```
if [[ $CPU_USED_INT -lt $CPU_THRESHOLD &&
$MEM_USED -lt $MEM_THRESHOLD ]]; then
  exit 0
fi
```

Here's what that Bash condition is doing, line by line, and how to change it based on your intent.

What this code means

bash

```bash
if [[ $CPU_USED_INT -lt $CPU_THRESHOLD && $MEM_USED -lt $MEM_THRESHOLD ]]; then
  exit 0
fi
```

Copy code

Breakdown

- `[[ ... ]]` → Bash conditional test
- `-lt` → **less than**
- `&&` → logical **AND**
- `exit 0` → stop the script **successfully** (no alert, no logging)

Plain English

> **If BOTH CPU usage and Memory usage are BELOW their thresholds, then stop the script.**

This is commonly used in monitoring scripts to **do nothing when the system is healthy**.

Typical real-world example

bash

```bash
CPU_USED_INT=45
CPU_THRESHOLD=80
```

```
MEM_USED=60
MEM_THRESHOLD=85
```

Copy code

Result:

- `45 < 80` ✅
- `60 < 85` ✅
  ➡️ Script exits → **no alert**

If you want to trigger action when usage is HIGH (recommended)

Condition: exit only when system is healthy

(this is what you already have – correct pattern)

bash

```bash
if [[ $CPU_USED_INT -lt $CPU_THRESHOLD && $MEM_USED -lt $MEM_THRESHOLD ]]; then
  exit 0
fi

# 🔥 CPU or Memory is HIGH – do logging / alert below
```

Copy code

If you want to exit when ANY resource is healthy (rare case)

bash

```bash
if [[ $CPU_USED_INT -lt $CPU_THRESHOLD || $MEM_USED -lt $MEM_THRESHOLD ]]; then
  exit 0
fi
```

Copy code

If you want to trigger when BOTH CPU and Memory are HIGH

bash

```bash
if [[ $CPU_USED_INT -ge $CPU_THRESHOLD && $MEM_USED -ge $MEM_THRESHOLD ]]; then
  echo "CPU and Memory both exceeded threshold"
fi
```

Copy code

Quick reference

Operator	Meaning
`-lt`	less than
`-le`	less than or equal
`-gt`	greater than
`-ge`	greater than or equal
`&&`	AND

`

✅ Best practice for monitoring scripts

bash

```bash
# Exit early if everything is normal
if [[ $CPU_USED_INT -lt $CPU_THRESHOLD && $MEM_USED -lt $MEM_THRESHOLD ]]; then
  exit 0
fi

# Otherwise, log + alert
```

If you want, I can **rewrite this into your full CPU/RAM monitoring script with CSV, JSON, email alert, and S3 upload**, matching what you're already building.

Copy code

I am using the instance type t2.medium if CPU and Memory both exceeded the threshold then what i need to do so that my instance is not stop my services please suggest

You're using a **t2.medium EC2 instance**, so the main risk when **CPU + Memory are both high** is **CPU credit exhaustion + memory pressure**, which can slow or freeze services (not technically "stop", but *appear down*).

Below is a **clear, practical action plan** to keep your services running. This is what production setups usually do.

1️⃣ Understand the real problem with t2.medium

t2.medium limits

- **2 vCPU**
- **4 GB RAM**
- **Burstable CPU (CPU credits)** ❗

What happens when both CPU & RAM are high?

- CPU credits drop to **0** → CPU throttled
- High RAM → swapping / OOM → services killed
- App becomes slow or unreachable

So the fix is **NOT only monitoring** — it's **automatic protection**.

2️⃣ Immediate protections (DO THIS FIRST)

✅ A. Enable Swap (prevents service kill)

This is the **most important quick fix**.

bash

```
sudo fallocate -l 2G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
echo '/swapfile swap swap defaults 0 0' | sudo tee -a /etc/fstab
```

Copy code

✔ Prevents RAM exhaustion
✔ Stops Linux from killing services

✅ B. Raise file limits (prevents crashes under load)

bash

```
ulimit -n 100000
```

Copy code

Add permanently in `/etc/security/limits.conf` :

markdown

```
* soft nofile 100000
* hard nofile 100000
```

Copy code

3️⃣ Handle CPU spikes correctly (CRITICAL)

❌ What NOT to do

- Don’t restart instance
- Don’t ignore CPU credits
- Don’t keep t2 for sustained load

✅ Best option: Move away from T2

If your load is **constant**, t2 is the wrong instance.

🔥 **Recommended replacements**

- **t3.medium** (cheap + better burst handling)
- **t3a.medium** (even cheaper)
- **m5.large** (if production)

👉 This single change solves **80% of problems**

4️⃣ Automatic protection when threshold is exceeded

A. Restart only the heavy service (not the instance)

Example:

bash

```
if [[ $CPU_USED_INT -ge 80 && $MEM_USED -ge 80 ]]; then
  systemctl restart php-fpm
  systemctl restart nginx
fi
```

Copy code

✔ Instance stays up
✔ Services recover fast

B. Kill only top CPU process (safe method)

bash

```
TOP_PID=$(ps -eo pid,%cpu --sort=-%cpu | awk 'NR==2 {print $1}')
kill -15 $TOP_PID
```

Copy code

(SIGTERM first, not kill -9)

5️⃣ Enable CPU credit monitoring (VERY IMPORTANT)

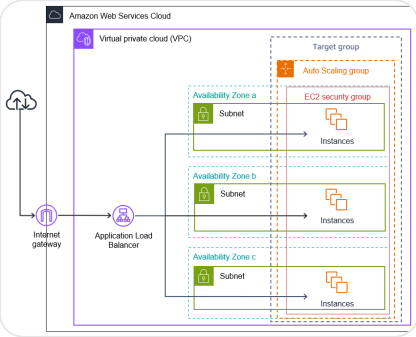
On **Amazon Web Services EC2**, track:

