIME\_WAIT | wc -l)

TCP\_CLOSE\_WAIT=$(netstat -an | grep CLOSE\_WAIT | wc -l)

# Format and log metrics in JSON

echo "{\"timestamp\": \"$TIMESTAMP\", \"tcp\_established\": $TCP\_ESTABLISHED, \"tcp\_time\_wait\": $TCP\_TIME\_WAIT, \"tcp\_close\_wait\": $TCP\_CLOSE\_WAIT}" >> /var/log/network\_metrics.log

Make the script executable:  
  
chmod +x /opt/metric-scripts/cpu\_metrics.sh  
chmod +x /opt/metric-scripts/memory\_metrics.sh  
chmod +x /opt/metric-scripts/network\_metrics.sh

1. To run a script every 5 minutes, schedule cron (with more detailed debugging information)  
     
   sudo crontab –e

\*/5 \* \* \* \* /opt/metric-scripts/cpu\_metrics.sh >> /var/log/cron\_debug.log 2>&1

\*/5 \* \* \* \* /opt/metric-scripts/memory\_metrics.sh >> /var/log/cron\_debug.log 2>&1

\*/5 \* \* \* \* /opt/metric-scripts/network\_metrics.sh >> /var/log/cron\_debug.log 2>&1

restart cron

# sudo systemctl restart crond

1. Verify and Test Scripts

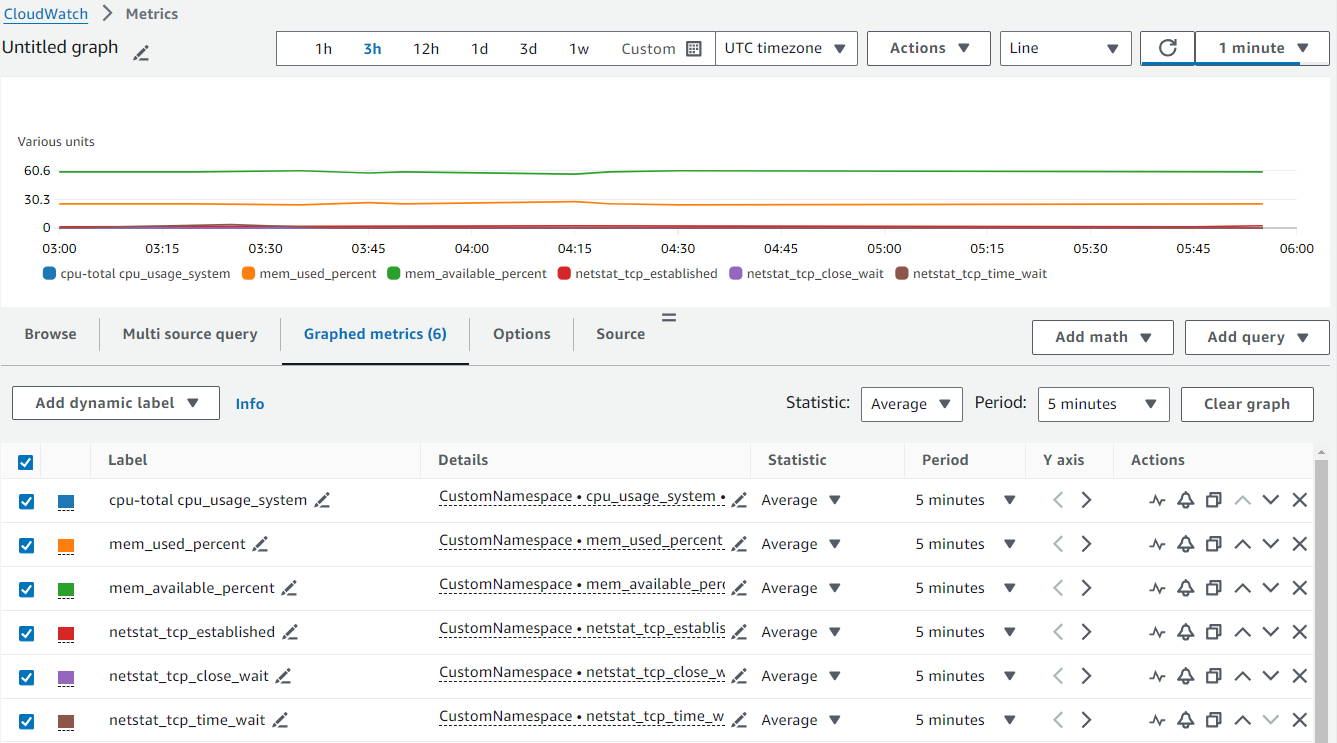
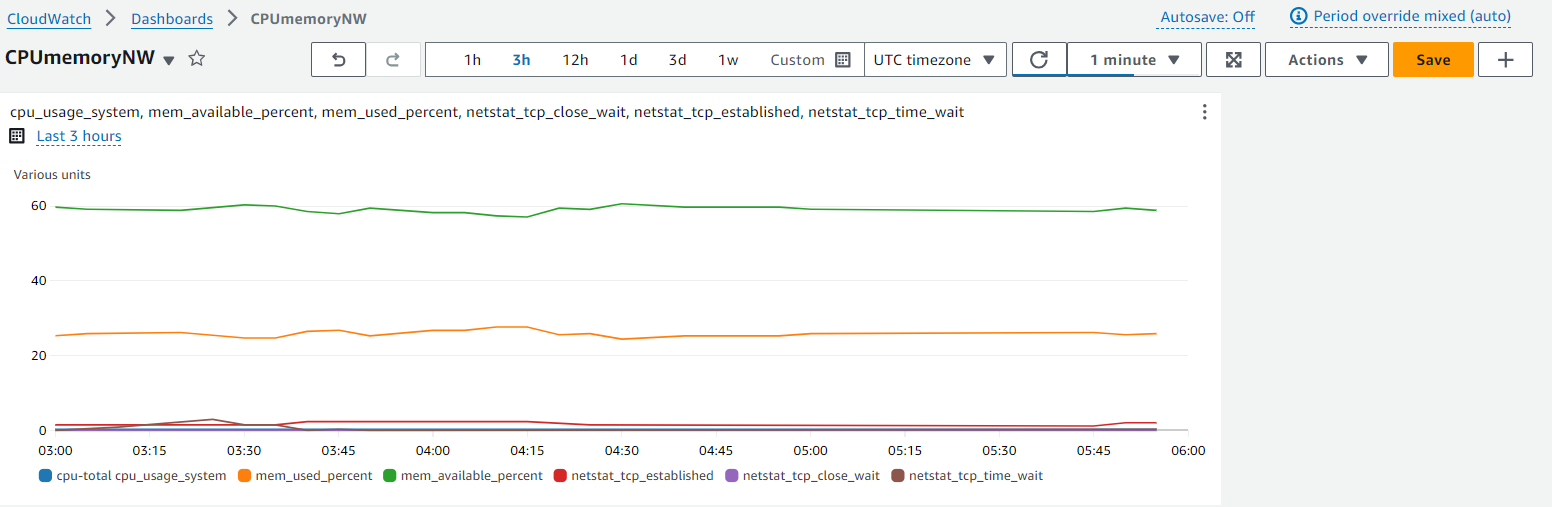
sudo bash /opt/metric-scripts/cpu\_metrics.sh

