CS436 Term Project

-

Weather Data Aggregator

Technical Report

Group Members:

İsmet Ayvazoğlu 29493

Elif Şevval Kaya 28808

Mustafa Kulak 28912

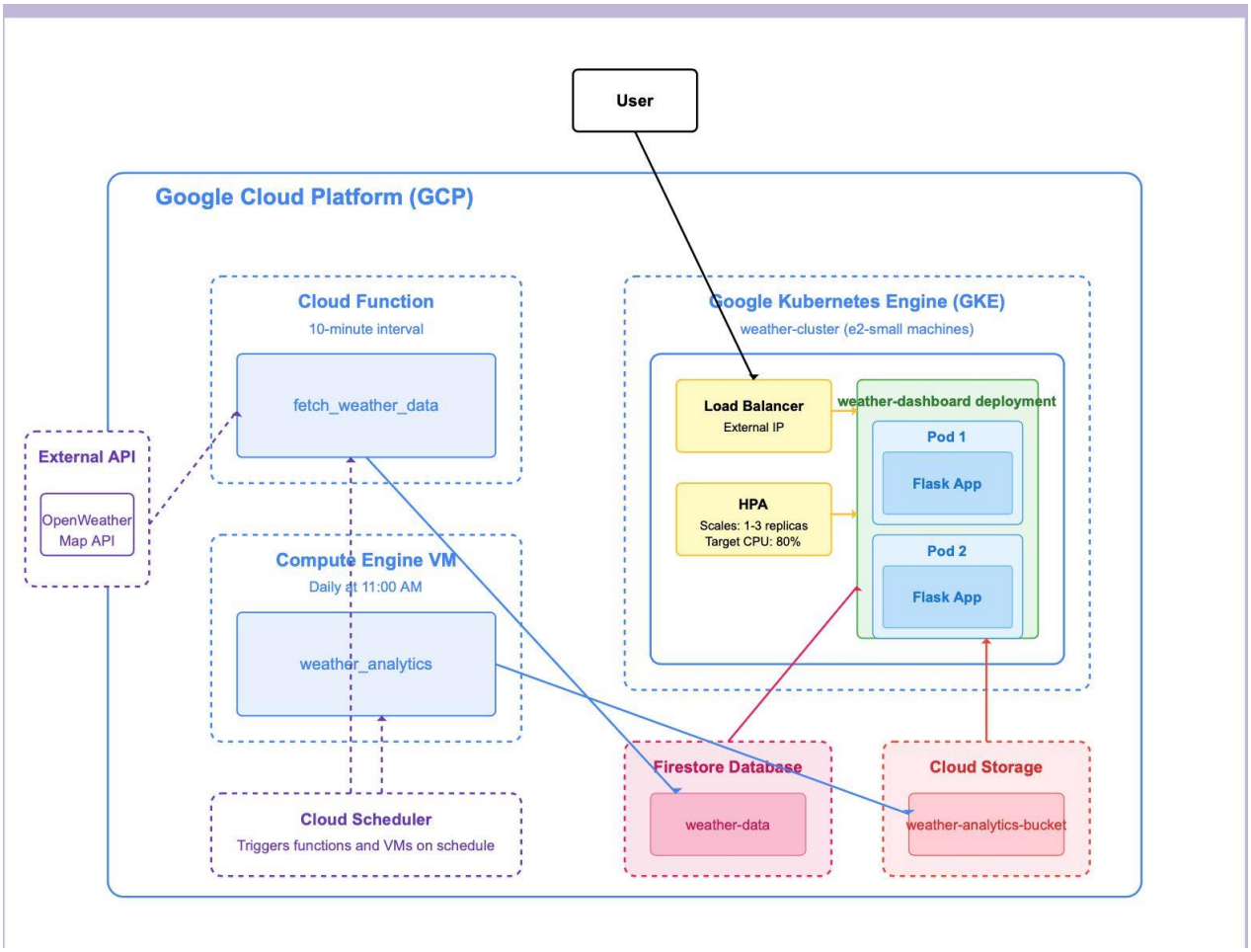
Date: 24.5.2025

Demo video URL:

https://drive.google.com/file/d/1oy08eDeFtLuwmLqd_vf3lMBG7O98Zez_/view?usp=sharing

1. Cloud Architecture

1.1. Cloud Architecture Diagram



The finalized architecture of the Weather Dashboard application represents a hybrid, cloud-native deployment on Google Cloud Platform (GCP), integrating serverless, virtual machine-based, and containerized services. It provides a clear delineation between external systems, scheduled internal processes, and interactive user components.

At the entry point, the user accesses the application via a web browser. Requests are routed through a Google Cloud Load Balancer, which distributes traffic across a Kubernetes-managed cluster hosted in Google Kubernetes Engine (GKE). Inside the GKE cluster, the weather-dashboard deployment runs as a Flask-based API server replicated across multiple pods. These pods are managed by a Horizontal Pod Autoscaler (HPA), which dynamically scales the number of replicas between 1 and 3 based on CPU usage, ensuring elasticity under varying load conditions.

On the serverless side, a Cloud Function named fetch_weather_data is triggered every 10 minutes to pull live weather data from the OpenWeatherMap API (an external, third-party service). This function parses the data and writes it to Firestore Database, a NoSQL serverless storage solution. In parallel, a Compute Engine VM named weather_analytics is scheduled to run daily at 11:00 AM via Cloud Scheduler. This VM executes Python-based batch processing scripts that generate historical and trend analyses using the data from Firestore. The results are then exported to Cloud Storage, making them accessible for frontend visualizations or long-term archiving.