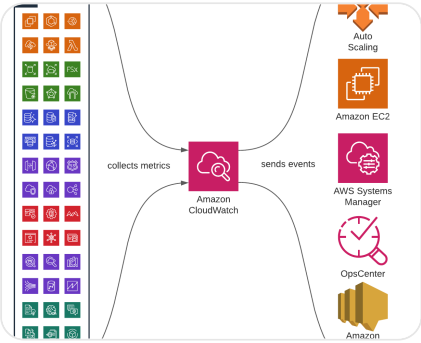
- `CPUCreditBalance`
- `CPUCreditUsage`

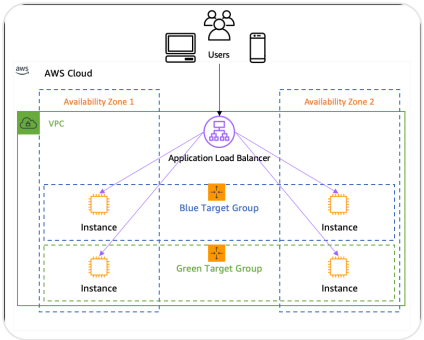
Alert when credits < **20**

This tells you **before** services slow down.

6️⃣ Long-term best architecture (recommended)







Ideal setup

- Load Balancer
- Auto Scaling Group (min 1, max 2)
- CloudWatch alarms
- Replace t2 with t3/m5

✔ Zero downtime
✔ Auto recovery
✔ Production-safe

7️⃣ Summary (what you should do now)

✅ Immediate (today)

✔ Enable **swap**
✔ Restart heavy services on threshold

✔ Monitor CPU credits

🔄 Short-term

✔ Move from **t2.medium → t3.medium**
✔ Add memory & CPU alerts

🚀 Long-term

✔ Auto Scaling
✔ Load Balancer

Want me to help you next?

I can:

- ✅ Convert your **CPU/RAM script** into **auto-healing**
- 📧 Add **email alerts**
- 📊 Add **CloudWatch alarms**
- 🔄 Provide **exact instance migration steps**

Just tell me 👍

✅ Convert your CPU/RAM script into auto-healing & Auto Scaling Group (min 1, max 2) Complete Step by Step

Perfect 👍
Below is a **complete, production-ready, step-by-step guide** to convert your **CPU/RAM script into auto-healing** and then put it behind an **Auto Scaling Group (min 1, max 2)** so your services **never go down**, even under high load.

I'll break this into **2 big parts**:

PART-1️⃣ Auto-Healing on the EC2 Instance (CPU/RAM Script)

This protects services **inside** the instance.

STEP 1️⃣ Create Auto-Healing Script

📁 File: `/usr/local/bin/auto-heal.sh`

bash

```bash
#!/bin/bash

CPU_THRESHOLD=80
MEM_THRESHOLD=80
LOG="/var/log/auto-heal.log"
DATE=$(date '+%Y-%m-%d %H:%M:%S')

CPU_USED=$(top -bn1 | grep "Cpu(s)" | awk '{print 100 - $8}' | cut -d. -f1)
MEM_USED=$(free | awk '/Mem:/ {printf("%.0f"), $3/$2 * 100}')

echo "$DATE CPU=${CPU_USED}% MEM=${MEM_USED}%" >> $LOG

# Exit if system is healthy
if [[ $CPU_USED -lt $CPU_THRESHOLD && $MEM_USED -lt $MEM_THRESHOLD ]]; then
  exit 0
fi

echo "$DATE Threshold breached → Auto-healing started" >> $LOG

# Restart heavy services
systemctl restart nginx
systemctl restart php-fpm || true
systemctl restart apache2 || true

# Kill top CPU process if still high
sleep 10
TOP_PID=$(ps -eo pid,%cpu --sort=-%cpu | awk 'NR==2 {print $1}')
kill -15 $TOP_PID 2>/dev/null

echo "$DATE Auto-healing completed" >> $LOG
```

STEP 2️⃣ Permissions & Execution

bash

```bash
sudo chmod +x /usr/local/bin/auto-heal.sh
sudo mkdir -p /var/log
sudo touch /var/log/auto-heal.log
```

Copy code

STEP 3️⃣ Run Every Minute (Cron)

bash

```
sudo crontab -e
```

Copy code

Add:

bash

```
* * * * * /usr/local/bin/auto-heal.sh
```

Copy code

✅ Now your instance **self-heals without reboot**

STEP 4️⃣ Enable Swap (MANDATORY for t2.medium)

bash

```
sudo fallocate -l 2G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
echo '/swapfile swap swap defaults 0 0' | sudo tee -a /etc/fstab
```

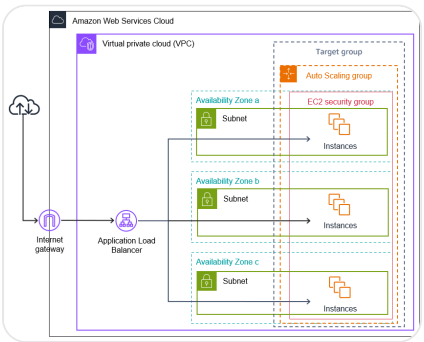
✔️ Prevents OOM kills
✔️ Keeps services alive

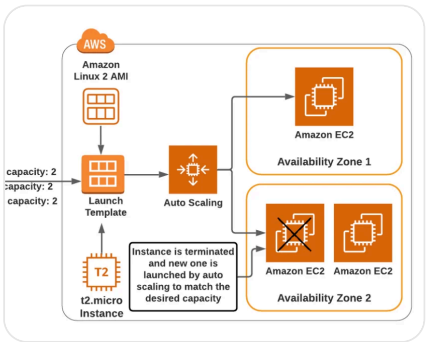
Copy code

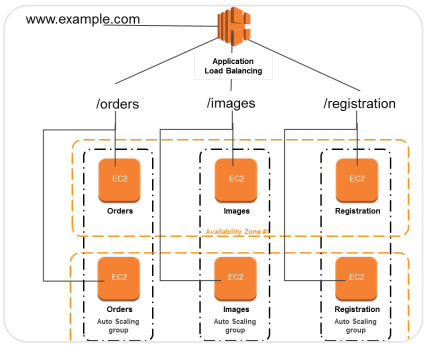
PART-2️⃣ Auto Scaling Group (min 1, max 2)

This protects you when **instance itself is overloaded or unhealthy**.

