Universidade do Minho
Escola de Engenharia

# Aplicações e Serviços de Computação em Nuvem

2025/2026

Trabalho Prático

Daniel Francisco Teixeira Andrade pg60242

José Diogo Azevedo Martins pg59788

Luis Enrique Díaz De Freitas pg59789

Maria Miguel Machado Loureiro pg58714

Pedro André Ferreira Malainho pg61005

# Contents

# List of Figures

# List of Tables

# 1. Air Trail

AirTrail is a modern, open-source web application designed as a personal flight tracking and logging system. It serves as a self-hosted alternative to commercial platforms, allowing users to maintain more ownership and privacy over their aviation travel data. For it's users, AirTrail provides an easy and clean interface, present at Figure 1:

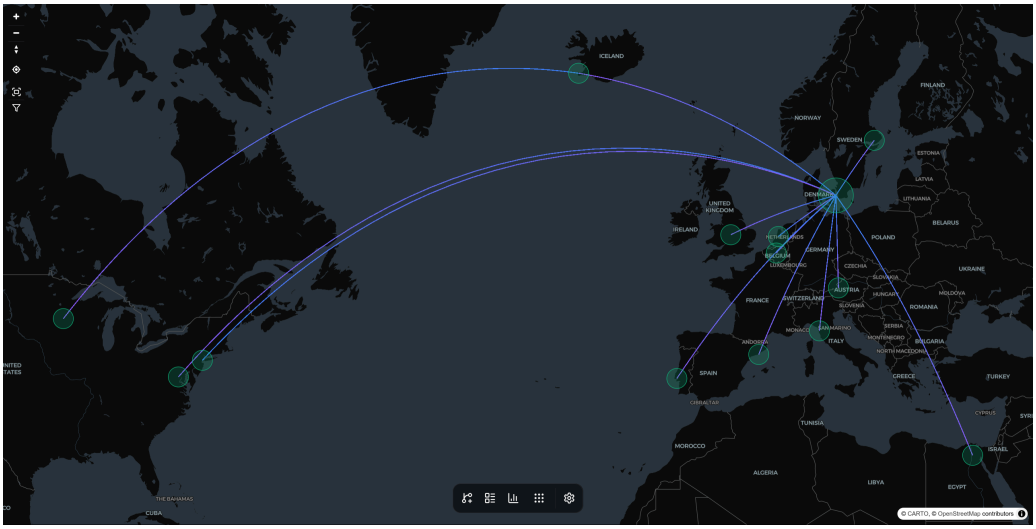


Figure 1: AirTrail Web Interface

AirTrail provides a comprehensive set of functionalities aimed at the efficient management, visualization, and analysis of personal flight data:

- **Personal Flight Logging:** Users can store and organize their flight history by manually logging or importing detailed flight information by including dates, airlines, aircraft types, and flight numbers. You can visualize their routes on an interactive global map that highlights flight paths, airport connections, and overall geographical coverage.
- **Data Analytics:** The platform also offers built-in analytics that automatically compute key statistics such as total distance flown, time spent in the air, most visited airports, and most frequently used airlines, enabling users to better understand their travel patterns.
- **Interoperability:** Additionally, AirTrail ensures seamless interoperability by supporting direct import of flight history from platforms such as MyFlightRadar24, App in the Air, and Flighty, as well as TripIt (ICS) files and AirTrail's own JSON format.

In the context of Cloud Computing, AirTrail serves as an ideal use case for containerized deployment. It is primarily distributed via Docker, facilitating easy orchestration on cloud platforms.

# 2. Architecture and Components

## 2.1. General Overview

The AirTrail application is built on a modular yet monolithic architecture, integrating all core functionalities within a single repository to simplify deployment and maintenance.

### 2.1.1. Architectural Layers

As illustrated in Figure 2, the system is organized into four primary logical layers that govern the flow of data from the user to the underlying infrastructure.