sudo bash /opt/metric-scripts/memory\_metrics.sh

sudo bash /opt/metric-scripts/network\_metrics.sh  
  
cat /var/log/cpu\_metrics.log

cat /var/log/memory\_metrics.log

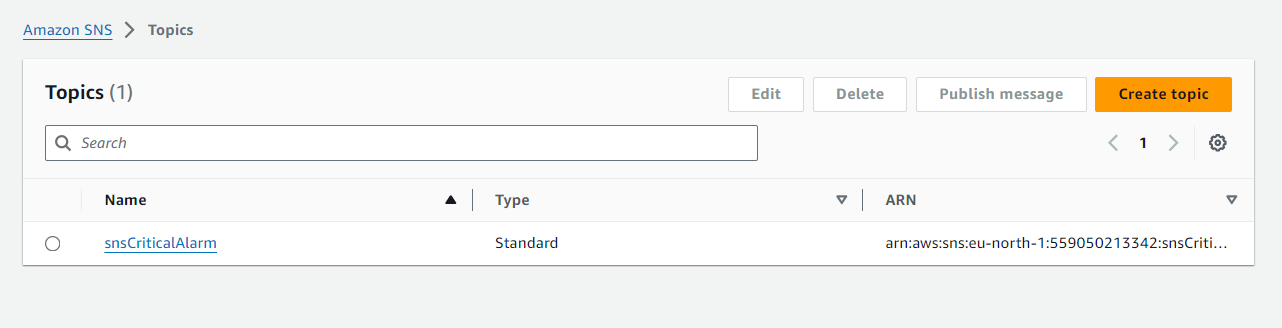
cat /var/log/network\_metrics.log

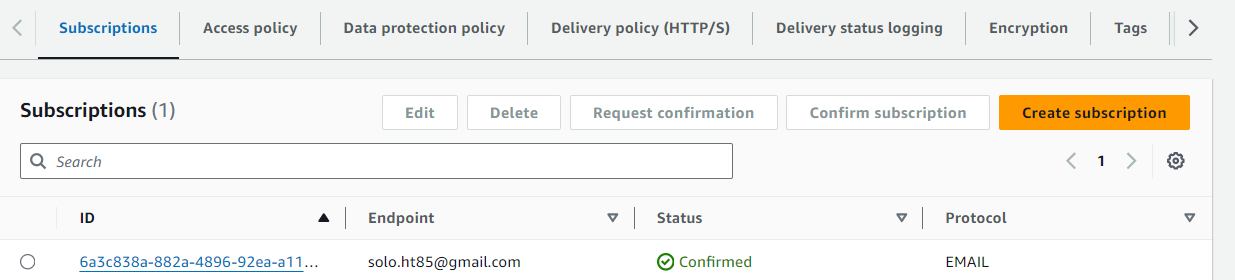
Since the CloudWatch Agent has already been configured to collect these logs and