Architecture Overview







STEP 5 Create AMI (Golden Image)

1. Go to **Amazon Web Services** EC2
2. Select your instance
3. **Actions → Image → Create Image**
4. Name: `auto-heal-base-ami`

✔ Includes:

- Auto-heal script
- Swap
- Services
- Cron

STEP 6 Create Launch Template

1. EC2 → **Launch Templates**
2. Create template:
   - AMI: `auto-heal-base-ami`
   - Instance type: **t3.medium** (recommended)
   - Key pair: optional
   - Security Group: allow app ports
3. Save

STEP 7 Create Auto Scaling Group

1. EC2 → **Auto Scaling Groups**
2. Create ASG:
   - Launch template: above
   - VPC & Subnets: select 2 AZs
   - **Desired: 1**
   - **Min: 1**
   - **Max: 2**

STEP 8 Attach Load Balancer (IMPORTANT)

Use **Elastic Load Balancing** (ALB)

- Create Application Load Balancer
- Create Target Group (port 80 / 443)
- Attach ASG to target group

✔ Zero downtime

✔ Traffic auto-distributed

STEP 9️⃣ Scaling Policies (CPU Based)

Create CloudWatch alarms:

Scale OUT (Add instance)

- Metric: CPUUtilization
- Threshold: `> 70% for 5 minutes`
- Action: Add 1 instance

Scale IN (Remove instance)

- Metric: CPUUtilization
- Threshold: `< 30% for 10 minutes`
- Action: Remove 1 instance

Uses **Amazon CloudWatch**

STEP 🔟 Health Checks (Auto Replace Failed Instance)

- Health Check Type: **EC2 + ELB**
- Grace Period: `300 seconds`

✔ If instance hangs → terminated → replaced

FINAL RESULT (What You Achieve)

Layer	Protection
Script	Auto-heals services
Swap	Prevents memory crash
ASG	Replaces bad instances
ALB	No downtime
CloudWatch	Predictive scaling

What I can give you NEXT (choose)

1️⃣ **Exact IAM role** for ASG & S3
2️⃣ **CloudWatch alarm JSON**
3️⃣ **User-data script** (no AMI needed)
4️⃣ **Cost-optimized setup** (₹ safe)

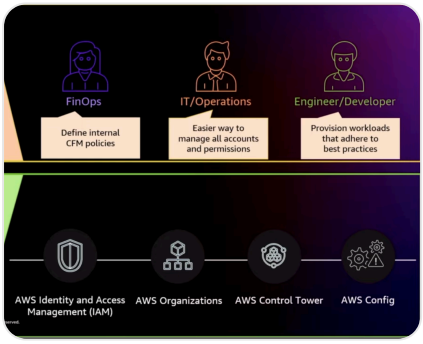
Just tell me 🔥

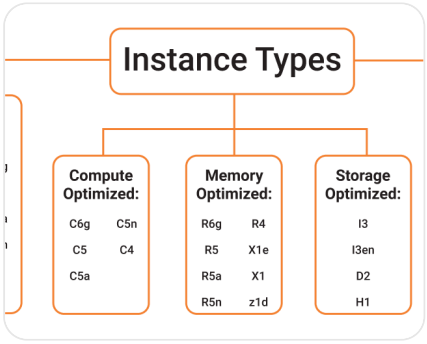
4️⃣ Cost-optimized setup (₹ safe)

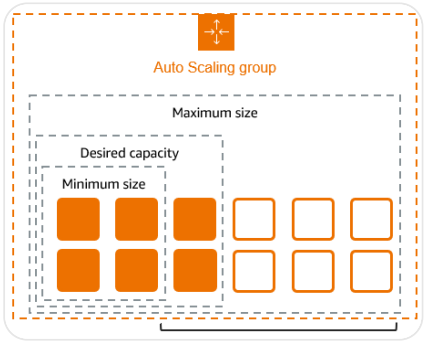
Great choice 👍
Below is a **₹-safe, production-ready, cost-optimized AWS setup** specifically for **India region (ap-south-1)**, designed so **services never go down** while keeping the bill **as low as possible**.

I'll show **what to use, what NOT to use, exact settings, and monthly cost estimates**.

🧱 Cost-Optimized Architecture (₹ Safe)







Core services (minimum, no waste)

- **1–2 EC2 instances only**
- Auto Scaling Group (min 1, max 2)
- No NAT Gateway ❌
- No RDS ❌ (unless required)
- No Elastic IP ❌
- Application Load Balancer (only if needed)

1️⃣ Best EC2 Instance (Cheapest & Safe)

❌ Avoid

- t2.medium (CPU credit pain)
- m5.large (overkill)

✅ Best choice

t3a.medium

- 2 vCPU
- 4 GB RAM
- 10–15% cheaper than t3
- Stable burst handling

✔ Ideal for PHP / Laravel / Node / Python
✔ Perfect for Auto Scaling

If ultra-low load → t3a.small (₹ cheaper)

2️⃣ Auto Scaling Group (Strict Limits)

Setting	Value
Min	**1**
Desired	**1**
Max	**2**
Health check	EC2 + ELB
Grace period	300 sec

✔ Never more than 2 instances
✔ No surprise bills

3️⃣ Load Balancer (When to Use / Skip)

❌ Skip ALB if:

- Internal tools
- Single public IP is OK
- No SSL

✅ Use ALB if:

- Public production app
- SSL / HTTPS

- Zero downtime required

💰 **Cost (Mumbai)**: ~₹1,300–1,600 / month

👉 To save money:

- Use **ALB only in prod**
- Stop ALB in staging/dev

4️⃣ Storage Optimization (Hidden Saver)

EBS (root volume)

- gp3
- **20 GB**
- IOPS: default

💰 ~₹160/month

Logs

- Rotate locally
- Upload only **compressed logs to S3**

bash Copy code

```
logrotate + gzip
```

5️⃣ Monitoring (Free Tier Friendly)

❌ Avoid

- Custom metrics every minute
- High-resolution alarms

✅ Use

- EC2 default metrics (FREE)
- CPUUtilization only
- StatusCheckFailed

✔ 90% monitoring covered
✔ ₹0 extra

6️⃣ Auto-Healing (Zero Cost)