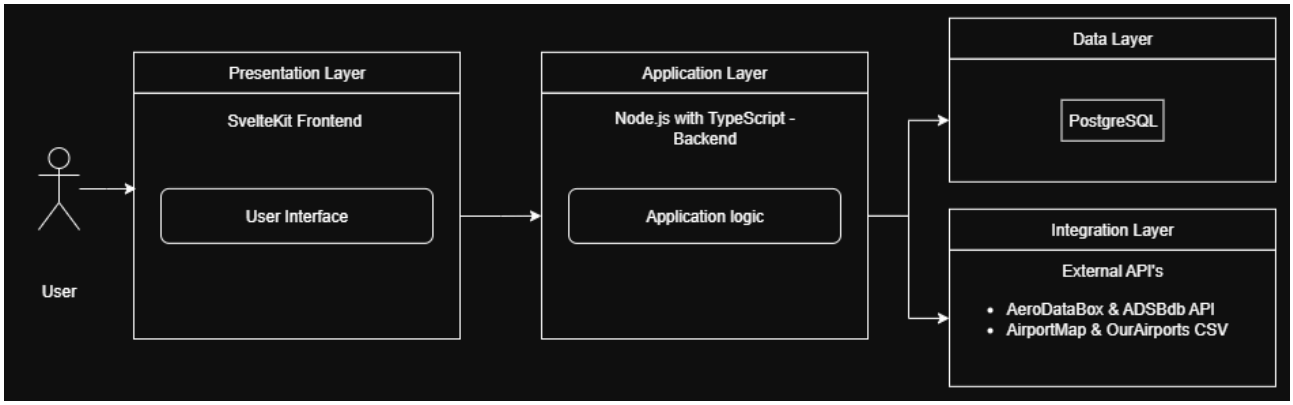


Figure 2: Application Architecture and Components

- **Presentation Layer**: This layer consists of a SvelteKit Frontend, which handles the User Interface (UI). It utilizes Server-Side Rendering (SSR) to pre-render content, ensuring faster initial load times and improved SEO. The SvelteKit Server is organized into several core components:

  – **Page Routes**: define the application routes and determines which page should be rendered for each request;

  – **Load Functions**: executed prior to rendering each page, the functions are responsible for preloading the necessary data;

  – **API Routes**: expose HTTP endpoints that facilitate communication between the frontend and backend;

  – **Services**: contain the application's business logic, including database operations and other specific functionalities.

- **Application Layer:** Acting as the orchestrator, this layer uses Node.js with TypeScript. It contains the application logic, processing requests from the frontend through API Routes and executing business rules via dedicated Services;
- **Data Layer:** For persistent storage, AirTrail relies on a PostgreSQL database. Communication between the Application Layer and the database is managed by Prisma ORM, which provides a type-safe interface for querying and managing relational data;
- **Integration Layer:** To enrich the user experience with real-time aviation data, the system connects to External APIs. This includes AeroDataBox and ADSBdb for flight lookups. These are

external components because they exist outside the monolithic codebase. AirTrail uses ADSBdb by default for basic flight number lookups, while AeroDataBox provides enhanced data coverage and the ability to search flights by date. Flight lookup features that rely on AeroDataBox are unavailable until a valid API key is configured, but the rest of AirTrail continues to operate normally.

## 2.2. Features and API

### 2.2.1. Features

The system supports multiple user accounts with secure authentication and OAuth integration. The responsive interface ensures usability across devices, while light and dark mode options allow users to customize the interface according to their preference. Additionally, Airtrail allows the import of light data from external services such as MyFlightRadar24, App in the Air, and JetLog, simplifying data migration.

### 2.2.2. API

To support these functionalities, AirTrail exposes a set of RESTful APIs for flight management. The `/flight/list` endpoint retrieves all flights associated with a user. The `/flight/save` endpoint allows the creation of new flights or updates to existing ones based on the presence of an ID. Specific flight details can be obtained using `/flight/get/{id}`, while `/flight/delete` enables the removal of flights by ID.

| Endpoint | Method | Summary | Description |
|---|---|---|---|
| /flight/list | GET | List flights | List all your flights |
| /flight/save | POST | Create or update a flight | Create a new flight or update an existing one. If thr `id` field is present, the flight will be updated. Otherwise, a new flight will be created. |
| /flight/get/{id} | GET | Get flight details | Get details of a specific flight by ID |
| /flight/delete | POST | Delete a flight | Delete a flight by ID |

Table 1: RESTful API Table

# 3. Automation: Installation and Configuration

To ensure a reproducible and "one-click" setup, we used Ansible to automate the deployment on Google Kubernetes Engine (GKE). This approach moves away from manual console configuration toward Infrastructure as Code (IaC), ensuring that the environment is documented, version-controlled, and easily replicable.

## 3.1. The Role of Ansible

Ansible serves as the primary Infrastructure as Code (IaC) and automation engine for this project, acting as the central orchestration layer that governs the setup and configuration of the entire cloud environment. Instead of requiring a developer to manually execute commands to configure and manage servers, Ansible utilizes human-readable scripts called playbooks to standardize and automate these workflows.

