Cloud Deployment Choices Impacting Performance of a Web Application Using Microservices Architecture

Weather Notification System

Swaifa Haque, Dhivya Mohan, Pegah Pourabdollah, Anila Satyavolu, Maaheen Yasin

1. System Design

Our system is designed to deliver timely weather-based notifications to subscribed users through a modular and scalable architecture. The system comprises three core services: User Subscription, Weather API Service, and Notification Service, which are orchestrated using Google Cloud Scheduler and supported by BigQuery for persistent data storage. The overall system architecture and interaction between components are illustrated in Figure 1.

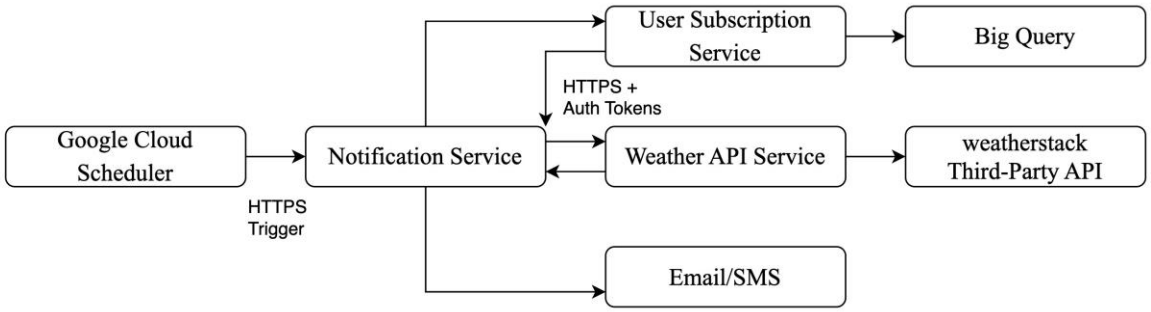

Figure 1. System architecture diagram showing microservice components and their interactions.

1.1 User Subscription

This component is responsible for registering users by collecting their details, including location and preferred mode of communication (Email or SMS). Once the user is subscribed, their information is securely stored in Google BigQuery, which serves as the central data repository for the system.

1.2 Weather API Service

The Weather API Service fetches real-time weather data using the external API provided by weatherstack.com. It queries weather information based on the location data retrieved from the user subscription records. All API calls are made via secure HTTPS connections with authorization tokens to ensure data privacy and secure communication.

1.3 Notification Service

The Notification Service is the central controller that sends timely alerts to users. Triggered by Google Cloud Scheduler at regular intervals, which fetches user data from the user subscription service and weather data from weather API, sends personalized alerts to users via SendGrid. This service communicates with both the micro services to collect the required user and weather data. Based on this data, it generates and sends notifications via Email or SMS. The status of each notification is logged and stored in BigQuery for further analysis and monitoring.

1.4 System Workflow

As shown in Figure 1, the system is triggered by Google Cloud Scheduler using an HTTPS call.

- The Notification Service then fetches user details from the User Subscription Service and weather data from the Weather API Service.
- The Weather API Service, in turn, fetches data from the weatherstack third-party API.
- The Notification Service processes this information and dispatches alerts to the users.
- Both user and notification data are stored in BigQuery for reporting and auditing purposes.

1.5 Design Choices Rationale

We chose this microservices architecture to ensure modularity, scalability, and ease of maintenance, allowing independent deployment and scalability. GCP was selected due to its integrated tools such as Cloud Scheduler and BigQuery, which simplify orchestration and data storage. BigQuery provides cost-efficient, serverless analytics suitable for our ETL and reporting needs. HTTPS-based inter-service communication offers simplicity and compatibility with external APIs like weatherstack.

2. Implementation

The system is implemented using a modular microservices architecture, with three primary services: User Subscription Service, Weather API Service, and Notification Service. Each service is built using Python Flask and exposes RESTful endpoints to enable synchronous communication via HTTPS. The User Subscription Service handles user inputs such as email, phone number, location, and notification preferences, and stores this data in Google BigQuery for storage and future analytics. The Weather API Service interacts with WeatherStack to retrieve real-time, historical, and forecast weather data. It provides multiple endpoints for location-based and date-based queries, with a dedicated helper function that ensures consistent API request formatting, error handling, and data extraction.

All services are containerized using Docker, which enables environment-independent builds. Each microservice is deployed on three different Google Cloud Platform (GCP) environments to compare performance across deployment models:

2.1 Cloud Run

The microservices were deployed on Cloud Run, a fully managed serverless platform on GCP. Container images were built with Docker and pushed to Google Container Registry. Cloud Run services automatically scale based on incoming HTTPS requests, offering a cost-efficient and hands-free deployment model.