send them to CloudWatch Logs, the logs should appear in the CloudWatch console under the respective log groups (CustomNamespace/CPU, CustomNamespace/Memory, CustomNamespace/Network).  
  
  
  
  


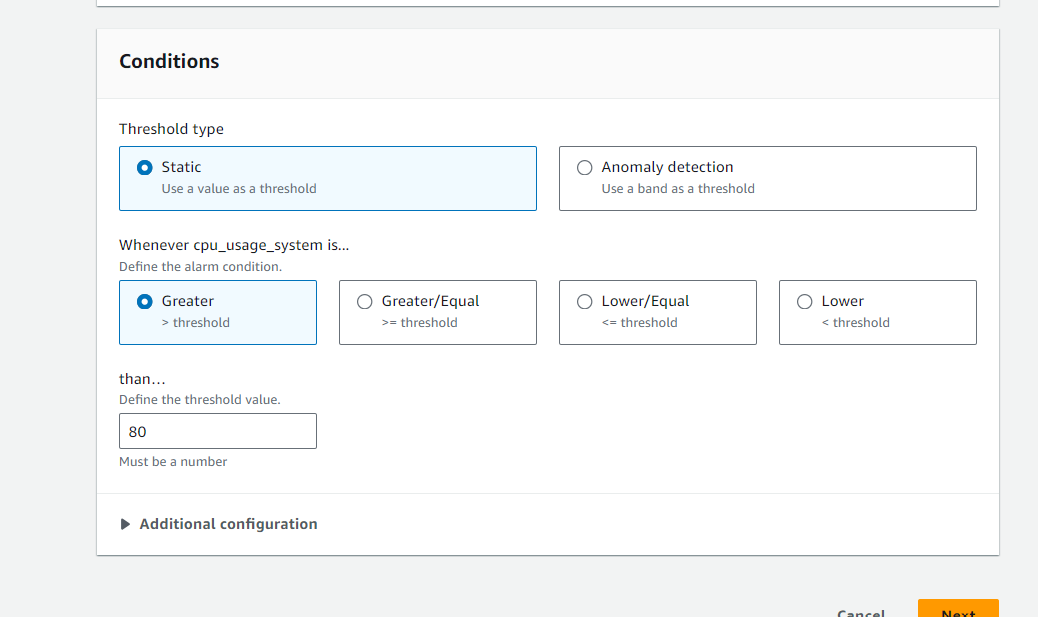
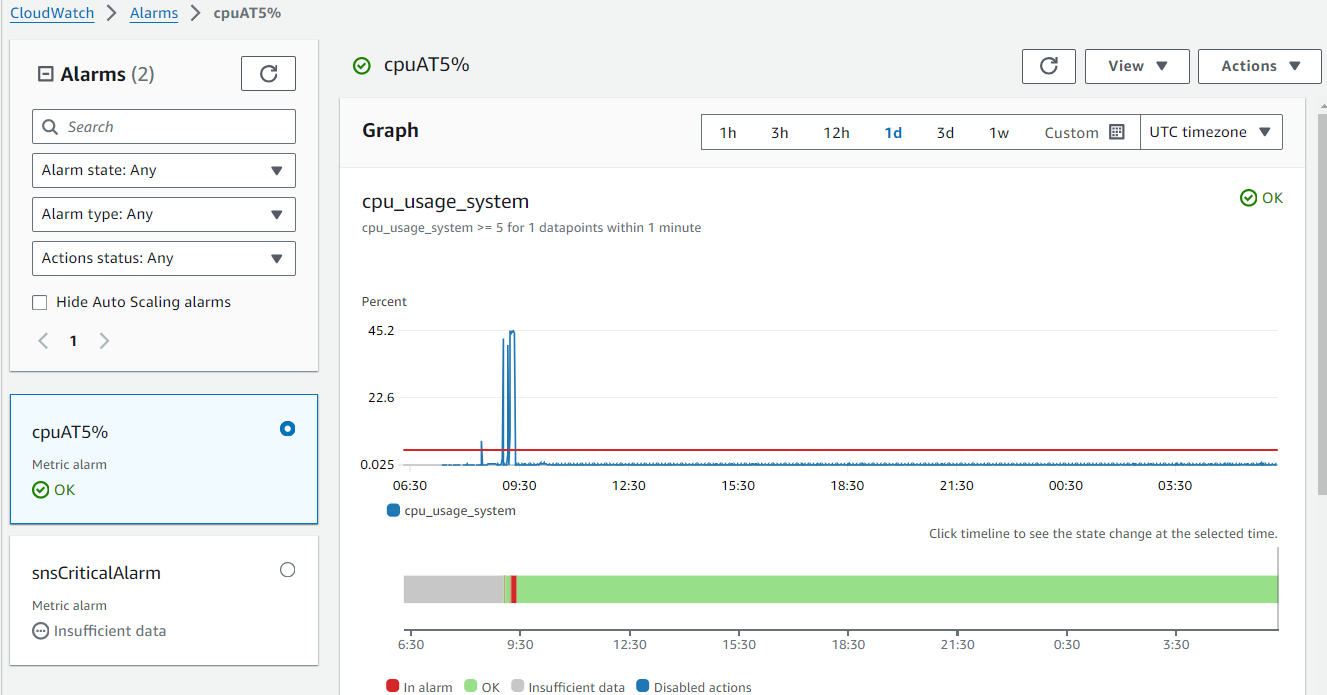
### **Triggers an alert (e.g., email, SMS) when certain thresholds are exceeded.**

