**Project 1: Monolithic App Migration to AWS Cloud**

**Scenario Brief – Real-World Context**

**Client:**

**Skymet Tracker** is a small but growing startup that runs a weather tracking and forecast app. Their current app is a **Python (Flask)** monolith backed by a **PostgreSQL** database, served from a single Linux VM hosted on-premises. The frontend is built with basic HTML and CSS.

**Challenges They’re Facing:**

* Frequent **downtime** during sudden weather spikes
* **Manual scaling** and configuration
* **No backup or recovery mechanism**
* **No CI/CD** for deployments
* **No centralized monitoring**
* High risk of **data loss or service failure**

**Budget Constraints:**

The team is operating on a **tight budget**, aiming to stay within the **AWS Free Tier**, or as close to it as possible.

**My Role as the Solution Architect**

* **As the Solution Architect on this project, I was responsible for:**
* **Analysing the existing monolithic application and its infrastructure**
* **Identifying key performance, scalability, and operational pain points**
* **Designing a secure, cost-effective, and scalable architecture on AWS aligned with the client’s current needs and future goals**
* **Defining a phased roadmap that supports future enhancements like auto-scaling, CI/CD integration, monitoring, and disaster recovery**

**Discovery Phase**

**1. Project Overview**

| **Item** | **Details** |
| --- | --- |
| **Client Name** | Skymet Tracker (fictional) |
| **Industry** | Weather Tracking & Analytics |
| **Current Infra** | On-prem Linux VM with Flask app + PostgreSQL |
| **Business Goal** | Migrate to AWS Cloud for uptime, scalability & future readiness |
| **Budget** | AWS Free Tier or low-cost solution |

**2. Business Understanding**

“Our weather forecast app is growing, and users depend on it — especially during climate events. But our current server setup is fragile. We experience outages during user spikes, and we don’t have any monitoring or backup system.

We want to move to AWS, but we’re new to cloud and need help designing something reliable and cost-effective.”

**From this, my key takeaways are:**

* Focus on **stability and high availability**
* Use AWS services that fall within **Free Tier limits**
* Keep the architecture **simple and manageable** for their small team
* Leave room to **improve over time** (e.g., add autoscaling, CI/CD, monitoring)

**3. Solution Goals (Mapped to Cloud Strategy)**

| **Business Need** | **My Architecture Focus** |
| --- | --- |
| 99.9% uptime | Design for minimal single points of failure (e.g., separate DB tier, health checks) |
| Cost-effective | Use Free Tier-eligible services and avoid unnecessary complexity |
| Ready to scale | Use modular setup (EC2 now, containerization later) |
| Easy to maintain | Use standard AWS tooling, IAM roles, and tagging for clarity |
| Secure by design | Use VPC, security groups, and IAM best practices from Day 1 |

**4. Technical Discovery – Simulated Client Interview**

Here’s a mock technical discussion I documented to better understand the client environment:

| **Question** | **Client’s Response** |
| --- | --- |
| What tech stack are you using? | Python (Flask), PostgreSQL, basic HTML |
| How is the app hosted? | On a single Linux VM, no orchestration |
| DB size and structure? | Around 1 GB, basic read/write ops |
| Average daily traffic? | ~10,000 users; spikes during bad weather |
| Any scheduled jobs or tasks? | Yes, an hourly cron job to fetch climate data |
| Any compliance concerns? | Not currently, but may consider GDPR |
| Backup policy? | None now — high-risk setup |
| Who maintains the infra? | Just the CTO and one developer with basic AWS knowledge |

**5. Summary of Discovery**

Skymet Tracker runs a simple but critical weather forecast app, fully hosted on a single server. As user demand increases, their system becomes unstable. With no backup, no CI/CD, and no monitoring, the app is vulnerable to both performance and reliability risks.

**Architectural Intent:**

I aim to design a **minimal but production-grade AWS setup** that:

* Works within Free Tier
* Separates app and database layers
* Enables future scaling (load balancing, CI/CD, caching)
* Improves security and observability

This will be **Phase 1** of a multi-phase roadmap, where the first goal is **cloud migration and stabilization**.