Its role is twofold: **first**, it interfaces with Google Cloud to provision the Google Kubernetes Engine (GKE) cluster; **second**, it manages the continuous deployment of the AirTrail application.

By using Ansible, we achieve a high level of consistency and replicability, ensuring that the infrastructure is instantiated exactly the same way every time, effectively eliminating configuration drift and human error.

Ultimately, allows us to manage complex cloud resources through a unified, version-controlled pipeline, making the deployment process fast reliable, and production-ready

## 3.2. System Configuration

The automation lifecycle is iniciated by the System Configuration, which servers as the central repository for all environment parameters required for deployment.

This stage is managed through a dynamic inventory system and comprehensive variable files that define operational parameters of the Google Cloud Plataform (GCP) environment, such as the project ID, regional zones, and authentication credentials. These settings act as the essential configuration layer, granting the automation tools the necessary permissions and context to interact with the Google Cloud API, ensuring that the infrastructure is built exactly where and how its intended.

This stage is critical for establishing the secure tunnel and resource definitions required to move from an abstract design to the physical instalation of the GKE Cluster.

## 3.3. Automated Infrastructure

Once the configuration parameters are established, the process transitions into the Infrastructure Installation phase. This stage is the transition from design to the physical provisining phase.

```
.
├── airtrail-deploy.yml
├── airtrail-undeploy.yml
├── credentials
├── gcp-create-vm.yaml
├── gke-cluster-create.yml
├── gke-cluster-destroy.yml
├── inventory
└── roles
```

To manage the infrastructure described above, we developed five main Ansible playbooks, each responsible for a specific stage of the environment lifecycle:

- **gke-cluster-create.yml**- This playbook sends a request to Google Cloud to provision the GKE Clusterand its node pools, setting up the foundation where our containers will run.
- **gke-cluster-destroy.yml**- This playbook deletes the GKE cluster and all associated cloud resources.
- **airtrail-deploy.yml**- This playbook handles the application setup. It deploys the AirTrail, including frontend and database.
- **airtrail-undeploy.yml**- This playbook removes the application from the cluster. It cleans up the deployments while keeping the cloud hardware active.
- **gcp-create-vm.yaml**-This playbook was used to provision a Virtual Machine in Google Cloud specifically to run JMeter. Allowing us to perform load testing on the application, simulating real-world traffic to evaluate the system's performance.

### 3.3.1. Installation

Using Ansible playbooks, a request is sent to Google Cloud Plataform to instantiate the GKE cluster and its associated node pools.This phase manages the complex orchestration of cloud resources, ensuring that computing power, memory, and networking are precisely allocated to meet the application's demands.

The deployment of the AirTrail application is managed by the `airtrail-deploy.yml` playbook, which orchestrates the setup through two distinct roles:

- **airtrail_deploy**: This role manages the frontend application logic.
  - It creates the **Kubernetes Deployment** using the `airtrail-deployment.yaml` template, configuring the application to connect to the database via the Pgpool middleware.
  - It establishes external access by provisioning a **LoadBalancer Service** defined in `airtrail-service.yaml`, mapping port 80 to the application's port 3000.
  - Finally, it automatically retrieves the external IP assigned by GCP and updates the local Ansible inventory, allowing subsequent test playbooks to interact with the deployed application immediately.
- **airtrail_db**: This role orchestrates the complex persistence and data layer architecture.

- **Database Nodes**: It deploys a primary PostgreSQL StatefulSet (`postgres-master`) for write operations and a secondary StatefulSet (`postgres-slave`) for read replicas.
- **Autoscaling**: It applies a Horizontal Pod Autoscaler (`postgres-slave-hpa`) to the slave nodes, ensuring the read tier scales dynamically based on CPU utilization.
- **Middleware**: It deploys Pgpool-II (`pgpool-deployment.yaml`) to act as a sophisticated proxy. This component manages connection pooling and routes traffic appropriately between the master and slave nodes.

```
airtrail_deploy/
├── tasks
│   └── main.yaml
├── templates
│   ├── airtrail-deployment.yaml
│   └── airtrail-service.yaml
└── vars
    └── main.yaml
```

```
airtrail_db/
├── tasks
│   └── main.yaml
├── templates
│   ├── pgpool-deployment.yaml
│   ├── pgpool-service.yaml
│   ├── postgres-master-service.yaml
│   ├── postgres-master-statefulset.yaml
│   ├── postgres-slave-hpa.yaml
│   ├── postgres-slave-service.yaml
│   └── postgres-slave-statefulset.yaml
└── vars
    └── main.yml
```

**Note:** The shown `airtrail_db` role refers to the optimized database. Initially the role contained a simple postgres service and deployment. More details about the optimization in Chapter 5.