1. Create a New Topic Name call (eg :- snsCriticalAlarm)

Create a Subscription:  
 Choose the Protocol (e.g., Email, SMS, HTTPS)  
 Enter the **Endpoint** (e.g., your email address or phone number)  
 Click "Create subscription"  
 Confirm the subscription if needed (e.g., by clicking a confirmation link sent to your email).



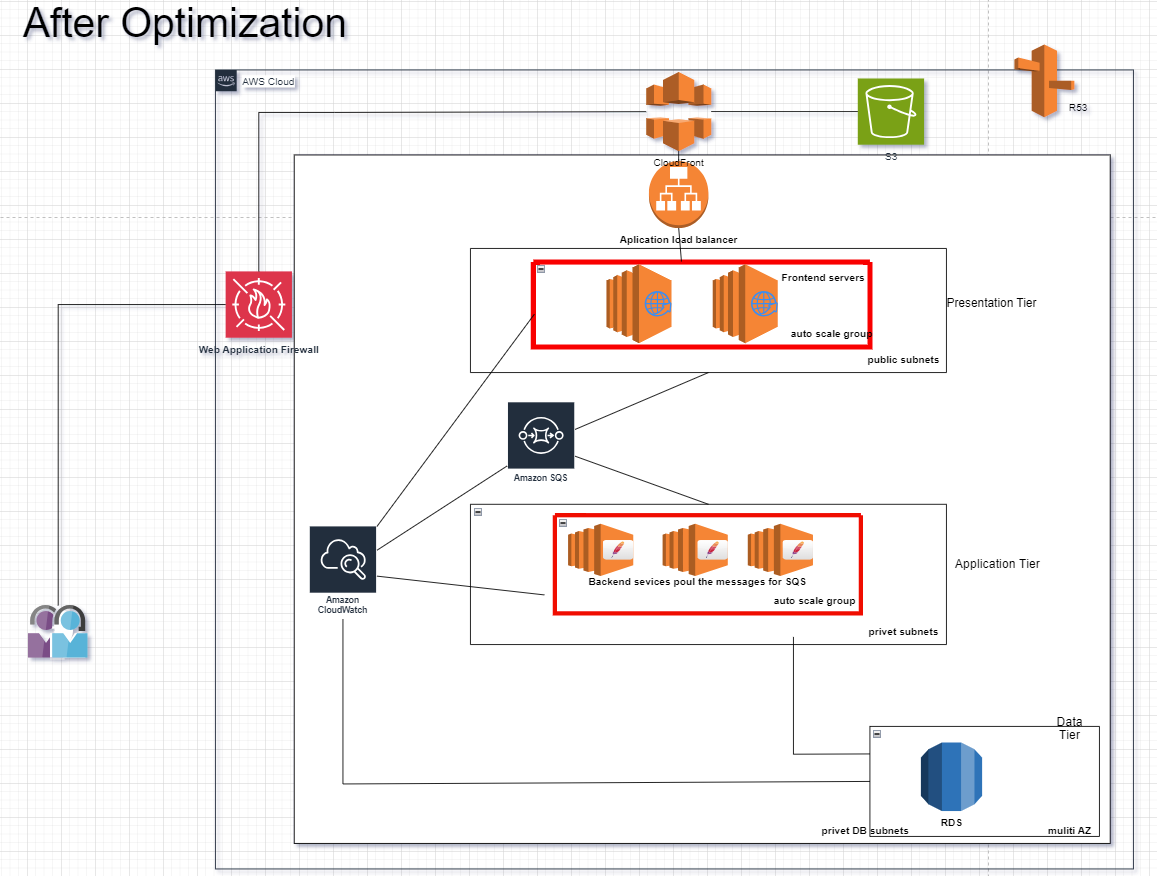


1. Create a CloudWatch Alarm  
     
   Navigate to alarms and create a New Alarm with a **"Select metric"**.  
     
   Eg: Set Alarm condition to trigger **Whenever cpu\_usage\_system is Greater/Equal to 80%** as below in the conditions section and setup.  
     
     
     
     
   

# **Optimization**

## **Resource Scaling**

Automatically scale resources (e.g., add/remove server instances) based on performance metrics.



Implemented Auto Scale group in Application Tire. This will cater the load according to the configured thresholds by add/remove server instances

On this case for adding /remove server instances based on the performance metrics (Eg: CPU usage ) we can use **Auto Scale Group**.

Also we can written all in the **AWS CloudFormation** and used to implement and manage infrastructure on AWS in an automated and consistent manner.

Steps:-

1. First we have to create “launch template” with the same configurations which we used to spin the earlier EC2 (we can add all the required installations/ configurations in “user data” section and crate this template).  
   Or  
   We can Create a AMI by using the already created/configured EC2 instance and use that AMI for the auto scale group.
2. Create an Auto Scaling Group with Choosing the **Launch Template** you created earlier.

Configure the Group Size and Scaling Policies:  
 Eg:-  
 Set the Desired Capacity: Enter 2 for the initial number of instances.  
 Minimum Capacity: Enter 2.  
 Maximum Capacity: Enter 6.

OR  
 Create a **CloudFormation** Script with using the created AMI ID and auto scale group.

Eg:-  