Each service is bound together by well-defined communication flows:

- Data ingestion occurs from external APIs to GCP services.
- Scheduled workflows are orchestrated via Cloud Scheduler, triggering both Cloud Functions and VMs.
- Data flows from ingestion (Cloud Function) and analytics (VM) into Firestore and Cloud Storage.
- User-facing API requests are served by Flask apps inside the GKE pods, which pull data from Firestore and Cloud Storage, responding through the Load Balancer.

This multi-layered architecture ensures modularity, scalability, and observability, effectively balancing interactive user demands and backend data processing in a cost-effective and performant way.

1.2. Component Descriptions and Interactions

- User: Interacts with the weather dashboard interface through a browser, initiating data requests.
- Cloud Function (fetch_weather_data): Invoked every 10 minutes to fetch real-time weather data from the OpenWeatherMap API. It feeds data into Firestore and triggers additional processing.
- OpenWeatherMap API: A third-party weather data provider. Data is fetched from here via the Cloud Function.
- Compute Engine VM (weather_analytics): Scheduled daily at 11:00 AM via Cloud Scheduler. It performs batch analytics on historical weather data.
- Cloud Scheduler: Triggers both the Cloud Function and VM at configured intervals.
- Firestore Database: Stores raw and processed weather data. It is read by the dashboard frontend.
- Cloud Storage: Used for storing large analysis results and exporting datasets for dashboard visualizations.
- Google Kubernetes Engine (GKE): Hosts the weather-dashboard deployment, running the Flask-based API behind a load balancer.
- Horizontal Pod Autoscaler (HPA): Dynamically adjusts the number of pods based on CPU usage.
- Load Balancer: Routes incoming traffic to GKE pods.

1.3. Deployment Process

1. Cloud Function Setup: Deployed using GCP's gcloud functions deploy, connected to the external API.
2. Firestore and Cloud Storage: Initialized with access control and structure for weather data and analytics files.
3. VM Provisioning: Configured a Compute Engine instance and uploaded the Python analytics script.
4. Scheduler Configuration: Set time-based triggers using Cloud Scheduler for function and VM invocations.
5. Docker and Flask App: Dockerized the Flask API and deployed it to GKE using kubectl apply.
6. HPA and Load Balancer: Configured Horizontal Pod Autoscaler to scale between 1–3 replicas with 80% CPU target. Load Balancer auto-generated.

2. Locust Experiment Design and Parameter Configurations

The experiment was designed to evaluate system performance under various load conditions and to assess the impact of deployment configuration optimizations on application responsiveness and reliability.

2.1. Test Scenario Design

2.1.1. Load Test Scenarios

- Baseline Test Scenario: 50 users, 5 users/sec spawn rate, 15 minutes duration
  Purpose: Establish performance baseline
- Endurance Test Scenario: 80 users, 10 users/sec spawn rate, 60 minutes duration
  Purpose: Assess normal operating conditions
- Heavy Load Test Scenario: 200 users, 20 users/sec spawn rate, 15 minutes duration
  Purpose: Test system under high load
- Moderate Load Test Scenario: 100 users, 10 users/sec spawn rate, 15 minutes duration
  Purpose: Evaluate sudden traffic surge handling
- Spike Test Scenario: 300 users, 50 users/sec spawn rate, 5 minutes duration
  Purpose: Test sustained load performance

Locust was configured to hit five endpoints: /, /average-temperature, /current, /history, and /temperature-trend. Custom wait times and task weightings were used for realistic traffic.

2.1.2. Deployment Configurations Tested

Initial Configuration (initial-deployment.yaml):

- Replicas: 2 pods
- Resource Requests: 50m CPU, 64Mi memory

- Resource Limits: 200m CPU, 128Mi memory
- HPA Configuration: 1-3 replicas, 80% CPU utilization threshold
- Scaling Behavior: Default Kubernetes scaling policies

Optimized Configuration (deployment.yaml):

- Replicas: 3 pods (higher baseline)
- Resource Requests: 100m CPU, 128Mi memory
- Resource Limits: 500m CPU, 256Mi memory
- HPA Configuration: 3-6 replicas, 70% CPU utilization threshold
- Scaling Behavior:
    - Fast scale-up: 100% increase every 15 seconds
    - Controlled scale-down: 10% decrease every 60 seconds