### 3.3.2. Resource Teardown and Cost Management

The final part of ourt workflow is the Resource Teardown. While the first part of the project focuses on building the environment, it is equally important to have a way to completely remove it. To do this, we created a specific "destroy" playbook that handles the decommissioning of all services.

```
airtrail_undeploy/
├── tasks
│   └── main.yaml
└── vars
    └── main.yaml
```

```
undeploy_airtrail_db/
├── tasks
│   └── main.yaml
└── vars
    └── main.yaml
```

As shown in the directories above, the airtrail_undeploy.yml playbook is organized into specific roles to ensure modular and clean removal of the application. Both roles target different layers of the system independently:

- **airtrail-undeploy:** This role is responsible for removing the core application components refered above from the GKE cluster.
- **undeploy_airtrail_db:** This role handles the removal of PostgreSQL database.

In Cloud Computing, this is a vital step for Cost Management. Since Google Cloud charges for every hour a resource is active, leaving a cluster running when it is not needed would lead to unnecessary expenses. By automating the destruction of the infrastructure, we make sure that no "orphaned" resources—like disks or load balancers-are left behind accidentally.

This automation allows us to follow the “pay-as-you-go” principle, ensuring we only pay for exactly what we use during our work. Furthermore, it gives us the ability to reset everything to a clean slate with a single command, making it easy to restart the entire process from scratch whenever necessary without any leftover “garbage” from previous tests.

# 4. Experimental Analysis of the Base Architecture

## 4.1. Base Architecture

The primary objective is to analyze how the system behaves under a controlled infrastructure and to validade the effectiveness of the automated deployment process, using the **baseline infrastructure (before the otimization explained in the chapter 5)**. The application was deployed on a Google Kubernetes Engine (GKE) clustes, utilizing the automated workflows previously described. This setup ensures that the environment is consistent and scalable.

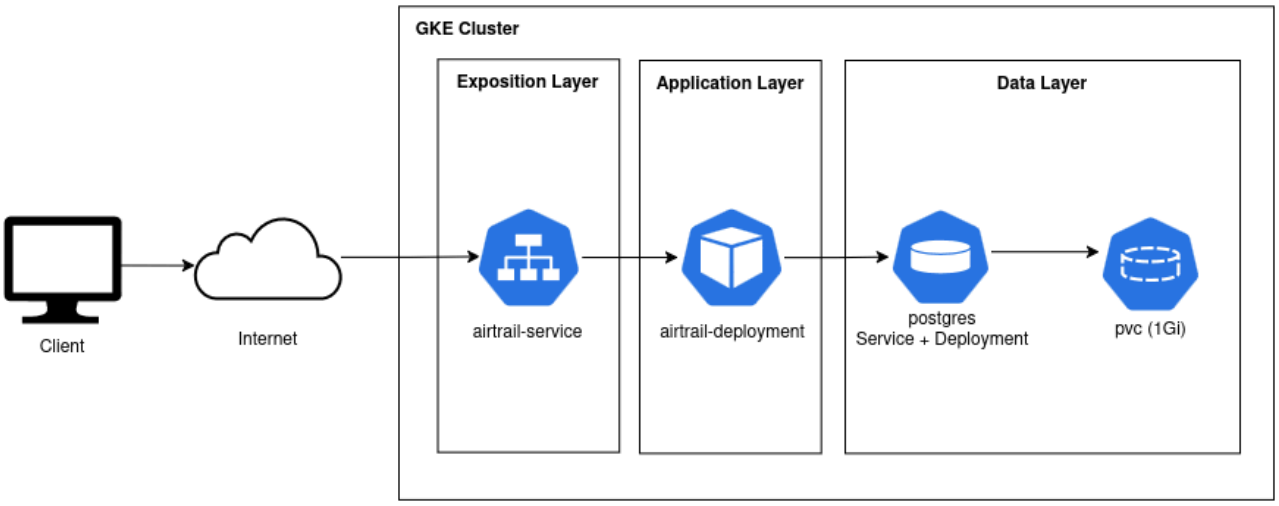


Figure 3: Baseline Deployment Architecture

As illustrated in the Deployment Architecture (Figure 3) once the infrastructure is ready, the automation orchestrates the internal components of the AirTrail system into three distinct functional zones:

- **Exposition Layer:** The automation establishes the airtrail-service, which acts as the entry point for external traffic from the Internet.
- **Application Layer:** In this layer, the airtrail-deployment is instantiated, running the core SvelteKit and Node.js logic.
- **Data Layer:** The installation concludes with the deployment of the PostgreSQL database, coupled with a Persistent Volume Claim (PVC 1Gi) to ensure data persistence.

