Edge-Monitor

Application Overview Edge Monitor

Edge Monitor is a lightweight, scalable system health monitoring solution designed for edge devices. It continuously collects system performance metrics such as CPU usage, memory utilization, disk space, (optionally) GPU status, and (optionally) temperature data, then transmits these metrics to a central server or cloud endpoint for analysis and visualization.

This repository contains a **Dockerized Full-Stack Application** with the following tech stack:

- **Edge Agent and API**: Python
- **Backend**: Spring Boot
- **API Gateway**: Spring Cloud Gateway
- **Mailing Micro-Service**: Spring Boot
- **Frontend**: Angular
- **Database**: MySQL
- **Caching**: Redis
- **Queueing**: Kafka
- **Deployment**: AWS EC2 instance

Core Components of Application

1. Edge Agent (Python Service)

- **Runs on each edge device.**
- **Collects metrics periodically (CPU, RAM, Disk, GPU, etc.)**
- **Uses python libraries for data collection (psutil and nvidia-ml-py3)**
- **Sends data to a configurable endpoint using HTTP (Fast Api and Aiohttp)**
- **Includes graceful shutdown handling and configurable intervals.**
- **Built with FastAPI for the server endpoint and asyncio for concurrent monitoring.**

2. Server API

- **A lightweight FastAPI application receives metrics from multiple devices.**
- **Stores or forwards incoming data to our Edge-Monitor Dashboards.**
- **Provides REST endpoints such as:**
    - **POST /ingest – Receives metrics from edge devices.**
    - **GET /metrics – Returns the latest metrics for visualization.**

3. Backend: Spring Boot

- **Used for creating APIs for login and Signup**
- **It stores data of logged In users in MySQL DB**
- **It creates JWT token for session management.**
- **It also publishes notification to Kafka.**

4. Gateway: Spring Boot

- **Used for additional security functionalities for Backend Apis**
- **It makes sure the Backen APIs are not exposed directly to clients**
- **It uses rate limiting and Throttling to prevent DDOS attacks**
- **It exposes APIS gateway to the Client application**

5. Mailing Micro-Service: Spring Boot

- **Used to provide Notification service to the application.**
- **It send user registration mails for now can be extended for providing notifications and alerts.**
- **This service is the client service for Kafka**

6. Frontend UI (Angular)

- **It provides login, signup functionalities, authentication and session management**
- **It also contains a simple web dashboard that visualizes system metrics in real-time.**
- **Auto-refreshes every few seconds.**
- **Displays CPU, memory, and temperature data using clean charts and cards.**

7. MySql : Used for maintaining User Info for Signin

8. Redis: Used for caching and session management.

9. Kafka: Used as a Messaging queue between Backend and Notification Service

Trade-offs Considered

Decision	Rationale	Trade-off
FastAPI over Flask	Async I/O, lightweight, high performance	Slightly higher complexity
HTTP transport	Simplicity, works with existing REST tools	Higher overhead vs MQTT
Poetry for dependency management	Reproducible, isolated builds	Slight learning curve
Dockerized	Simplifies deployment	Larger image size
Local metrics storage	Simplifies prototype	No persistence if container restarts
Rest API	For login and Signup, lightweight, high performance	Best for Microservice Architecture communications
Kafka	Decoupling of notification service	Bit more complex than Rest APIs
Angular	Structured Framework	Bit heavy and steep learning curve compared to React
EC2	For faster deployment and as its available in free Tire	AWS IoT Core or Greengrass better for real time

Scalability

1. Each Edge Device acts independently and sends periodic data, avoiding inter-device dependencies.
2. **The backend and all the Dockerized microservices can be horizontally scaled** (with multiple instances behind a load balancer).
3. As its **loosely coupled** some microservices **notification microservice** need no scaling as it typically will have less load,
4. Already used **Redis** for session management but can also **replace in-memory metrics** storage with Redis.
   Kafka for async message handling.
   Implement batch aggregation to reduce network overhead.

Security Considerations

1. Authentication & Authorization: Used JWT tokens for device-to-server communication.
2. Transport Security: Can extend HTTP communication to use TLS (HTTPS). Can also buy a domain and use AWS Certificate manager to generate SSL certificate.
3. Stored all the secrets in .env files so that its not pushed into the server. Also used git secrets for storing keys to deploy in EC2 instances
4. Data Exposure: Limit API responses to necessary metrics only. Avoid exposing system paths, IPs, or sensitive identifiers. Used Gateway so that clients can't access backend directly.
5. Container Security: Use minimal base images (python:3.11-slim) and run containers as non-root users.