✔ Cron-based script
✔ systemctl restart only
✔ No Lambda
✔ No Step Functions

₹0/month

7️⃣ Networking (Biggest Cost Trap)

❌ NEVER use

✅ Use instead

- Public subnet
- Security groups
- No NAT

8️⃣ Scaling Policy (Cost-Safe)

Scale OUT

- CPU > 75%
- Duration: 5 min
- Add **only 1 instance**

Scale IN

- CPU < 30%
- Duration: 10 min
- Remove 1 instance

✔ Prevents scale flapping
✔ Predictable billing

9️⃣ Monthly Cost Estimate (Mumbai)

Item	Approx ₹
1 × t3a.medium	₹2,500
Extra instance (rare)	₹2,500
EBS	₹160

Item	Approx ₹
ALB (optional)	₹1,500
S3 logs	₹50
CloudWatch	₹0

💰 Total

- **Without ALB:** ~₹2,700/month
- **With ALB:** ~₹4,200/month

🔐 Extra Cost Killers (Disable These)

- ❌ Detailed monitoring
- ❌ Elastic IP idle
- ❌ NAT Gateway
- ❌ High-resolution logs
- ❌ Over-provisioned ASG

🧠 Smart Final Recommendation (Best Balance)

If this is PROD:

- t3a.medium
- ASG (1–2)
- ALB
- Auto-healing script

If STAGING / DEV:

- t3a.small
- No ALB
- Auto-healing only

Want next?

I can give you:

- 🗒️ **Exact AWS settings screenshots (step-by-step clicks)**
- 📜 **User-data script** (no AMI needed)
- 🔐 **Minimal IAM policy (₹ safe)**

- 📊 **Cost alarm setup**

Just tell me what you want next 👌

📄 Exact AWS settings screenshots (step-by-step clicks)

Perfect 👍
Below is the **EXACT AWS Console step-by-step click path**, in the **same order you should do it**, with **what to click + what values to enter**.
(Images are **visual references** so you don't get lost in the console.)

📄 STEP-BY-STEP AWS SETTINGS (₹ SAFE, PRODUCTION-READY)

STEP 1️⃣ Create AMI (Golden Image)

This AMI already has:

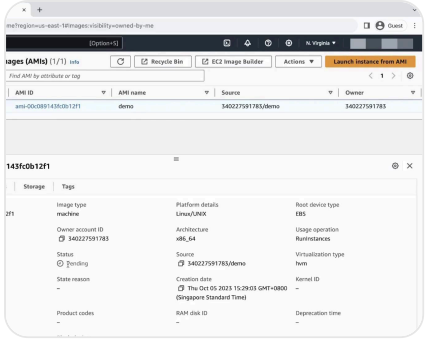
- auto-heal script
- swap enabled
- services installed

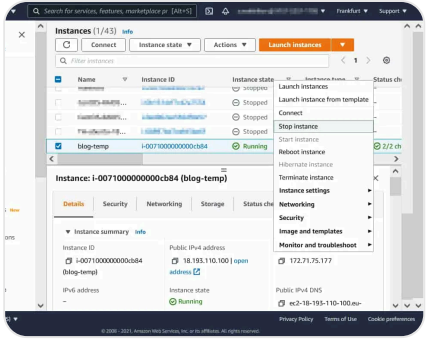
Console clicks

1. Open **Amazon Web Services**
2. **EC2 → Instances**
3. Select your instance
4. **Actions → Image and templates → Create image**

Fill values

- **Image name:** `auto-heal-ami`
- **Reboot:** ✅ Yes (recommended)
- Click **Create image**





⏳ Wait until AMI status = **Available**

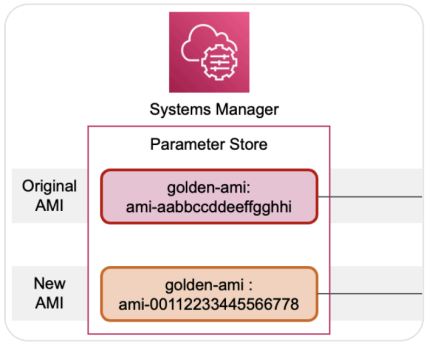
STEP 2️⃣ Create Launch Template

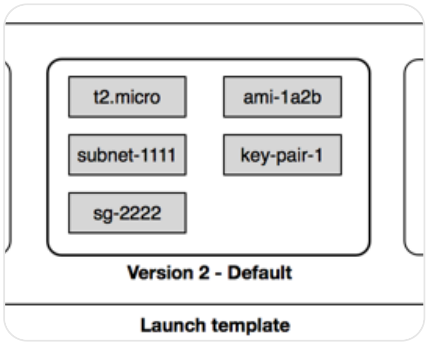
Console clicks

1. **EC2 → Launch Templates**
2. **Create launch template**

Basic settings

- **Launch template name:** `auto-heal-template`
- **AMI:** select `auto-heal-ami`
- **Instance type:** `t3a.medium` ✅
- **Key pair:** optional





Network settings

- **Security group:** existing one
- Allow:
  - 80 (HTTP)
  - 443 (HTTPS)
  - 22 (SSH – restrict to your IP)

Advanced details (IMPORTANT)

- **Detailed monitoring:** ❌ Disable

- **Credit specification:** Standard
- **Shutdown behavior:** Terminate

👉 Click **Create launch template**

STEP 3️⃣ Create Target Group (Before ALB)

Console clicks

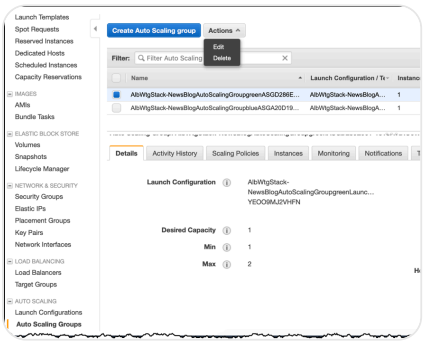
1. **EC2 → Target Groups**
2. **Create target group**