## 4.2. Theoretical Bottlenecks and SPOF

### 4.2.1. Performance Bottlenecks

Application performance bottlenecks occur when a specific component of the system limits the overall capacity of the application to deliver results efficiently and quickly, even when other resources remain available. In other words, the system is only as strong as its weakest link. In the AirTrail application, potential performance bottlenecks can emerge both at the server level (containers and database) and at the client level (user interface and frontend performance).

**CPU and Memory**

Compute resources are finite constraints in any containerized environment. Both the AirTrail application and the PostgreSQL database compete for CPU cycles and memory. Under high concurrency or heavy load, CPU saturation results in process throttling and increased latency, while memory exhaustion can lead to instability or cause Kubernetes to terminate pods.

**External calls**

Apart from relying in the communication with the PostgreSQL database, AirTrail also uses external APIs to retrieve or update data. Each interaction adds latency, especially if handled synchronously. When the backend waits for these responses before continuing, total response time increases. High numbers of simultaneous API or database calls may also exceed connection limits.

**Processing and Threading**

In AirTrail's current containerized setup, the backend runs as a single service, which may limit its ability to handle requests asynchronously. Bottlenecks can occur when heavy computations or asynchronous logic block the handling of new requests. Operations like flight path processing are executed sequentially, the number of concurrent users the system can serve decreases, leading to reduced scalability and slower response times under load.

**Databases Performance and Concurrency**

High CPU and memory usage, combined with poorly optimized queries, can slow response times and overload the server, especially under heavy load or with many concurrent users. In multi-user environments, concurrent write operations can cause blocking or deadlocks, slowing down transactions.

#### 4.2.2. Single Points of Failure

Potential single points of failure during the installation were analyzed in the current AirTrail setup and identified areas where a single component failure could impact the whole system.

- **Database Server:** The database is one of the main single points of failure. If the database server fails, the application will no longer be able to store or retrieve information about users and flights.
- **Web/Application Server:** If there is only one instance of the application running on a single node, any failure in that pod will result in service unavailability.
- **External Dependencies and APIs:** AirTrail relies on external flight tracking APIs. A failure in these services can affect core functionality.

## 4.3. Evaluation Methodology and Tools

#### 4.3.1. Monitoring Tool

For real-time monitoring and performance analysis, Google Cloud Monitoring was employed, using the integrated dashboards available in the Google Cloud Console.

This tool was selected because it provides native and seamless integration with Google Kubernetes Engine (GKE), enabling the collection of performance metrics at the pod and container levels without requiring additional monitoring agents or external tools.

Monitoring focused on Kubernetes container level metrics for both the AirTrail application container and the PostgreSQL database container. The following metrics were analyzed:

- CPU usage, to identify processing bottlenecks and saturation points under load.
- Memory usage, to detect memory pressure and potential resource exhaustion.
- Network traffic (bytes sent and received), used as an indirect indicator of I/O activity, due to the absence of dedicated disk I/O or database-level metrics.

Because no database-specific instrumentation (such as PostgreSQL exporters) was deployed, direct metrics related to disk I/O, query latency, or active database connections were not available. Consequently, network traffic metrics were used as a proxy to infer I/O-related behavior, particularly during periods of high concurrency and intensive database access.

The monitoring dashboards were observed during the execution of load tests, allowing the correlation of resource utilization patterns with applied load levels and test duration. This approach provided sufficient visibility to identify real performance bottlenecks prior to optimization.

#### 4.3.2. Load Test Tool

To evaluate the baseline performance and scalability of the system, Apache JMeter was used as the load testing tool. The load tests were executed using a custom Apache JMeter test plan, with JMeter running on a dedicated virtual machine in the same Google Cloud region as the AirTrail application cluster and its PostgreSQL database. By executing the load generator within the same cloud environment and region as the application and database cluster, external wide-area network latency was minimized, ensuring that the observed performance results primarily reflect application and database behavior rather than internet-scale network delays.

The JMeter test plan simulated authenticated users interacting with the system through its REST API endpoints. Each test iteration consisted of:

- One **POST** request to create a flight record.
- One **GET** request to retrieve the list of flights.

Authentication was performed using an API key passed through HTTP headers, allowing all requests to be processed as authenticated API calls.

To assess system behavior under increasing load, two distinct load test scenarios were executed: **1000**, and **5000** concurrent threads.