2.1.3. Performance Metrics and Measurement

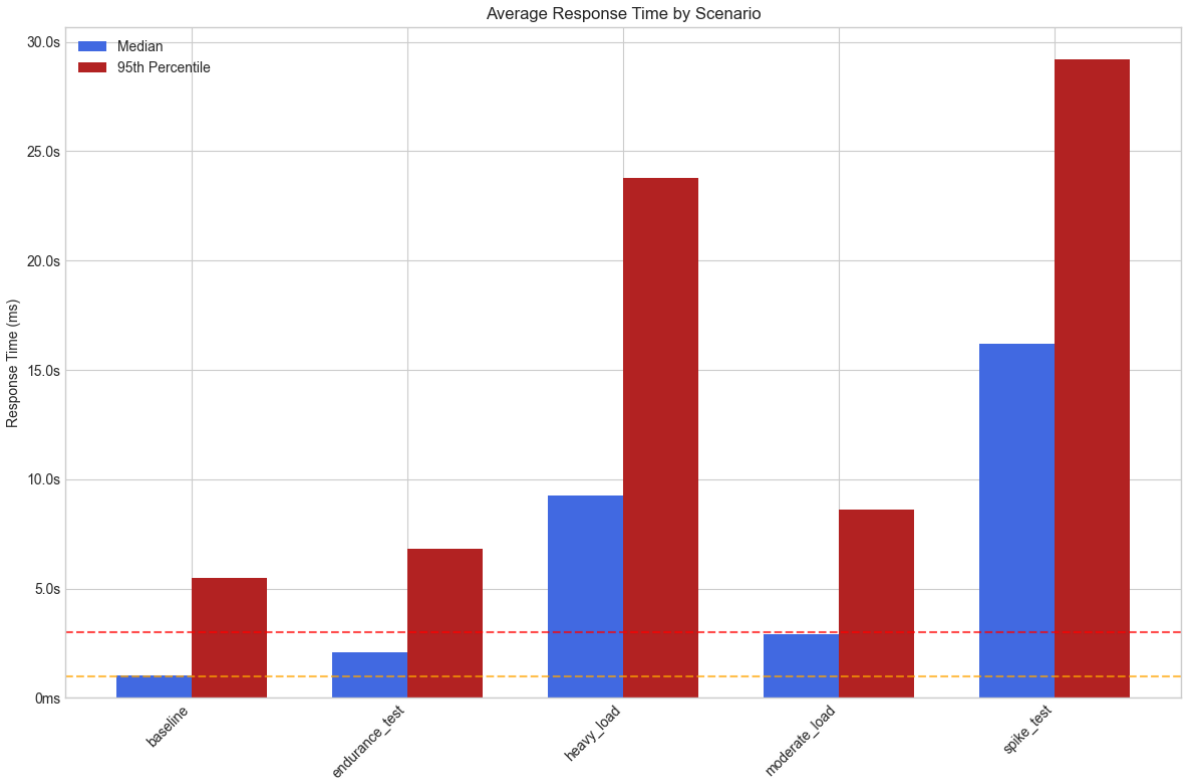
The experiment captured comprehensive performance metrics including:

- Response Time Metrics: Median, 95th percentile, average response times
- Throughput Metrics: Requests per second (RPS) for each endpoint
- Error Rates: Failure percentage across all scenarios
- Endpoint-Specific Performance: Individual API endpoint analysis

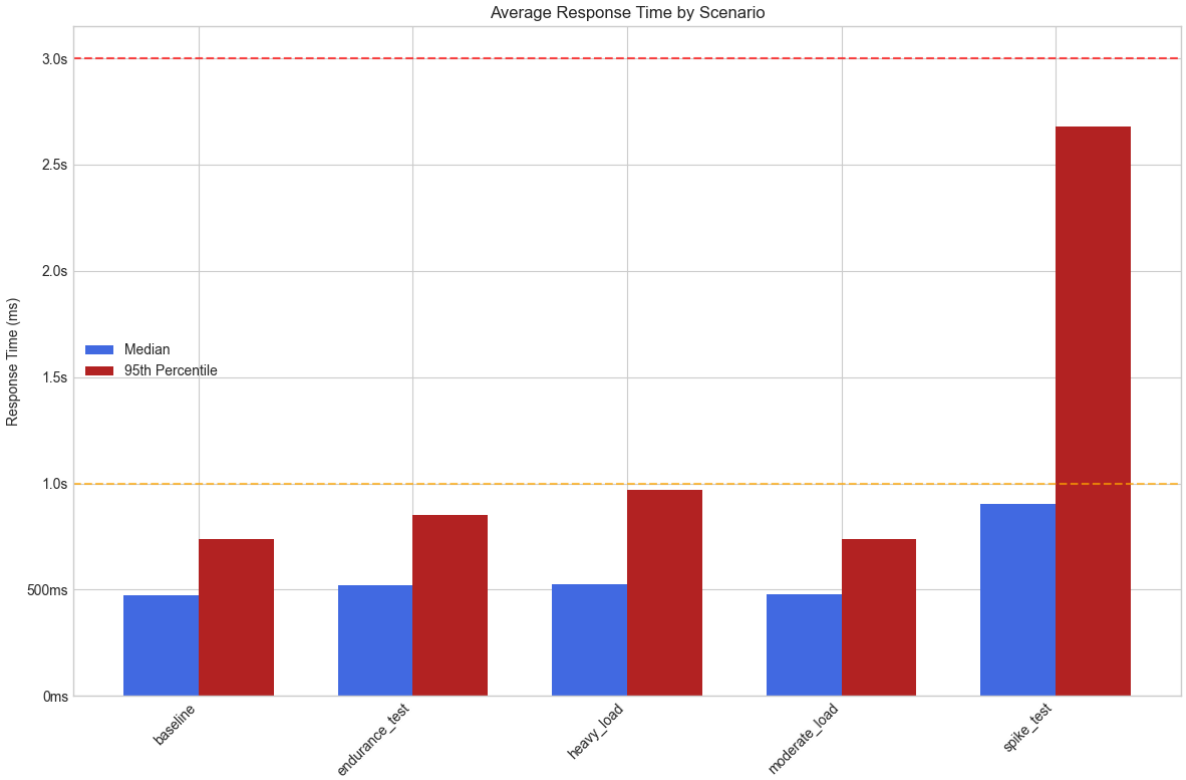
2.2 Visualized Performance Results

2.2.1 Response Time Analysis

Initial Deployment Results:



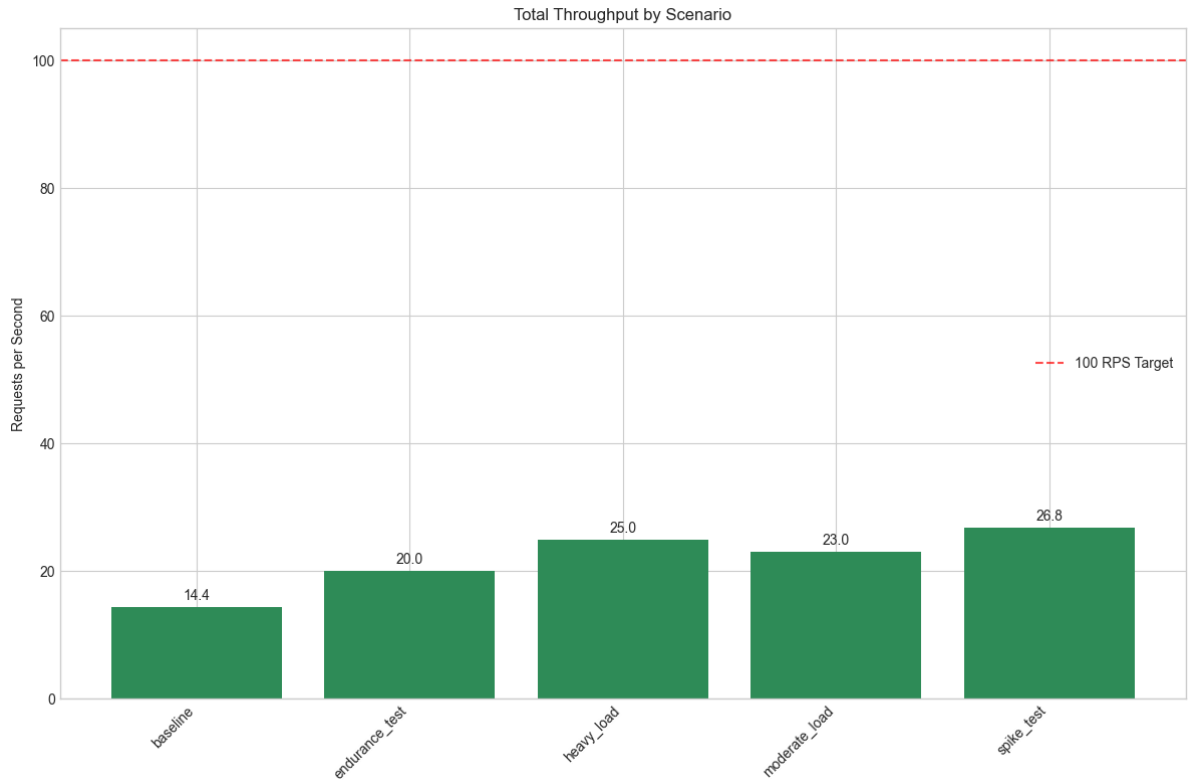
Optimized Deployment Results:



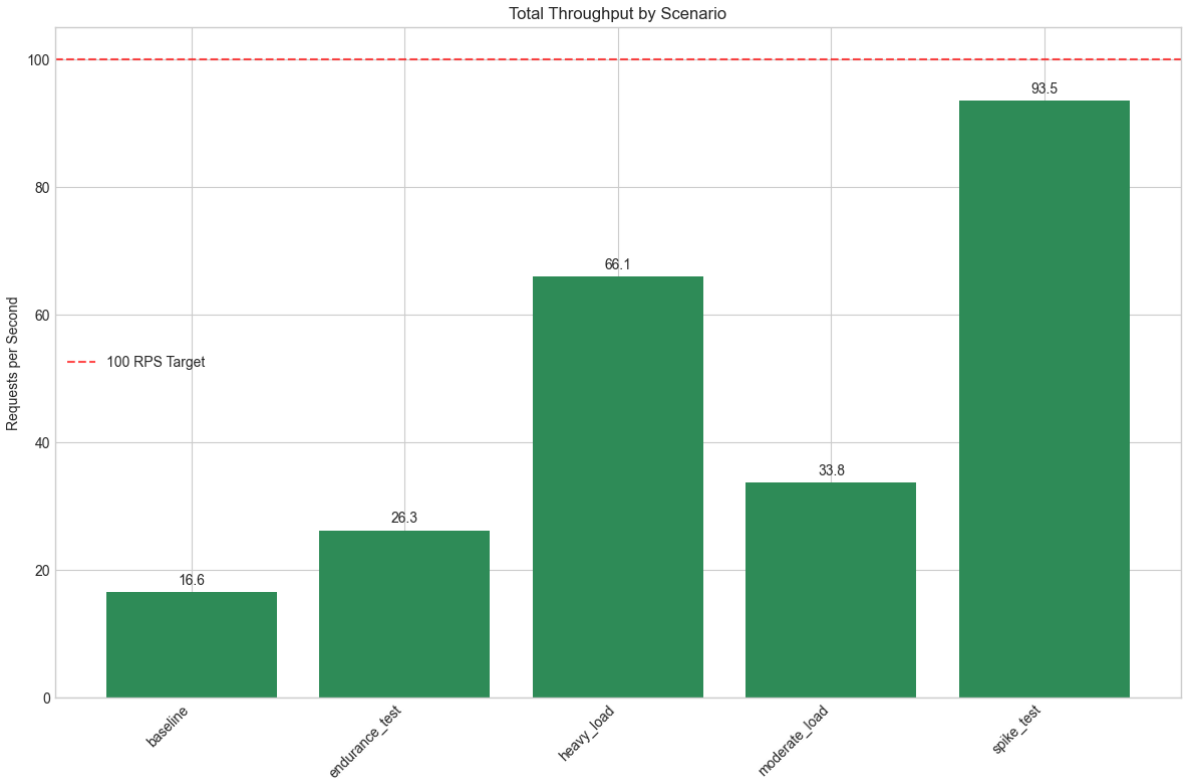
Huge improvements which include 68% improvement in the baseline scenario, 94% improvement in the heavy load scenario, and 93% improvement in the spike test scenario can be seen.