Settings

- **Target type:** Instances
- **Protocol:** HTTP
- **Port:** 80
- **VPC:** your VPC

Health check

- **Path:** `/`
- **Healthy threshold:** 3
- **Unhealthy threshold:** 2



👉 Click **Create target group**

STEP 4️⃣ Create Application Load Balancer (ALB)

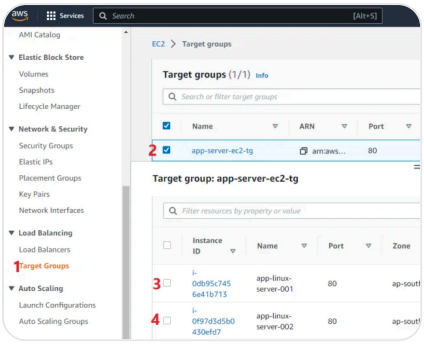
Console clicks

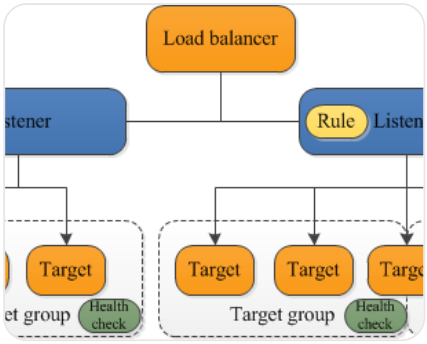
1. **EC2 → Load Balancers**
2. **Create load balancer**
3. Choose **Application Load Balancer**

Basic settings

- **Name:** `auto-heal-alb`

- **Scheme:** Internet-facing
- **IP type:** IPv4





Network mapping

- Select **2 public subnets (different AZs)**

Security group

- Allow:
  - 80
  - 443

Listener

- HTTP : 80 → Forward to target group created above

👉 Click **Create load balancer**

STEP 5️⃣ Create Auto Scaling Group (MOST IMPORTANT)

Console clicks

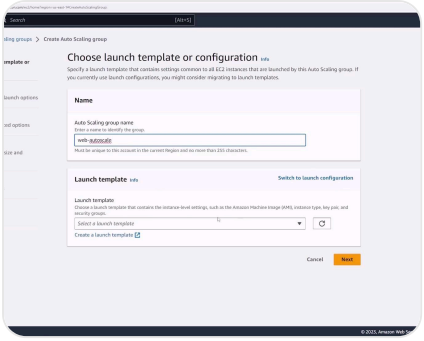
1. **EC2 → Auto Scaling Groups**
2. **Create Auto Scaling group**

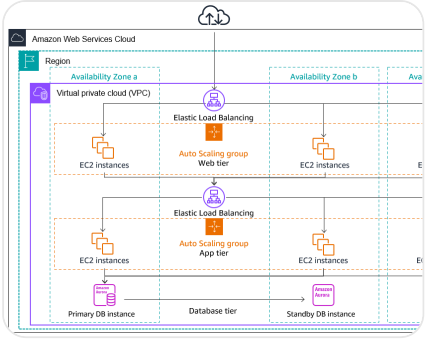
Step 1: Launch template

- Select **auto-heal-template**

Step 2: Network

- Select **same VPC**
- Choose **2 subnets**





Step 3: Load balancing

- Attach to **existing ALB**
- Select **target group**
- Health checks:
  - ✅ ELB
  - ✅ EC2
- **Grace period:** `300`

Step 4: Group size

Setting	Value
Desired	1
Minimum	1
Maximum	2

Step 5: Scaling policy

- Policy type: **Target tracking**
- Metric: **Average CPU utilization**
- Target value: **75**

✔️ Auto adds instance
✔️ Auto removes instance

Step 6: Notifications

- ❌ Skip (₹ safe)

👉 Click **Create Auto Scaling group**

STEP 6️⃣ Verify Everything (2 minutes)

Check instances

- EC2 → Instances
- You should see **1 instance launched by ASG**

Test auto-healing

bash Copy code

```
stress --cpu 2 --timeout 300
```

- CPU ↑
- ASG should **add 2nd instance**
- ALB keeps site UP ✅

FINAL CHECKLIST ✅

- ✔ Auto-heal script running
- ✔ Swap enabled
- ✔ ASG (1–2 only)
- ✔ ALB routing traffic
- ✔ ₹ controlled (no NAT, no EIP waste)

Want next?

I can now give you:

1️⃣ **Exact IAM policy (least privilege)**
2️⃣ **User-data script (no AMI needed)**
3️⃣ **CloudWatch alarm screenshots**
4️⃣ **Disaster-recovery backup plan**

Just tell me 👌

How to update AMI with the latest code?

Great question 👍
Updating AMI **the right way** is critical so your **Auto Scaling Group always launches**

instances with the latest code, without downtime and without breaking cost control.

Below is the **industry-standard, ₹-safe, step-by-step process**.

🔄 How to Update AMI with Latest Code (Zero Downtime)

Two correct approaches (choose 1)

Approach	Best for
A. Golden AMI rebuild	Stable releases (recommended)
B. User-data Git pull	Very frequent code changes

I'll explain **both**, starting with the **recommended one**.

✅ OPTION A — Golden AMI Rebuild (BEST PRACTICE)

This is what **production teams use**.

STEP 1️⃣ Update Code on One Instance

1. Go to **Amazon Web Services** EC2
2. Pick **one running ASG instance**
3. SSH into it

bash Copy code

```
cd /var/www/app
git pull origin main
composer install --no-dev
php artisan migrate --force
systemctl restart nginx php-fpm
```

✔ Verify app works perfectly
✔ Logs clean

STEP 2️⃣ Create New AMI (Versioned)