All tests used a ramp-up period of 30 seconds and 20 iterations per thread. The results of these tests are summarized and compared in subsection 4.4 (Baseline Performance), allowing a clear comparison of system behavior as concurrency increases.

Among these scenarios, the 5000-thread configuration represents the most demanding workload and was selected as the critical stress test. For this reason, the detailed performance analysis presented in the following sections focuses primarily on the results obtained from this test.

During test execution, system behavior was monitored in real time using Google Cloud Monitoring dashboards, enabling correlation between applied load and resource utilization across the application and database components.

## 4.4. Baseline Performance

The analysis focuses on the behavior of the AirTrail application container and the PostgreSQL database container under high concurrency, using CPU, memory, and network traffic metrics, **before the optimization**.

To provide a consolidated overview of the system behavior under different concurrency levels, Table 2 summarizes the main resource utilization metrics observed during the baseline load tests. The table presents CPU usage, memory consumption, and network traffic for both the AirTrail application container and the PostgreSQL database container under the 1000- and 5000-thread configurations, serving as a reference for the detailed analysis discussed in this subsection.

| Number of Threads | CPU Usage(Total: 2vCPU) | | Memory usage(Total: 2GiB) | | Network Traffic | |
|---|---|---|---|---|---|---|
| | Airtrail | Database | Airtrail | Database | Airtrail | Database |
| 1.000 | Low–Moderate (0.2–0.45 cores) | Medium-High (1.3–1.7 cores) | Stable (600–700 MiB) | Stable (70–75 MiB) | Moderate | High |
| 5.000 | Low-Moderate (0.2-0.45 cores) | Medium-High (1.2-1.6 cores) | Stable (800-950 MiB) | Stable (60-75 MiB) | Moderate | High |

Table 2: Tests Results and Comparison

At the end of all monitoring graphs, shown in the next subsections, a sharp decrease in resource usage is observed. This behavior corresponds to the termination of the load test execution or a container lifecycle event, rather than a gradual change in system behavior. For this reason, the analysis of each metric focuses primarily on the active execution period of the workload, during which the system is under load and the observed resource utilization patterns are representative of baseline performance.

### 4.4.1. Application Container (AirTrail)

Figure 4 illustrates the CPU usage of the AirTrail application container during the baseline load test with 5,000 concurrent threads. CPU usage remains consistently low throughout the execution window, with only minor fluctuations and no sustained peaks or signs of CPU saturation. This indicates that the application layer operates efficiently under the applied load.

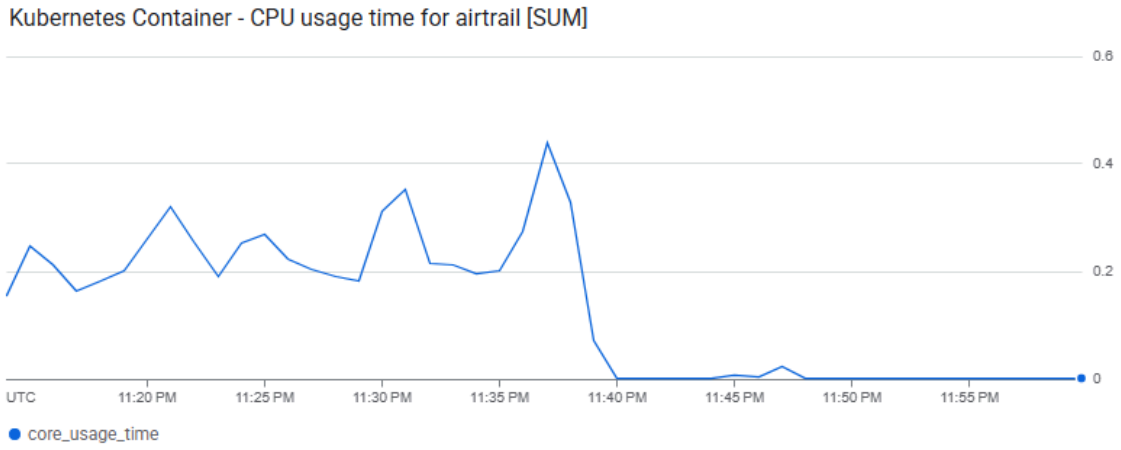


Figure 4: AirTrail container CPU usage during baseline load test

Memory usage for the AirTrail application container, shown in Figure 5 increases rapidly at the beginning of the test and remains relatively stable during the active execution period. However, a sharp and abrupt decrease in memory usage is observed towards the end of the graph, indicating a container lifecycle event, such as workload termination or container restart.