If we go with the **CloudFormation,** we can use the below code to implement the application tire infrastructure (cfdemo.yaml is a sample code and its written only to the application tire with **minimal of 1 instances and auto scale up to 4 instances** based on the **CPU performance** metrics and alarms. In this approach we can implement entire infastrctur by a **CloudFormation** script)   
CPUUtilizationHigh >= 70 (GreaterThanOrEqualToThreshold - CloudWatch Alarm for Scaling Out)

CPUUtilizationHigh >= 30 (LessThanThreshold - CloudWatch Alarm for Scaling In)

MinSize: 1

MaxSize: 4

cfdemo.yaml

AWSTemplateFormatVersion: '2010-09-09'

Description: CloudFormation template to create an Auto Scaling Group in the Stockholm (eu-north-1) region.

Resources:

# Launch Configuration

MyAppLaunchConfiguration:

Type: AWS::AutoScaling::LaunchConfiguration

Properties:

ImageId: ami-064c5012ae6debf9b # this AMI is available in the eu-north-1 region

InstanceType: t3.micro # instance type

KeyName: ec2key # <-- Add the KeyName property

IamInstanceProfile: !Ref MyAppInstanceProfile

SecurityGroups:

- sg-09585339288ead35a

# IAM Instance Profile

MyAppInstanceProfile:

Type: AWS::IAM::InstanceProfile

Properties:

Roles:

- myApp\_Roll

# Auto Scaling Group

MyAppAutoScalingGroup:

Type: AWS::AutoScaling::AutoScalingGroup

Properties:

VPCZoneIdentifier:

- subnet-0e642b393b88a1507 # Update these with your actual subnet IDs

- subnet-0e3c4cca4b69c4039

LaunchConfigurationName: !Ref MyAppLaunchConfiguration

MinSize: 1 # minimum size

MaxSize: 4 # scale up to

DesiredCapacity: 1

HealthCheckType: EC2

HealthCheckGracePeriod: 300

MetricsCollection:

- Granularity: "1Minute"

Tags:

- Key: Name

Value: MyAppInstance

PropagateAtLaunch: true

# Scaling Policy - Scale Out (Increase Capacity)

ScaleOutPolicy:

Type: AWS::AutoScaling::ScalingPolicy

Properties:

AutoScalingGroupName: !Ref MyAppAutoScalingGroup

PolicyType: SimpleScaling

AdjustmentType: ChangeInCapacity

Cooldown: 300

ScalingAdjustment: 1

# Scaling Policy - Scale In (Decrease Capacity)

ScaleInPolicy:

Type: AWS::AutoScaling::ScalingPolicy

Properties:

AutoScalingGroupName: !Ref MyAppAutoScalingGroup

PolicyType: SimpleScaling

AdjustmentType: ChangeInCapacity

Cooldown: 300

ScalingAdjustment: -1

# CloudWatch Alarm for Scaling Out

CPUUtilizationHigh:

Type: AWS::CloudWatch::Alarm

Properties:

AlarmDescription: "Alarm if CPU Utilization is greater than or equal to 70%"

MetricName: CPUUtilization

Namespace: AWS/EC2

Statistic: Average

Period: 300

EvaluationPeriods: 1

Threshold: 70

ComparisonOperator: GreaterThanOrEqualToThreshold

Dimensions:

- Name: AutoScalingGroupName

Value: !Ref MyAppAutoScalingGroup

AlarmActions:

- !Ref ScaleOutPolicy

# CloudWatch Alarm for Scaling In

CPUUtilizationLow:

Type: AWS::CloudWatch::Alarm

Properties:

AlarmDescription: "Alarm if CPU Utilization is less than 30%"

MetricName: CPUUtilization

Namespace: AWS/EC2

Statistic: Average

Period: 300

EvaluationPeriods: 1

Threshold: 30

ComparisonOperator: LessThanThreshold

Dimensions:

- Name: AutoScalingGroupName

Value: !Ref MyAppAutoScalingGroup

AlarmActions:

- !Ref ScaleInPolicy

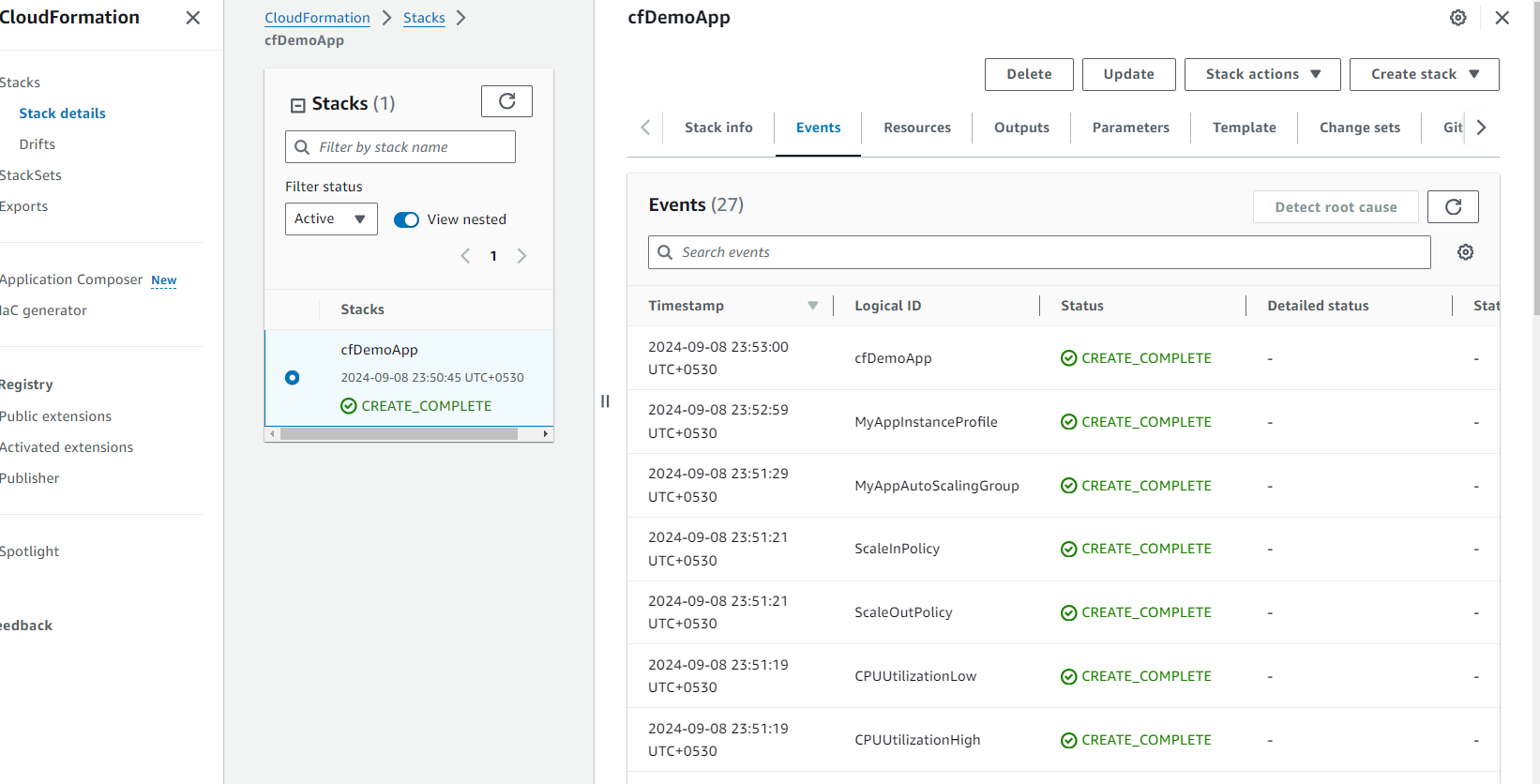
Outputs:

AutoScalingGroupName:

Description: "Name of the Auto Scaling Group"

Value: !Ref MyAppAutoScalingGroup

Test Sample in CF



# **Optional Enhancements**

## **Log Rotation**

Implement a log rotation script to manage log files, archiving or deleting old logs after a specified period.  
  
We can use the “logrotate.d” to get this done. To ensuring logs are regularly rotated, compressed, and cleaned up we can implement this.  
  
Create a file named /etc/logrotate.d/myapp

/var/log/myapp/\*.log {

daily # Rotate logs daily

rotate 7 # Keep only 7 rotations (one week of logs)

compress # Compress the rotated logs

delaycompress # Delay compression to the next rotation cycle

missingok # Ignore missing files and errors

notifempty # Do not rotate empty logs

dateext # Append date to rotated logs

dateformat -%Y-%m-%d # Specify date format in filenames

create 0644 root root # Specify permissions for new log files

olddir /var/log/myapp/archive # Archive rotated logs in a separate directory

postrotate

find /var/log/myapp/archive -mtime +90 -type f -delete # Delete logs older than 90 days

endscript

}

Configuration Logic:-  
  
Daily Rotation: Logs in /var/log/myapp/ are rotated daily.

File Naming: Rotated log files will have a date suffix (e.g., app.log-2024-09-08).

Compression: Logs older than the most recent rotation are compressed (e.g., app.log-2024-09-07.gz).

Archive Management: Rotated logs are moved to /var/log/myapp/archive.

Retention Policy: Only the last 7 days' worth of logs are kept in rotation.

Cleanup of Old Logs: Any log files older than 90 days in the archive directory are deleted to manage disk space.  
  
To Test the configurations manually.  
  
# sudo logrotate -f /etc/logrotate.d/myapp

**OR**

we can set this in a shell script like below.  
  
Create a configuration file named log\_rotation.conf to store parameters.

# log\_rotation.conf

# Directory containing log files

LOG\_DIR="/var/log/myapp"

# Directory to archive old logs

ARCHIVE\_DIR="/var/log/myapp/archive"

# Log file pattern

LOG\_PATTERN="\*.log"

# Number of days to retain logs

RETENTION\_DAYS=90

Shell script

#!/bin/bash

# Load configuration

source /path/to/log\_rotation.conf

# Create archive directory if it does not exist

mkdir -p "$ARCHIVE\_DIR"

# Move rotated logs to archive and compress them

for log\_file in "$LOG\_DIR"/$LOG\_PATTERN; do

if [ -f "$log\_file" ]; then

# Move the log file to the archive directory with a date suffix

mv "$log\_file" "$ARCHIVE\_DIR/$(basename "$log\_file")-$(date +'%Y-%m-%d')"

# Compress the archived log file

gzip "$ARCHIVE\_DIR/$(basename "$log\_file")-$(date +'%Y-%m-%d')"

fi

done

# Delete archived logs older than $RETENTION\_DAYS days

find "$ARCHIVE\_DIR" -type f -mtime +$RETENTION\_DAYS -exec rm -f {} \;

To make the Script Executable  
  
# chmod +x /path/to/log\_rotation.sh  
  
Add this cron to schedule the script to run daily at midnight:  
  
# 0 0 \* \* \* /path/to/log\_rotation.sh

## **System Simulation**

Write a script to simulate peak load conditions and measure the system’s performance, allowing you to test the effectiveness of your optimizations.  
  
To this we can use Locust ( A scalable load testing tool written in Python.)  
  
