

# API Health Monitoring System

## 1) Problem Statement

Teams often rely on a few critical APIs and need to know quickly when one goes down. This project builds a self-hosted monitoring system that checks API endpoints at a fixed interval, detects meaningful state changes, and notifies users—without using managed monitoring services. The goal is not to feature completeness but clear system design, reliability thinking, and solid infrastructure choices.

## 2) High-Level Architecture

The system is intentionally simple and AWS-native:

EC2 instance runs the Python monitoring app on a cron schedule.

DynamoDB stores:

- API configuration (api\_health\_configs)
- Last known health state (api\_health\_states)
- SNS delivers email alerts on state changes.
- IAM role on EC2 provides least-privilege access to DynamoDB + SNS.

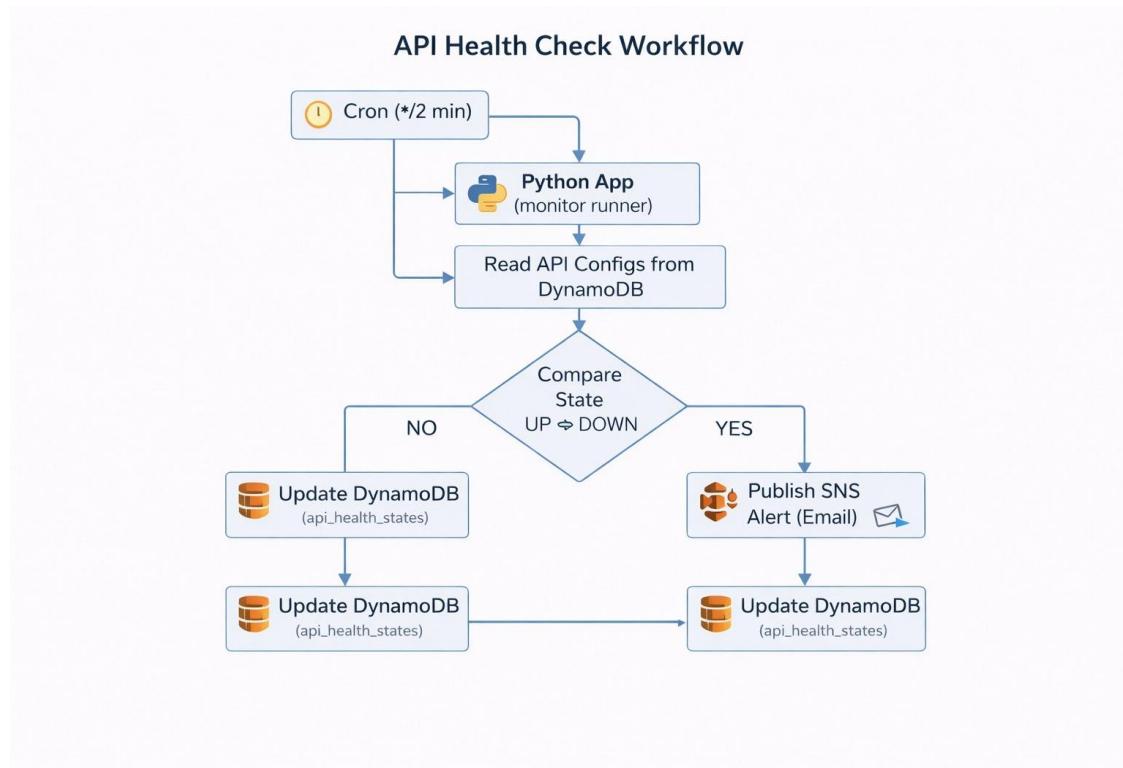


Fig1.API Health Check Work-flow

### **3) Component Interactions & Data Flow**

Health Check Flow (per run)

- Cron triggers the Python app every 2 minutes.
- App reads all API configs from DynamoDB.
- Each API is checked using requests with timeout + expected status rules.
- The result is compared with the last stored state.
- If the state changed (UP ↔ DOWN), an SNS alert is published.
- The new state is saved back to DynamoDB.