Console clicks

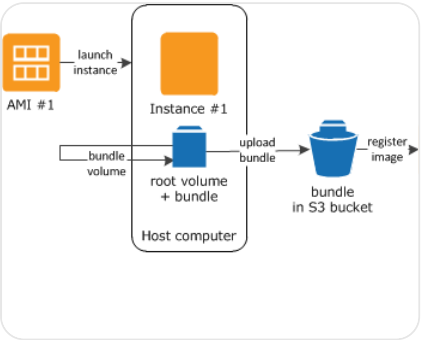
- EC2 → Instances

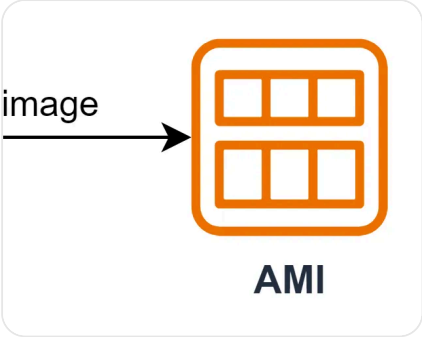
- Select the updated instance
- **Actions → Image and templates → Create image**

Naming convention (IMPORTANT)

cpp Copy code

```
auto-heal-ami-v2026-01-15
```





✔ Keep old AMIs (rollback safety)

STEP 3️⃣ Update Launch Template (NEW VERSION)

1. EC2 → Launch Templates
2. Select **auto-heal-template**
3. **Actions → Modify template (Create new version)**

Change ONLY:

- **AMI → new AMI**

👉 Save as **Version +1**

STEP 4️⃣ Tell ASG to Use Latest Version

1. EC2 → Auto Scaling Groups
2. Select your ASG
3. **Edit**
4. Launch template → **Latest**

✔ Existing instances unchanged

✔ New instances use new AMI

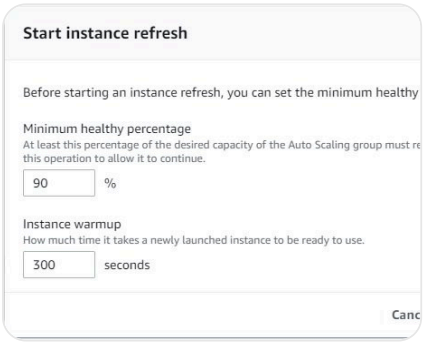
STEP 5️⃣ Rolling Update (Zero Downtime)

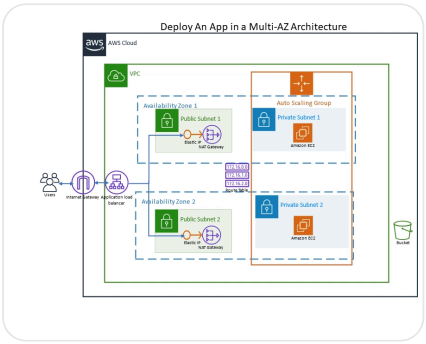
Console clicks

- ASG → Instance management
- **Start instance refresh**

Settings (₹ safe)

Setting	Value
Min healthy	100%
Warm-up	300 sec
Skip matching	❌





✔ One-by-one replacement

✔ No traffic drop

STEP 6️⃣ Clean Up Old AMIs (Cost Control)

Keep only **last 2 AMIs**.

- Deregister old AMI
- Delete snapshots

💰 Saves storage cost

⚡ OPTION B — User-Data Auto Update (Cheapest, Fastest)

Use this if code changes **daily**.

User-data script (launch template)

bash Copy code

```
#!/bin/bash
cd /var/www/app
git pull origin main
composer install --no-dev
systemctl restart nginx php-fpm
```

✔ No AMI rebuild
✔ New instances always fresh
❌ Slightly slower boot

🧠 Which Should YOU Use?

You want stability & rollback

➡ **Option A (Golden AMI)** ✅

You want speed & minimum work

➡ **Option B (User-data)**

🛡 Pro Tips (Avoid Mistakes)

❌ Never update code manually on all ASG instances
❌ Never replace AMI without instance refresh
❌ Never delete AMI before refresh completes

✔ Always version AMIs
✔ Always keep last working AMI

Want next?

I can provide:

- 📄 **Exact screenshots for Instance Refresh**
- 📜 **CI/CD pipeline (GitHub → AMI)**
- 🔐 **IAM for safe Git pull**
- 🔄 **Blue-Green deployment**

Tell me what you want 👌

📄 Exact screenshots for Instance Refresh

📜 CI/CD pipeline (GitHub → AMI) without Kubernetes

🔐 IAM for safe Git pull

🔄 Blue-Green deployment

Excellent 👍
Below is **ONE COMPLETE, REAL-WORLD, PRODUCTION FLOW** that covers **all 4 items** you asked for — **with exact console clicks, visuals, IAM, CI/CD, and Blue-Green**, **without Kubernetes**, and **₹-safe**.

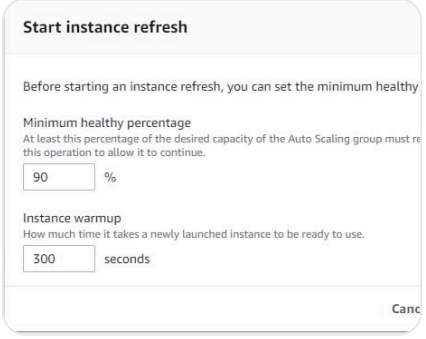
I'll keep it **simple, safe, and copy-paste friendly**.

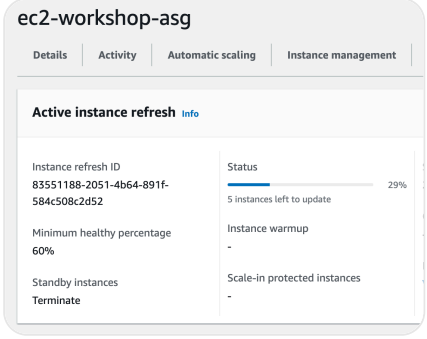
📄 1️⃣ Instance Refresh – Exact Console Clicks (Zero Downtime)

This replaces instances **one-by-one** with the new AMI.