Steps:-  
1. Install Locust: Install Locust using pip:

sudo yum install python3-pip -y  
pip install locust   
  
2. Create a Locust File: Create a file named locustfile.py

from locust import HttpUser, TaskSet, task, between

class UserBehavior(TaskSet):

@task

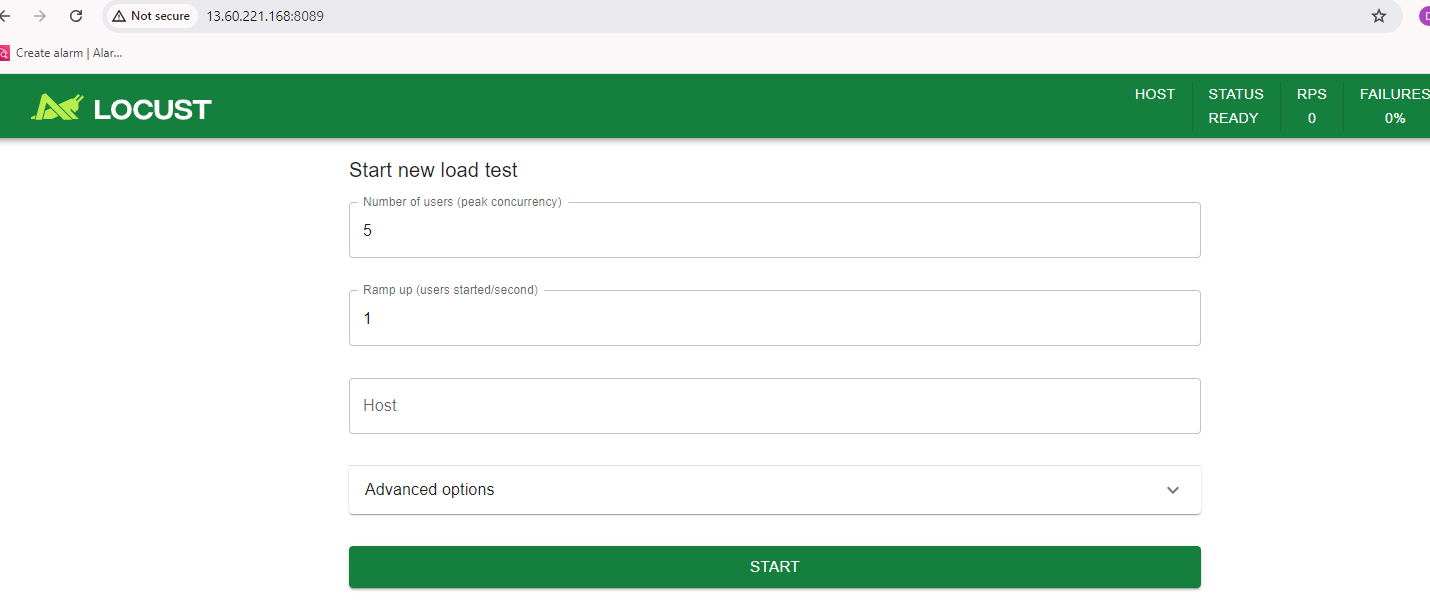
def get\_api(self):

self.client.get**("/your-api-endpoint")**

class WebsiteUser(HttpUser):

tasks = [UserBehavior]

wait\_time = between(1, 5) # Wait time between tasks

Execute Locust by  
  
# locust -f locustfile.py  
  
Open a browser and navigate to http://<IP>:8089 to configure and start the load test.  
(Need to configure inbound rule in security group allows traffic on port 8089)  
  
  
  
  
  
**OR**

In the optimization section, the proposed solution (automatically scaling resources by adding or removing server instances based on performance metrics) can be tested using a CPU stress test.  
  
  
Steps: -   
  
Run a CPU stress test to trigger scaling:-

# sudo yum install stress -y # For Amazon Linux or CentOS

# stress --cpu 2 --timeout 300 # Stress test for 5 minutes

Check the scaling actions in the **Auto Scaling Activity** section of the Auto Scaling group to see, instances are launching when CPU utilization exceeds the threshold.  
  
Auto Scaling group in AWS that automatically scales EC2 instances based on CPU utilization, with a starting capacity of 1 and a maximum of 4 instances. Once the CPU utilization decreases and stays low, the Auto Scaling group will automatically reduce the number of instances to match the lower load, scaling down to as few as 2 instances, according to your configuration.

## **Automated Deployment**

Develop a script to automate the deployment of the optimized system setup, making it easy to replicate in a different environment.

We can use **AWS CloudFormation** service to implement and manage infrastructure on AWS in an automated and consistent manner (infrastructure as code (IaC) using templates, which can be written in JSON or YAML format)  
We can provision the entire infrastructure stack, including VPC, subnets, EC2 instances for the web and application tiers, RDS for the database tier, and S3 buckets for static content.  
  
Below mentioned cfdemo.yaml is a sample template specifically designed for the application tier, with a minimum of 1 instance and the ability to automatically scale up to 4 instances based on CPU performance metrics and alarms. With this approach, we can deploy the entire infrastructure using a single CloudFormation script in a different environments.  
  
GitHub Code link for the Sample CF code.  
  
<https://github.com/harshat85/MadMobileAssignment/blob/main/CFdemo.yaml> (**CFdemo.yaml**)