### **4) Infrastructure & Deployment (Terraform)**

Infrastructure is defined in infra/terraform:

- EC2: runs the cron + Python app
- DynamoDB: api\_health\_configs, api\_health\_states
- SNS: alert topic
- IAM: least-privilege policy attached to EC2 role
- Outputs: EC2 public IP, table names, SNS topic ARN

Deployment Steps:

1. `terraform init / plan / apply`
2. SSH to the EC2 instance
3. `pip install -r app/requirements.txt`

Set env vars:

- `AWS_REGION, APP_ENVIRONMENT`
- `DDB_CONFIG_TABLE, DDB_STATE_TABLE`
- `SNS_TOPIC_ARN`
- Configure cron to run `python3 -m app.src.cron_entrypoint`

### **5) Code Structure**

- `app/src`
- `config.py`: loads env config
- `models.py`: dataclasses for config + state
- `dynamodb_client.py`: scan configs, get/update states
- `sns_client.py`: publish alerts
- `health_checker.py`: HTTP request + status/latency/error logic
- `monitor_runner.py`: orchestrates one run
- `cron_entrypoint.py`: cron entrypoint

- infra/terraform
- dynamodb.tf, sns.tf, iam.tf, ec2.tf, outputs.tf, etc.

## 6) Design Decisions, Assumptions, and Trade-offs

- Scheduling: Cron on EC2 is simple and transparent.
- Trade-off: if the instance is down, checks pause.
- Storage: DynamoDB on-demand avoids capacity planning.
- Trade-off: a full table scan doesn't scale forever.
- Alerting: SNS email is straightforward and cheap.
- Trade-off: limited customization.
- Logging: cron redirects to a local log file.
- Trade-off: not centralized.
- Assumptions: moderate number of APIs, per-minute checks, single AWS region.

## 7) Scalability & Reliability Considerations

- DynamoDB scales automatically for growth.
- Multiple EC2 workers could split work for larger fleets.
- Future scale could use a queue (e.g., SQS) with worker pools.
- State is stored persistently, so failures don't lose history.

## 8) Validation Steps

- Insert a test API config
- Run `python3 -m app.src.cron_entrypoint` once
- Verify a new record in `api_health_states`
- Flip to a failing URL and re-run to confirm SNS email alert