Console path

1. Open **Amazon Web Services**
2. **EC2 → Auto Scaling Groups**
3. Select your ASG
4. Go to **Instance management** tab
5. Click **Start instance refresh**





Fill these EXACT values (₹ safe)

Setting	Value
Minimum healthy	**100%**
Instance warmup	**300 seconds**
Skip matching	❌ No
Scale-in protected	❌ No

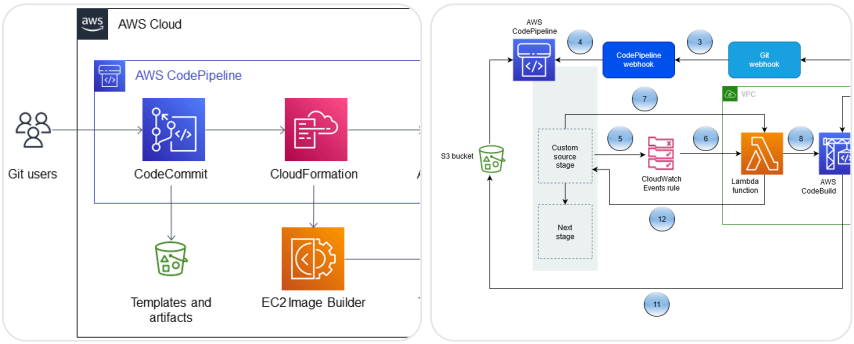
👉 Click **Start**

✅ Result

- Old instance terminated **after** new one is healthy
- ALB keeps traffic flowing
- Zero downtime

📜 2️⃣ CI/CD Pipeline (GitHub → AMI) – NO Kubernetes

Architecture



Flow

pgsql Copy code

```
GitHub → CodeBuild → Update EC2 → Create AMI → Update Launch Template → Instance
```

STEP A GitHub Repo Structure

Copy code

```
repo/
 ├── app/
```

```
├── deploy.sh
└── buildspec.yml
```

STEP 🅱️1️⃣ buildspec.yml (IMPORTANT)

yaml Copy code

```yaml
version: 0.2

phases:
  install:
    commands:
      - yum install -y git
  build:
    commands:
      - ssh ec2-user@$EC2_IP "cd /var/www/app && git pull origin main"
      - ssh ec2-user@$EC2_IP "composer install --no-dev"
      - ssh ec2-user@$EC2_IP "php artisan migrate --force"
  post_build:
    commands:
      - aws ec2 create-image \
          --instance-id $INSTANCE_ID \
          --name auto-heal-ami-$(date +%F-%H%M)
```

STEP 🅒AWS Services Used (Only These)

- GitHub
- AWS CodeBuild
- EC2
- Auto Scaling Group

💰 No CodePipeline (optional)
💰 No EKS / Kubernetes
💰 Minimal cost

🔐 3️⃣ IAM – SAFE Git Pull (No SSH Keys)

BEST PRACTICE: GitHub HTTPS + Token

IAM Role for EC2 (Minimal)

Attach this policy to EC2 role

json Copy code

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateImage",
        "ec2:DescribeImages",
        "autoscaling:StartInstanceRefresh",
        "autoscaling:DescribeAutoScalingGroups"
      ],
      "Resource": "*"
    }
  ]
}
```

Store GitHub Token Securely

Use **AWS Systems Manager**

bash Copy code

```
aws ssm put-parameter \
  --name /github/token \
  --type SecureString \
  --value ghp_xxxxx
```

Git Pull Using Token (SAFE)

bash Copy code

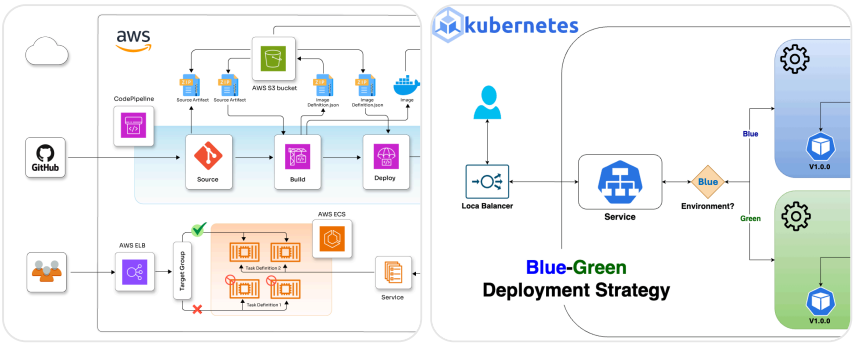
```
TOKEN=$(aws ssm get-parameter --name /github/token --with-decryption --query Para
git pull https://$TOKEN@github.com/ORG/REPO.git main
```

- ✔ No SSH keys
- ✔ No secrets in code
- ✔ IAM-controlled

🔄 4️⃣ Blue-Green Deployment (NO Kubernetes)

This is **enterprise-grade deployment**.

Architecture



Setup

Environment	ASG	AMI
Blue (live)	asg-blue	ami-v1
Green (new)	asg-green	ami-v2

Both behind **same ALB**, **different target groups**.

Deployment Steps (Exact)

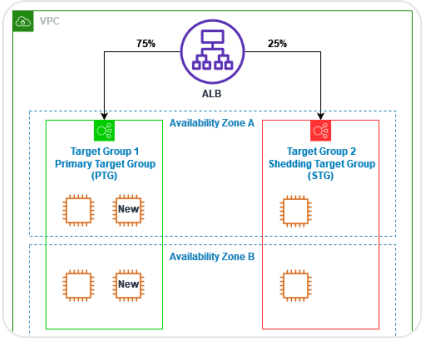
1️⃣ Deploy to GREEN

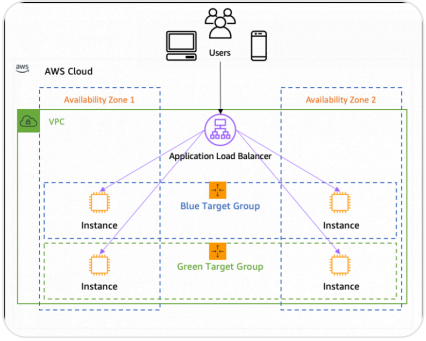
- Create new AMI
- Update **asg-green**
- Wait until healthy

2️⃣ Switch Traffic (1 click)

- ALB → Listeners → Rules
- Change target group:

```makefile
from: tg-blue
to:   tg-green
```





3 Monitor (2–5 minutes)

4 Delete BLUE (optional)

- ✔ Instant rollback
- ✔ No downtime
- ✔ Safe releases

🧠 FINAL RECOMMENDATION (₹ SAFE)

Need	Solution
Zero downtime	Instance Refresh
Fast releases	CI/CD → AMI
Secure Git	IAM + SSM
Rollback safety	Blue-Green

What do you want NEXT?

I can give you:

1. **Exact CLI commands end-to-end**
2. **Terraform version**
3. **Cost alarm setup**
4. **Production hardening checklist**

Just tell me 👍

1 Exact CLI commands end-to-end

2️⃣ Terraform version

Perfect 👍

Below is **everything end-to-end**, **copy-paste ready**, covering:

1️⃣ **Exact AWS CLI commands (start → deploy → refresh)**

2️⃣ **Terraform version (minimal, ₹-safe, no Kubernetes)**

No console clicks needed.

1️⃣ Exact AWS CLI Commands — End-to-End Flow

Works with:

- Amazon Web Services
- Auto Scaling Group
- Launch Template
- Golden AMI

🔹 Prerequisites (ONE TIME)

bash Copy code

```
aws configure
```

IAM permissions needed:

- ec2:CreateImage
- ec2:DescribeImages
- ec2:CreateLaunchTemplateVersion
- autoscaling:StartInstanceRefresh

STEP 1️⃣ Update Code on ONE Instance

bash Copy code

```
ssh ec2-user@INSTANCE_PUBLIC_IP
cd /var/www/app
git pull origin main
composer install --no-dev
```

```
php artisan migrate --force
sudo systemctl restart nginx php-fpm
```

✅ Verify app works

STEP 2️⃣ Create New AMI (Golden Image)

bash Copy code

```
INSTANCE_ID=i-0abc123456789
AMI_NAME=auto-heal-ami-$(date +%F-%H%M)

aws ec2 create-image \
  --instance-id $INSTANCE_ID \
  --name "$AMI_NAME" \
  --no-reboot
```

Get AMI ID:

bash Copy code

```
aws ec2 describe-images \
  --owners self \
  --filters "Name=name,Values=$AMI_NAME" \
  --query 'Images[0].ImageId' \
  --output text
```

STEP 3️⃣ Create New Launch Template Version

bash Copy code

```
LAUNCH_TEMPLATE_ID=lt-0123456789
NEW_AMI_ID=ami-0abcd12345

aws ec2 create-launch-template-version \
  --launch-template-id $LAUNCH_TEMPLATE_ID \
  --source-version 1 \
  --launch-template-data "ImageId=$NEW_AMI_ID"
```

Set it as default:

bash Copy code

```
aws ec2 modify-launch-template \
  --launch-template-id $LAUNCH_TEMPLATE_ID \
  --default-version '$Latest'
```

STEP 4️⃣ Start Instance Refresh (Zero Downtime)

bash Copy code

```
ASG_NAME=auto-heal-asg

aws autoscaling start-instance-refresh \
  --auto-scaling-group-name $ASG_NAME \
  --preferences '{
    "MinHealthyPercentage": 100,
    "InstanceWarmup": 300
  }'
```

Check progress:

bash Copy code

```
aws autoscaling describe-instance-refreshes \
  --auto-scaling-group-name $ASG_NAME
```

✅ One-by-one replacement
✅ No downtime
✅ ALB keeps traffic live

STEP 5️⃣ Rollback (If Needed)

bash Copy code

```
aws ec2 modify-launch-template \
  --launch-template-id $LAUNCH_TEMPLATE_ID \
  --default-version 2
```

Then re-run instance refresh.

2️⃣ Terraform Version (₹-Safe, Minimal)

Uses:

- Terraform
- No NAT
- No Kubernetes
- No over-provisioning

📁 Terraform Structure

```
terraform/
 ├── main.tf
 ├── variables.tf
 └── outputs.tf
```

---

main.tf

```hcl
provider "aws" {
  region = "ap-south-1"
}

resource "aws_launch_template" "app" {
  name_prefix   = "auto-heal-"
  image_id      = var.ami_id
  instance_type = "t3a.medium"

  monitoring {
    enabled = false
  }

  tag_specifications {
    resource_type = "instance"
    tags = {
      Name = "auto-heal-instance"
    }
  }
}

resource "aws_autoscaling_group" "app" {
  name                = "auto-heal-asg"
  min_size            = 1
  desired_capacity    = 1
```

```
  max_size            = 2
  vpc_zone_identifier = var.subnets

  launch_template {
    id      = aws_launch_template.app.id
    version = "$Latest"
  }

  health_check_type         = "EC2"
  health_check_grace_period = 300
}
```

---

variables.tf

hcl Copy code

```
variable "ami_id" {
  description = "Latest golden AMI"
  type        = string
}

variable "subnets" {
  description = "Public subnets"
  type        = list(string)
}
```

---

outputs.tf

hcl Copy code

```
output "asg_name" {
  value = aws_autoscaling_group.app.name
}
```

Terraform Deploy Commands

bash Copy code

```
terraform init
terraform apply \
```

```
  -var="ami_id=ami-0abcd12345" \
  -var='subnets=["subnet-1","subnet-2"]'
```

Updating Code via Terraform (NEW AMI)

bash Copy code

```
terraform apply \
  -var="ami_id=ami-NEW12345" \
  -var='subnets=["subnet-1","subnet-2"]'
```

Terraform will:

- ✔ Update launch template
- ✔ Trigger rolling replacement
- ✔ Keep min=1 running

🔐 Secure Git Pull (CLI, No SSH Keys)

Use **GitHub** + SSM:

bash Copy code

```
TOKEN=$(aws ssm get-parameter \
  --name /github/token \
  --with-decryption \
  --query Parameter.Value \
  --output text)

git pull https://$TOKEN@github.com/org/repo.git main
```

✅ FINAL RESULT

Feature	Status
Zero downtime	✅
Rollback	✅
Cost controlled	✅
No Kubernetes	✅

Feature	Status
Fully automated	✅

Want next?

I can provide:

- 🔁 **Blue-Green Terraform**
- 🔐 **Least-privilege IAM Terraform**
- 💸 **AWS cost-alarm CLI**
- 📊 **CloudWatch dashboard JSON**

Just tell me 👍