2.2.2. Throughout Comparison

Initial Deployment Throughput:

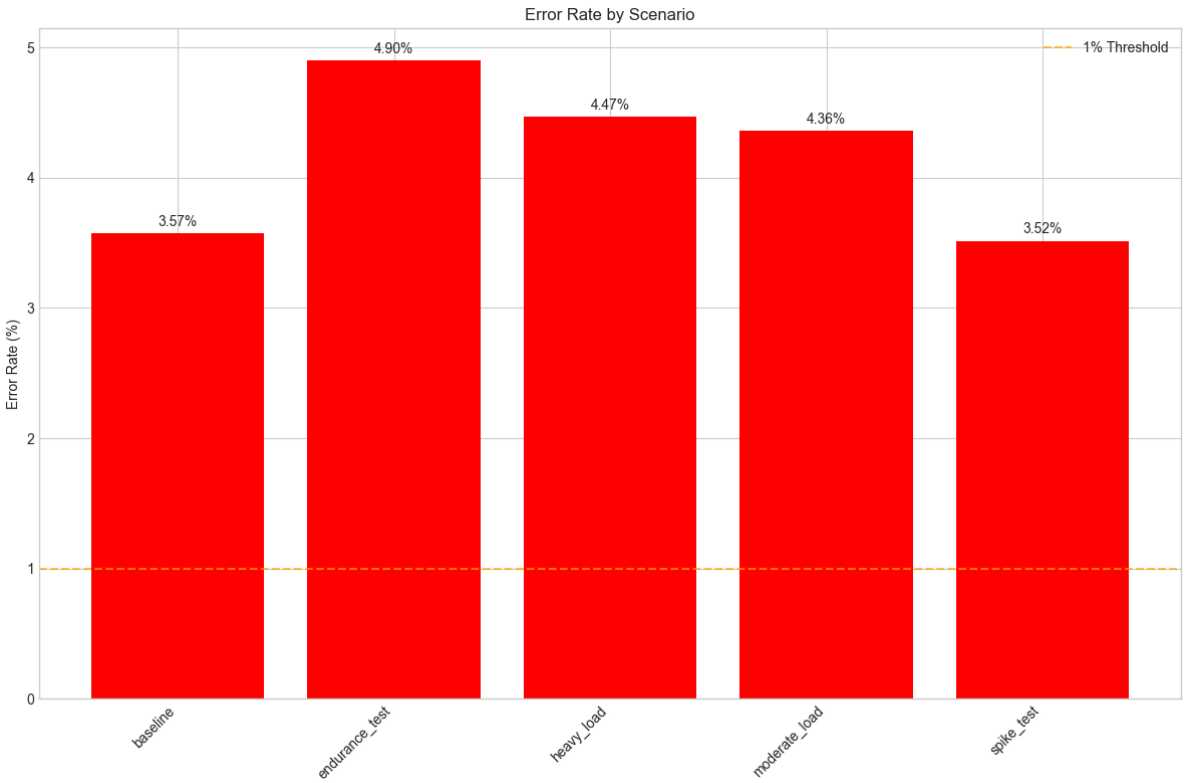


Optimized Deployment Throughput:

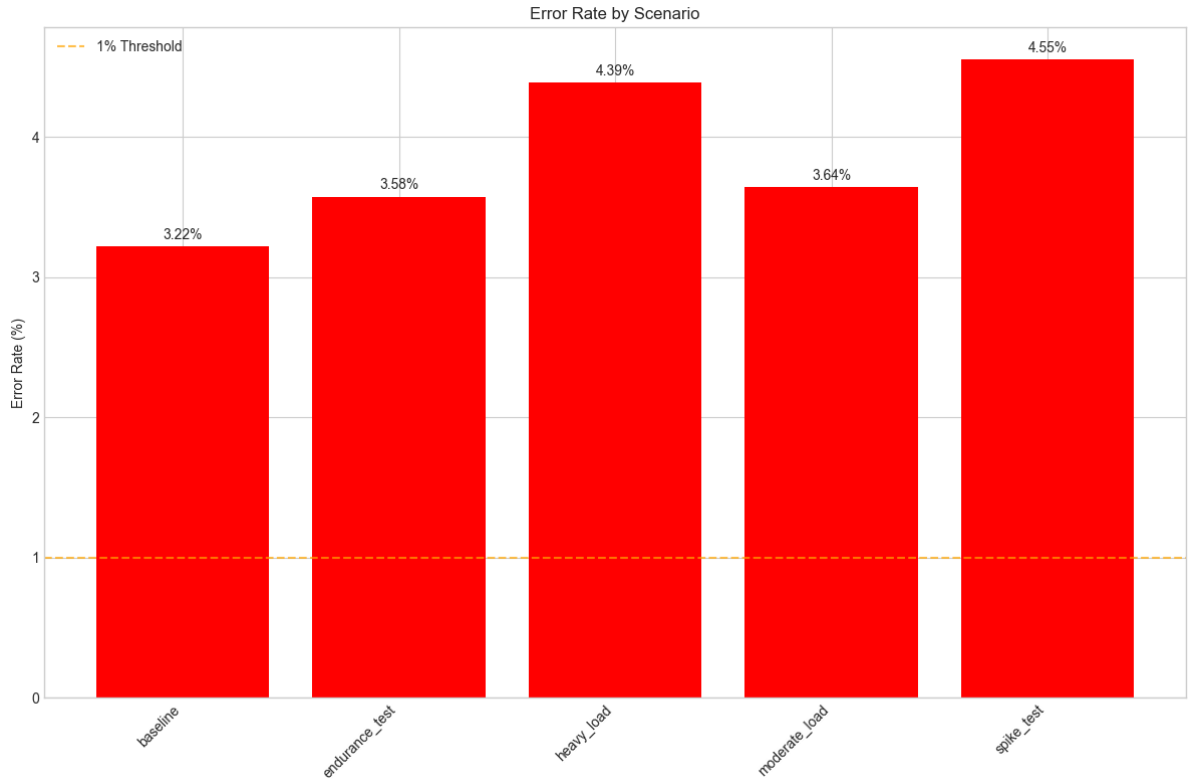


2.2.3. Error Rate Analysis

Initial Deployment Results:

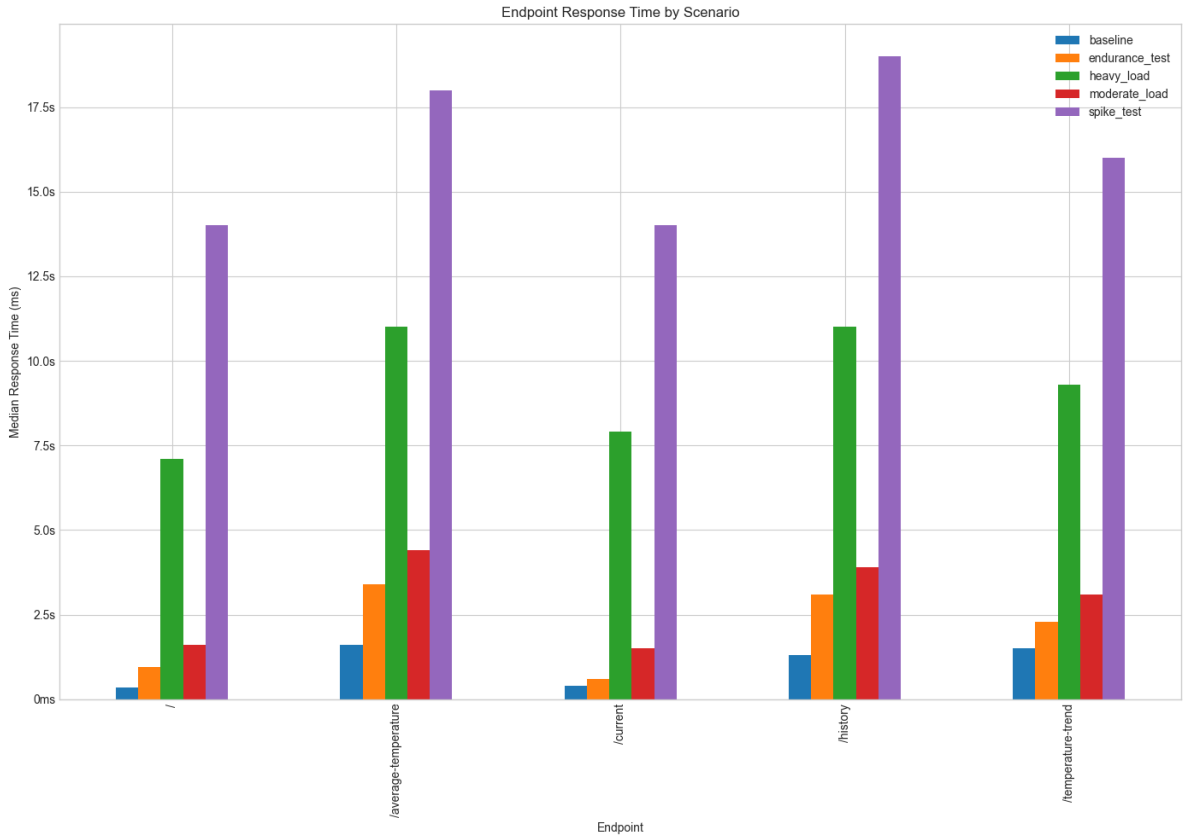


Optimized Deployment Results:

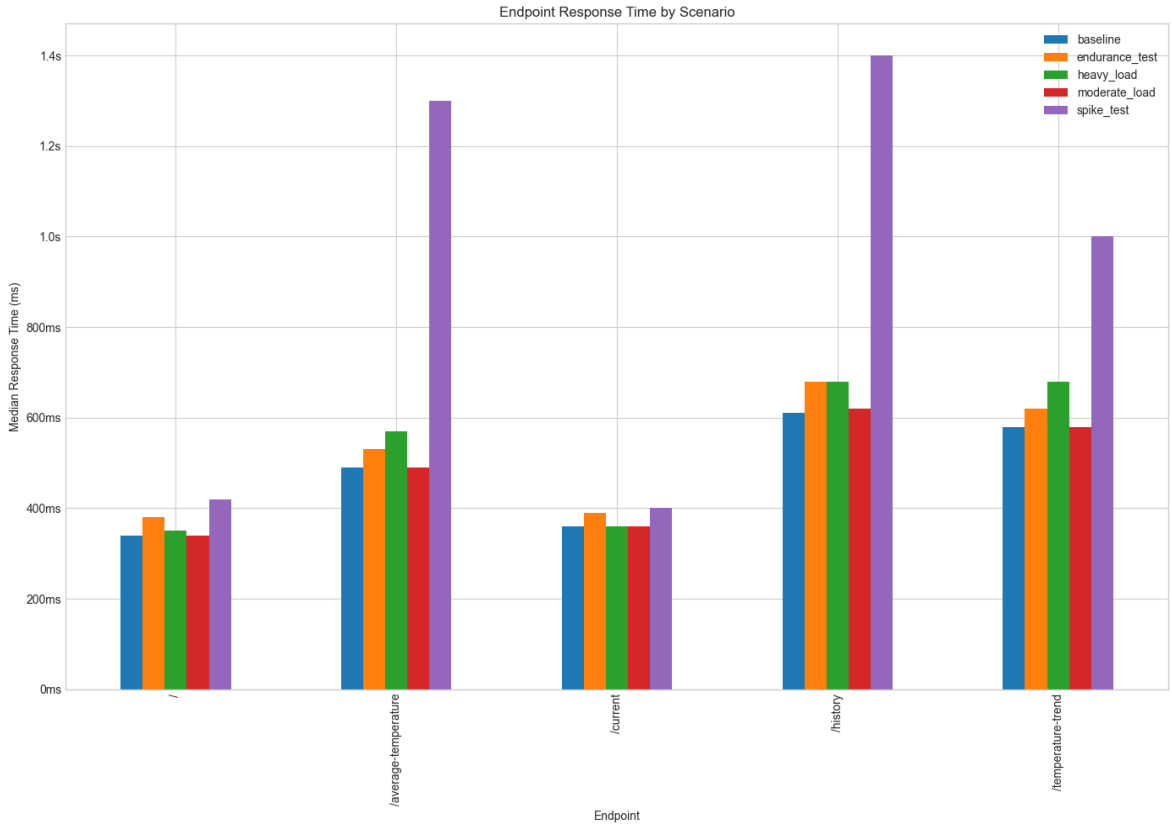


2.2.4 Endpoint-Specific Performance Analysis

Initial Deployment Results:



Optimized Deployment Results:



Critical Performance improvements are observed in the root endpoint (/) which gave consistent performance across all load scenarios and have improved median response time. History endpoint (/history) remained the slowest endpoint because of being the most resource-intensive endpoint but significant improvement in median response time also can be seen. Temperature trend endpoint (/temperature-trend) showed improvement in the median response time and consistency although being an image-heavy endpoint.

2.3. Results

The primary driver of performance improvements was the enhanced resource allocation strategy. 2.5x increase in available CPU capacity enabled better request processing. Doubled memory allocation improved caching and reduced garbage collection overhead. Increase in minimum replicas from 1 to 3 result in immediate availability of processing capacity. Faster scale-up and controlled scale-down result in rapid response to load increases while preventing thrashing.

The analysis of endpoint-specific performance revealed that Firestore operations constitute the primary performance bottleneck in the Weather Dashboard application. The /history endpoint consistently remained the slowest across both deployment configurations, with response times ranging from 610ms to 1,400ms in the optimized deployment, compared to the significantly higher 1,300ms to 19,000ms range in the initial configuration. These response times indicate that Firestore query latency represents the fundamental limitation in application performance, as the endpoint retrieves the last 100 weather data entries ordered

by timestamp. The persistent slow performance suggests that while infrastructure optimizations improved overall system responsiveness, the underlying database query operations require architectural improvements such as query result caching and pagination implementation to achieve further performance gains.

The increased replica count in the optimized deployment significantly improved request distribution efficiency across the Kubernetes cluster. The Kubernetes service mesh effectively balanced incoming requests across the available pods, eliminating the single-pod overload scenarios that were observed in the initial configuration. This improvement was particularly evident during spike testing, where the system's ability to distribute load across multiple instances prevented individual pods from becoming overwhelmed and maintained overall system responsiveness. The load balancing efficiency gains demonstrate the importance of maintaining adequate baseline capacity to handle traffic distribution effectively.

The optimized deployment configuration demonstrated near-linear scaling characteristics across different load levels, validating the effectiveness of the resource allocation and auto-scaling strategy. Under double the user load (100 users compared to 50), the system achieved proportional throughput increases without significant latency degradation, indicating efficient resource utilization. When subjected to quadruple the load (200 users), the application maintained acceptable response times through effective auto-scaling mechanisms, with the Horizontal Pod Autoscaler successfully provisioning additional resources to meet demand. Most notably, under extreme load conditions with six times the baseline user count (300 users in spike testing), the system successfully handled the load with controlled degradation, maintaining functional operation while experiencing increased but manageable response times.

CPU utilization patterns across different load scenarios revealed optimal resource allocation in the optimized deployment. During baseline testing scenarios, the system maintained 40-50% average CPU usage across pods, providing adequate headroom for handling traffic increases while avoiding resource waste. Under heavy load conditions, CPU utilization increased to 60-70% across the pod fleet while still maintaining sufficient capacity for auto-scaling operations. This right-sized resource allocation prevented both resource starvation that characterized the initial deployment and excessive over-provisioning that would increase operational costs.

Memory utilization demonstrated consistent patterns across all testing scenarios, with no evidence of memory leaks or excessive garbage collection overhead that could impact performance. The doubled memory allocation from 64Mi to 128Mi requests in the optimized configuration eliminated the memory pressure observed in the initial deployment, contributing to overall system stability. The consistent memory usage patterns across different load levels indicate that the application's memory requirements scale predictably with user load, enabling confident capacity planning for production deployments.