# **High-Level Architecture Design Phase**

**🎯 Design Goals**

This architecture is designed for:

* Zero-downtime AWS migration from on-premises
* Cost-effective, Free Tier–friendly setup
* A modular and secure foundation for future scaling
* Minimal DevOps load — easy to manage for a small team

**Component Breakdown (What Each Service Does)**

| **Component** | **Description & Role** |
| --- | --- |
| **VPC (Virtual Private Cloud)** | Isolated AWS network that contains all your resources. Ensures your app is separated from other tenants in the cloud. |
| **Public Subnet** | Hosts the EC2 instance (Flask app) and Internet Gateway. Resources here can communicate with the internet. |
| **Private Subnet** | Hosts the RDS PostgreSQL database. It's isolated from the internet for security reasons. |
| **Internet Gateway** | Enables internet access for the EC2 instance in the public subnet. |
| **EC2 Instance (Flask App)** | Runs the monolithic Python Flask application. Handles user requests and app logic. |
| **RDS PostgreSQL** | Managed relational database service for app data storage. Secure, reliable, and backed up (optional in later phases). |
| **S3 (Static Files)** | Stores static assets like images, CSS, JS, and optionally app logs or backups. Accessible by EC2 and external users. |
| **CloudFront (CDN)** | Optional edge caching layer that speeds up static content delivery (from S3 or EC2) globally. |
| **Route 53** | DNS service that maps domain names to CloudFront or EC2 IP. Helps access app via a custom domain (e.g., skymetapp.com). |
| **IAM Role** | Manages secure access between services (e.g., EC2 ↔ S3, EC2 ↔ CloudWatch). Uses least-privilege policy. |
| **CloudWatch** | Monitors resource usage (CPU, memory, I/O) and collects logs for EC2 and RDS. Alarms can be configured in later phases. |

**Data & Traffic Flow**

**Here’s how user requests and internal communication happen:**

1. **User → Route 53:  
   User accesses the app via a domain name like weather.skymetapp.com which is routed through Route 53.**
2. **Route 53 → CloudFront (optional):  
   Domain routes traffic to CloudFront, which caches static content for faster global delivery (can forward dynamically requests to EC2 too).**
3. **CloudFront → EC2 (Flask App):  
   CloudFront forwards dynamic content requests to the EC2 instance running Flask (if configured). Alternatively, users can directly hit EC2 via public IP (initial phase).**
4. **EC2 → RDS PostgreSQL (Private Subnet):  
   The Flask app talks to the database over a private network within the VPC. No public access to RDS is allowed.**
5. **EC2 ↔ S3 (Static Files):  
   EC2 uploads and reads static files (e.g., weather map images) from the S3 bucket via IAM role permissions.**
6. **External Users → S3 (via CloudFront or direct link):  
   Static assets like CSS/JS/images are served directly to users via S3 (or via CloudFront if configured).**
7. **EC2 & RDS → CloudWatch:  
   Both services push logs and metrics to CloudWatch for centralized monitoring and future alarm configuration.**
8. **IAM Roles:  
   IAM roles are attached to EC2 and other services to securely allow access to S3, CloudWatch, or future integrations — no hardcoded keys.**

**Architecture Diagram**

**A computer screen shot of a computer

AI-generated content may be incorrect.**

**Estimated AWS Cost & Billing Strategy:**

**Monthly Cost Breakdown Table**

| **Service** | **Purpose** | **Tier** | **Estimated Monthly Cost** |
| --- | --- | --- | --- |
| **EC2 (t2.micro)** | **App server** | **Free Tier** | **$0** |
| **RDS (PostgreSQL)** | **DB** | **Free Tier (750 hrs)** | **$0** |
| **S3** | **Static files** | **Free up to 5GB** | **$0** |
| **CloudFront** | **CDN** | **~1TB/month free** | **$0** |
| **Route 53** | **Domain & DNS** | **$0.50–$1/month** | **~$1** |
| **CloudWatch** | **Logs & metrics** | **Within Free Tier** | **$0** |
| **Total (est.)** |  |  | **$1–2/month** |