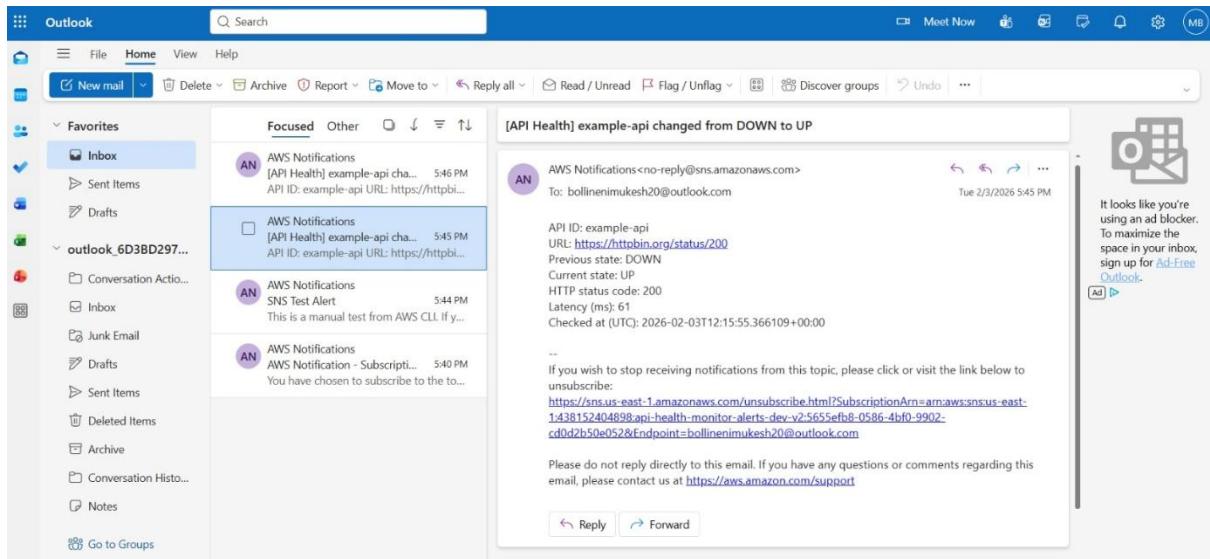


Fig2. API up email

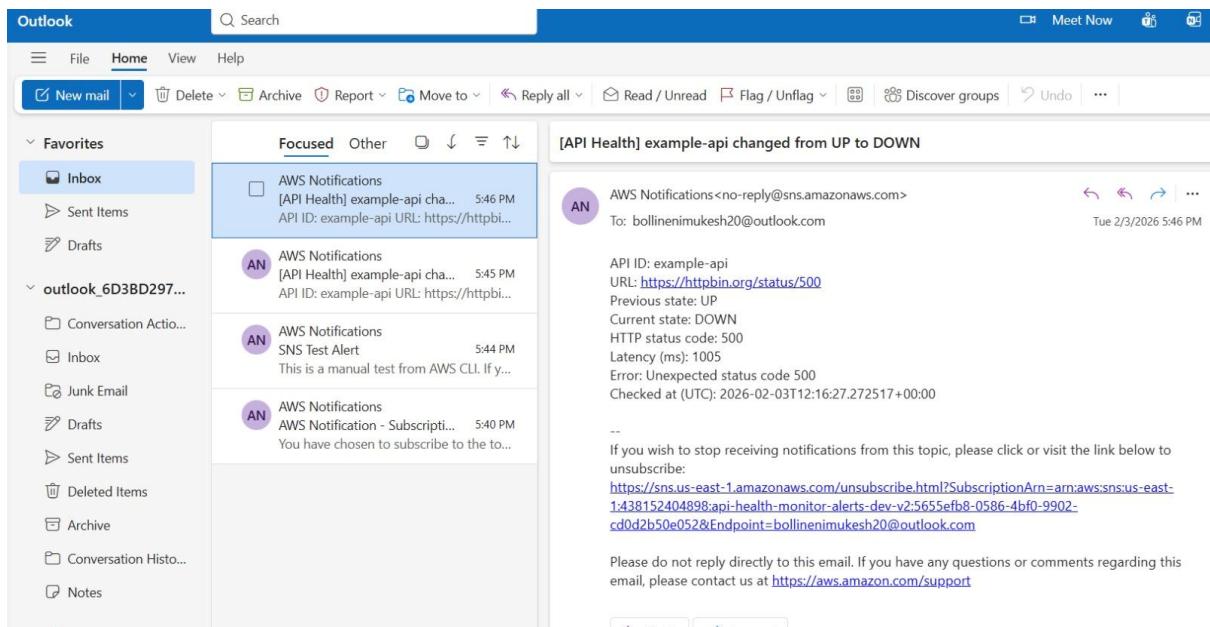


Fig3. API down email

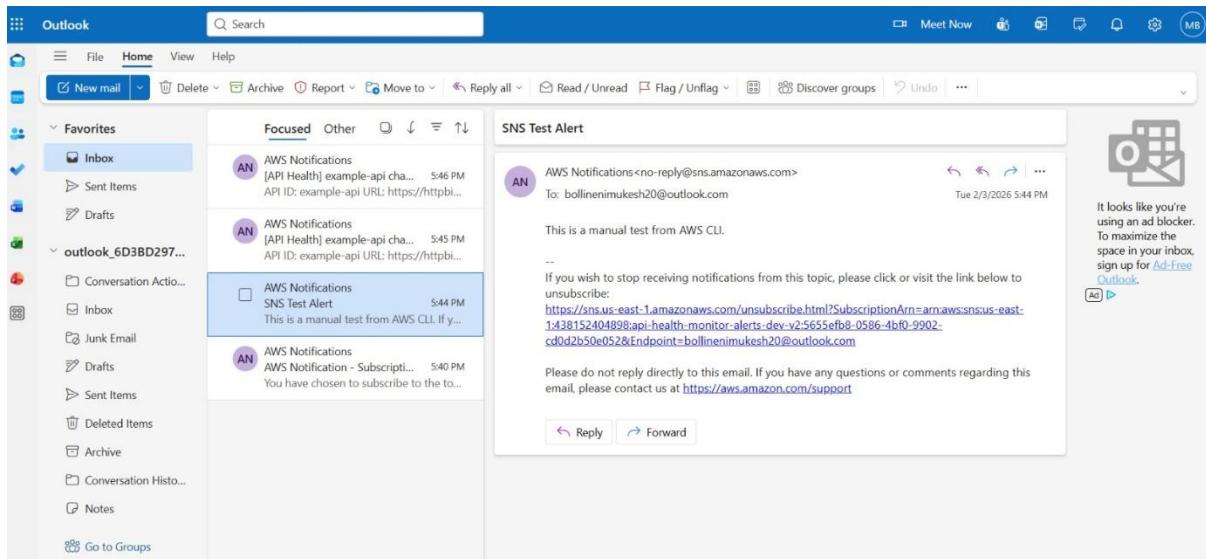


Fig4. API SNS alert mail