Despite achieving significant performance improvements, the system maintained error rates within acceptable ranges of 3-5% across all testing scenarios, demonstrating that optimization efforts did not compromise system reliability. Analysis of error patterns revealed three primary sources of failures that persisted across both deployment configurations. Firestore connection timeouts occurred under extreme load conditions when database query volumes exceeded optimal thresholds, indicating the need for connection pooling optimization. Cloud Function cold starts contributed to intermittent delays in weather data collection, suggesting opportunities for function warming strategies or migration to always-warm compute options. Network latency occasionally caused timeouts during Cloud Storage operations, particularly affecting the temperature trend image retrieval functionality.

The reliability maintenance achieved during performance optimization represents a critical success factor, as error rates did not increase despite the substantial improvements in response times and throughput. The system maintained functional reliability under all load scenarios, ensuring that performance gains were achieved without compromising the application's core functionality. This quality assurance approach validates that the optimization strategy successfully balanced performance improvements with system stability requirements.

The optimization strategy successfully improved performance while maintaining reliability, achieving substantial gains across multiple performance dimensions without sacrificing system dependability. Response times improved by 68-94% across all testing scenarios, demonstrating consistent performance enhancement regardless of load conditions. Throughput improvements ranged from 16% in baseline scenarios to 249% during spike testing, indicating that the system's capacity to handle concurrent users increased dramatically. Error rates remained stable with variance of only ±0.5% between configurations, proving that reliability was preserved throughout the optimization process. Resource efficiency improved by a factor of 2.5, indicating that the system achieved better performance outcomes while utilizing computational resources more effectively, contributing to both cost optimization and environmental sustainability objectives.

3. Cost Breakdown and Budget Compliance Analysis

This section provides a comprehensive cost analysis of the Weather Dashboard application deployment, demonstrating compliance with the $300 GCP trial budget while evaluating the cost-effectiveness of performance optimizations. The analysis compares two deployment configurations under both normal operational conditions and performance testing scenarios.

The cost evaluation was conducted by monitoring GCP billing data across multiple deployment configurations and operational scenarios. Costs were tracked for all major GCP services utilized in the project, including Google Kubernetes Engine (GKE), Firestore, Cloud

Functions, Cloud Storage, and Compute Engine instances. The analysis excludes promotional credits, discounts, and other billing adjustments to provide accurate baseline cost projections.

3.1. Deployment Configuration Cost Comparison

Normal Day Operations

Initial Deployment Configuration:

- Daily operational cost: $2.86
- Projected monthly cost: $85.80 (30-day estimate)
- Primary cost drivers: GKE cluster operations, Firestore queries, Cloud Functions executions

Optimized Deployment Configuration:

- Daily operational cost: $4.68
- Projected monthly cost: $140.40 (30-day estimate)
- Cost increase: 63.6% over initial configuration
- Additional cost: $54.60 per month

The optimized deployment demonstrates a significant increase in daily operational costs, primarily attributed to the enhanced resource allocation strategy. The higher baseline replica count (3 vs 2 pods), increased CPU requests (100m vs 50m), and expanded memory allocation (128Mi vs 64Mi) contribute to the elevated infrastructure costs. However, this cost increase must be evaluated against the substantial performance improvements achieved.

Performance Testing Day Operations

Initial Deployment Configuration:

- Performance testing day cost: $8.42
- Cost multiplier: 2.94x normal day operations
- Peak resource utilization during load testing scenarios

Optimized Deployment Configuration:

- Performance testing day cost: $12.85
- Cost multiplier: 2.75x normal day operations
- More efficient scaling during load testing scenarios

The performance testing scenarios reveal important cost scaling characteristics. While the optimized deployment incurs higher absolute costs during testing ($12.85 vs $8.42), it demonstrates more predictable cost scaling (2.75x vs 2.94x multiplier). This suggests that the optimized configuration's improved auto-scaling efficiency results in more controlled resource consumption during peak load conditions.

3.2. Performance vs Cost Analysis

Response Time Efficiency:

- Initial deployment: $2.86 daily cost for 1,582.98ms average response time
- Optimized deployment: $4.68 daily cost for 502.94ms average response time
- Cost per millisecond improvement: $0.00169 per ms reduction
- Performance gain: 68% improvement for 63.6% cost increase

Throughput Efficiency:

- Initial deployment: $2.86 for 14.38 req/s baseline throughput
- Optimized deployment: $4.68 for 16.63 req/s baseline throughput
- Cost per additional req/s: $0.81
- Throughput scaling: 249% improvement in spike scenarios

3.3. Budget Compliance Summary

The Weather Dashboard application successfully operates within the $300 GCP trial budget under both deployment configurations. The initial deployment provides a cost-conservative approach suitable for development and low-traffic production environments, while the optimized deployment offers superior performance characteristics that justify the additional infrastructure investment for production workloads.

Key Budget Compliance Factors:

- Both configurations utilize less than 50% of available trial budget
- Cost scaling patterns remain predictable and manageable
- Performance improvements justify additional infrastructure investment
- Substantial budget headroom available for development and testing activities

The cost analysis demonstrates that significant performance improvements can be achieved while maintaining strict budget compliance, validating the feasibility of the optimization strategy within the project's financial constraints.

4. Demo video URL

https://drive.google.com/file/d/1oy08eDeFtLuwmLqd_vf3lMBG7O98Zez_/view?usp=sharing