**Note: This estimation assumes Free Tier eligibility. In production or multi-team setups, Route 53 and CloudWatch costs may scale depending on traffic and data retention.**

# **Implementation & Deployment (Initial Infra Setup)**

**Objectives**

* Provision AWS resources based on the Phase 2 design
* Deploy the Flask app to EC2
* Ensure the EC2 instance connects to RDS securely
* Setup S3, IAM roles, and basic CloudWatch logging
* Stay within AWS Free Tier (or close)

**Components to Build**

| **Component** | **Task** |
| --- | --- |
| **VPC + Subnets** | Create 1 VPC with 2 subnets (public + private) |
| **Internet Gateway** | Attach to public subnet for EC2 internet access |
| **Route Tables** | Setup route from public subnet to IGW |
| **Security Groups** | Allow HTTP (80) and SSH (22) to EC2, DB access from EC2 only |
| **EC2 Instance** | Launch t2.micro instance, install Flask app |
| **RDS PostgreSQL** | Create DB in private subnet, no public access |
| **S3 Bucket** | Store static files, optionally app logs |
| **IAM Roles** | Assign to EC2 for S3 + CloudWatch access |
| **CloudWatch** | Enable logs and metrics for EC2 and RDS |

**Step-by-Step Tasks**

**1. Networking Setup (VPC)**

* VPC: 10.0.0.0/16
* Public Subnet: 10.0.1.0/24
* Private Subnet: 10.0.2.0/24
* Internet Gateway: Attach to public subnet
* Route Tables: Route 0.0.0.0/0 to IGW for public subnet

**Tools:** AWS Console (manual) or Terraform (automated)

**2. Security Groups**

* **EC2 SG:**
  + Inbound: HTTP (80), SSH (22) from your IP only
  + Outbound: Allow all
* **RDS SG:**
  + Inbound: PostgreSQL (5432) only from EC2 SG

**3. EC2 Instance Setup**

* t2.micro (Free Tier)
* Amazon Linux 2 or Ubuntu
* Install:

sudo yum update -y

sudo yum install python3 git -y

git clone <your-repo>

pip3 install -r requirements.txt

python3 app.py

* Configure EC2 to use IAM Role for S3/CloudWatch access

**4. RDS PostgreSQL Setup**

* PostgreSQL, db.t3.micro (Free Tier eligible)
* Subnet Group: Only private subnets
* DB Name, User, Password: Save in EC2 env vars or AWS Parameter Store
* Security: No public access, access allowed from EC2 SG only

**5. S3 Bucket Setup**

* Create a bucket (e.g., skymet-static-assets)
* Enable static website hosting (optional)
* Set permissions:
  + Public read for static files (images/CSS)
  + EC2 write access via IAM

**6. IAM Role Creation**

* **EC2 Role Policy:**
  + AmazonS3ReadOnlyAccess
  + CloudWatchAgentServerPolicy
* Attach this IAM role to EC2 instance

**7. CloudWatch Monitoring**

* Enable:
  + Basic EC2 monitoring
  + RDS enhanced monitoring (optional)
* Install CloudWatch agent on EC2:

sudo yum install amazon-cloudwatch-agent

**Test After Deployment**

* App is reachable via EC2 public IP or domain
* Flask app can query RDS (check DB read/write)
* Static files load from S3
* Logs appear in CloudWatch
* No access to RDS from public internet

**Terraform Plan Execution (No Cost Applied):**   
Since this is a portfolio project and AWS billing is a concern, I executed terraform plan only.  
This verified all modules (vpc, ec2, rds, s3, iam, cloudfront, route53) are syntactically and logically correct.

Below is the Terraform Plan output screenshot showing infrastructure is ready to be provisioned on AWS.