## 9) Diagram

The attached diagram shows:

High-level architecture

Data/alert flow

AWS logical layout

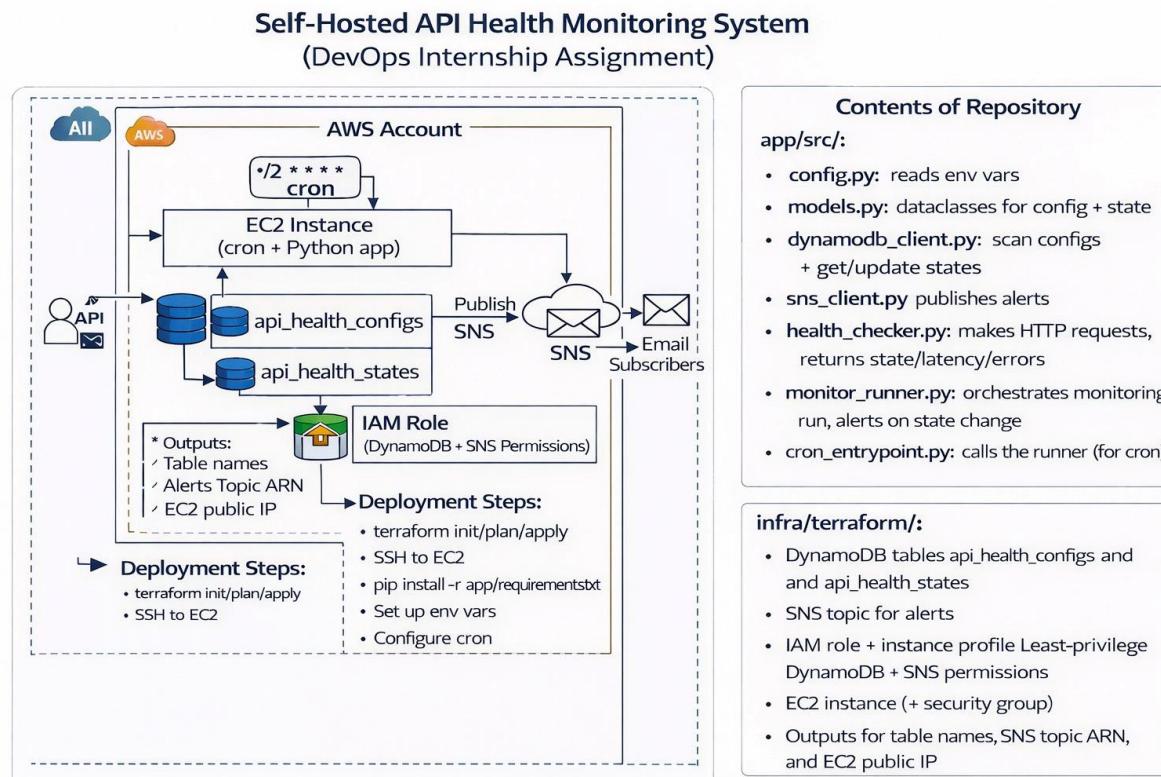


Fig5. Self-Hosted API Health Monitoring System – Architecture and Workflow