2.2 Google Kubernetes Engine

The microservices architecture was deployed on Google Kubernetes Engine (GKE) using autopilot mode with horizontal pod autoscaling enabled. Docker images for the three microservices were built and pushed to Google Container Registry (GCR). Each pod was configured to use the relevant Docker image and environment variables. The pods were exposed to handle traffic using LoadBalancer services for external access, with ports and protocols configured accordingly. A replication factor of 3 was implemented to ensure high availability and fault tolerance. Once the cluster and pods reached a running state, deployment testing was performed using curl commands in Google Cloud Shell to verify service functionality and inter-service communication.

2.3 Google Compute Engine

We launched virtual machines with Dockers installed. Containers were run manually from the command line using Docker commands. This setup allowed direct control over the runtime environment and served as a baseline for performance comparison.

For secure inter-service communication, all services use Google-issued OIDC-compliant ID tokens (gcloud auth print-identity-token) passed as Bearer tokens over HTTPS. GCP Cloud Monitoring is used to capture logs, request latencies, and resource utilization, ensuring full observability across the system. Together, this implementation demonstrates a robust, cloud-native solution built with a unified technology stack including Python, Flask, Docker, BigQuery, SendGrid, WeatherStack, Cloud Scheduler, and Google Cloud deployment platforms, providing modularity, maintainability, and scalability.

3. Results and Measurements

By using Google Cloud Monitoring, we measured Latency and CPU Utilization to compare performance among the 3 deployment types. Figure 2 shows the results for CPU Utilization among Serverless, GKE Kubernetes, and VM implementations.

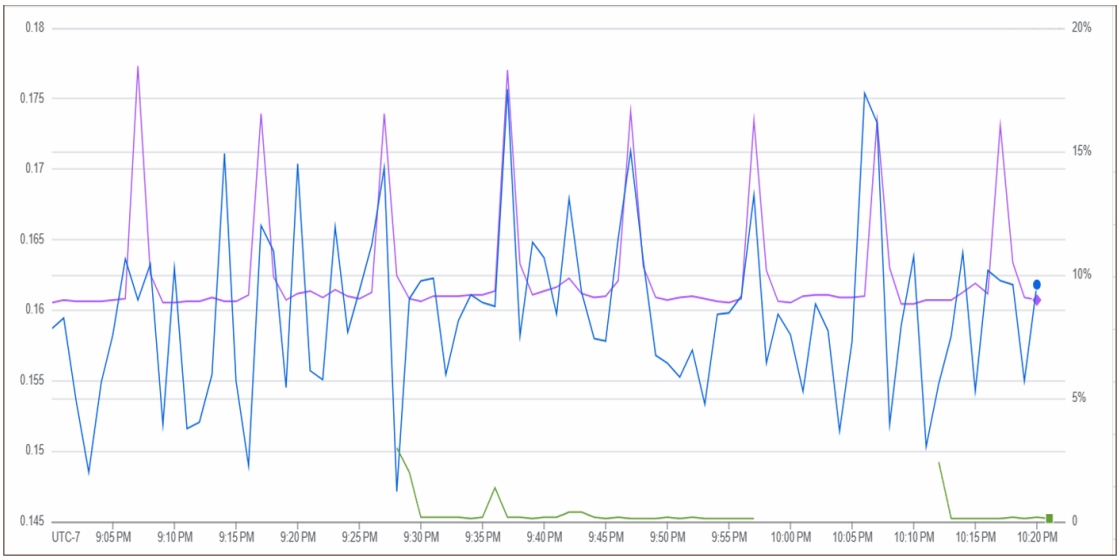

Figure 2. CPU Utilization among Serverless (Green), GKE Kubernetes (Blue) and VM (Purple).

4. Analysis of Results

4.1 Latency:

We observed that the request latency was 1.7s in Serverless, 4.1s in Kubernetes and 2.8s in VM, which clearly shows a good performance in Serverless.

4.2 CPU Utilization:

The CPU Utilization in serverless is incredibly low when seen against the other 2 deployments. On Serverless, as the setup is active only during an update such as adding a new user subscriber, we see the movement only during those times unlike on VM or on Kubernetes where the setup is active continuously. In conclusion, based on the observed performance metrics, serverless is a good deployment strategy for our project.

Link to our project repository WeatherNotificationService_Repository.