During the active phase of the test, there is no evidence of continuous memory growth or memory exhaustion, suggesting that memory usage remains controlled while the workload is running.

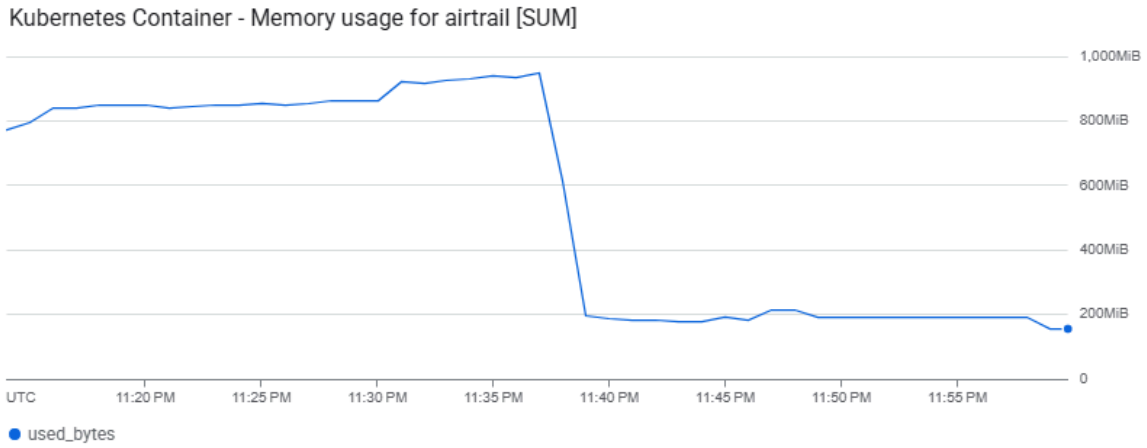


Figure 5: AirTrail container memory usage during baseline load test

Overall, the AirTrail application container demonstrates stable behavior during the baseline load test with 5,000 threads, with no signs of CPU saturation, excessive memory growth, or runtime instability.

### 4.4.2. Database Container (PostgreSQL)

In contrast to the application container, the PostgreSQL database container exhibits significantly higher resource utilization during the same test. As shown in Figure 6, CPU usage for the PostgreSQL container increases sharply at the beginning of the test and remains consistently high throughout the execution window. Several peaks are observed, indicating periods of intensive query processing and concurrent database access.

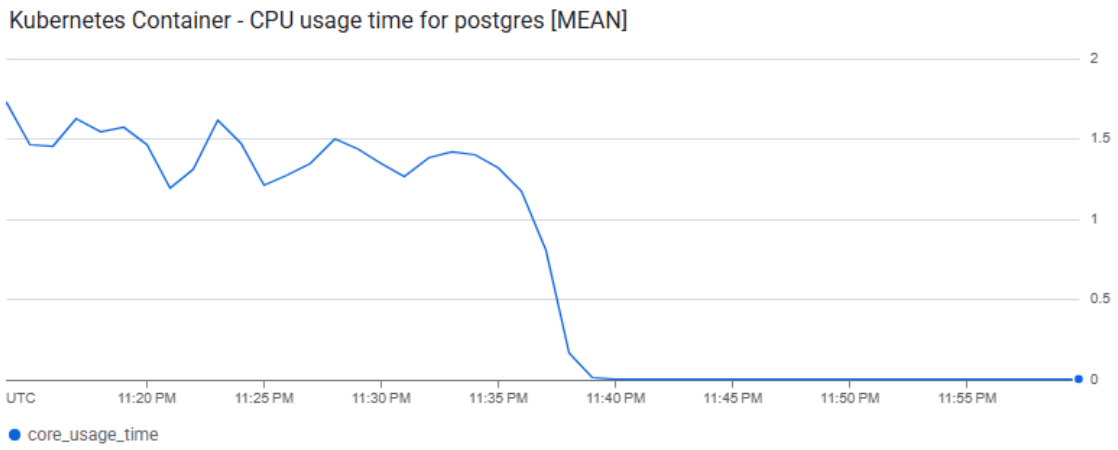


Figure 6: PostgreSQL container CPU usage during baseline load test

Memory usage for the PostgreSQL container, presented in Figure 7, shows an initial increase followed by stabilization at a sustained level during the test execution. This behavior is consistent with continuous memory utilization associated with database operations, such as caching and buffer management, throughout the workload.

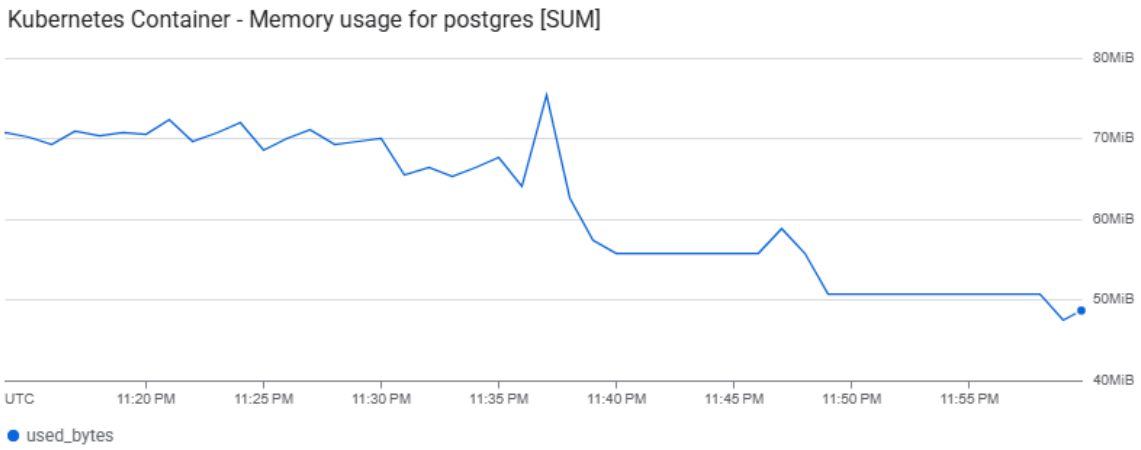


Figure 7: PostgreSQL container memory usage during baseline load test

Overall, these observations indicate that the database layer operates under considerably higher resource pressure during the test execution and may represent a performance bottleneck under this load scenario.

### 4.4.3. Network Traffic Analysis

Due to the absence of direct disk I/O metrics or database-level instrumentation, network traffic metrics were used as an indirect indicator of I/O activity. Figure 8 shows the total network traffic (bytes received and transmitted) at the pod level for the AirTrail application during the baseline load test with 5,000 concurrent threads. Periodic peaks in received bytes are observed, corresponding to bursts of incoming requests generated by the load test, while transmitted traffic remains comparatively low and stable.

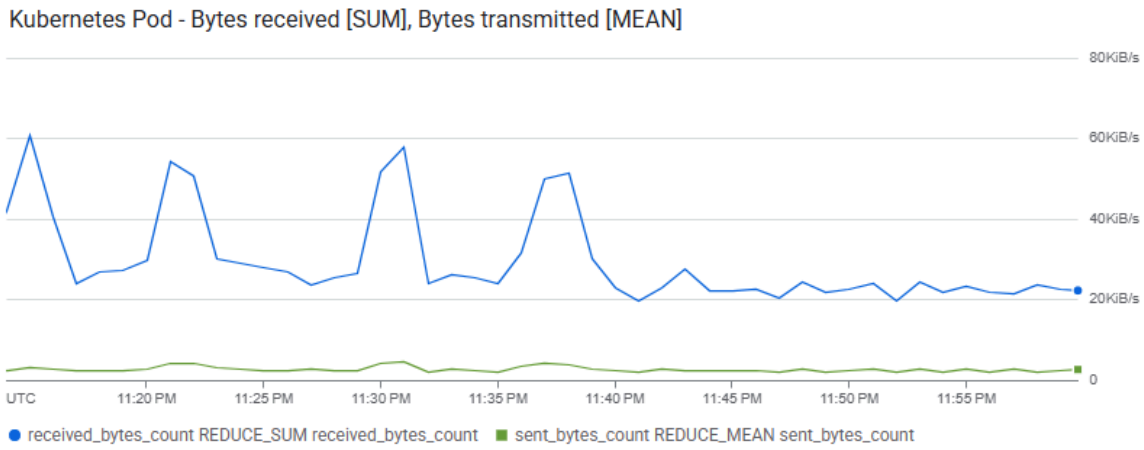


Figure 8: PostgreSQL container memory usage during baseline load test

Figure 9 presents the network traffic for the PostgreSQL pod during the same test. The received and transmitted bytes exhibit a fluctuating pattern with noticeable peaks, indicating intensive data exchange associated with database operations. These variations align temporally with the CPU usage patterns observed for the PostgreSQL container, reinforcing the conclusion that the database handles a high volume of I/O-related activity during the test.

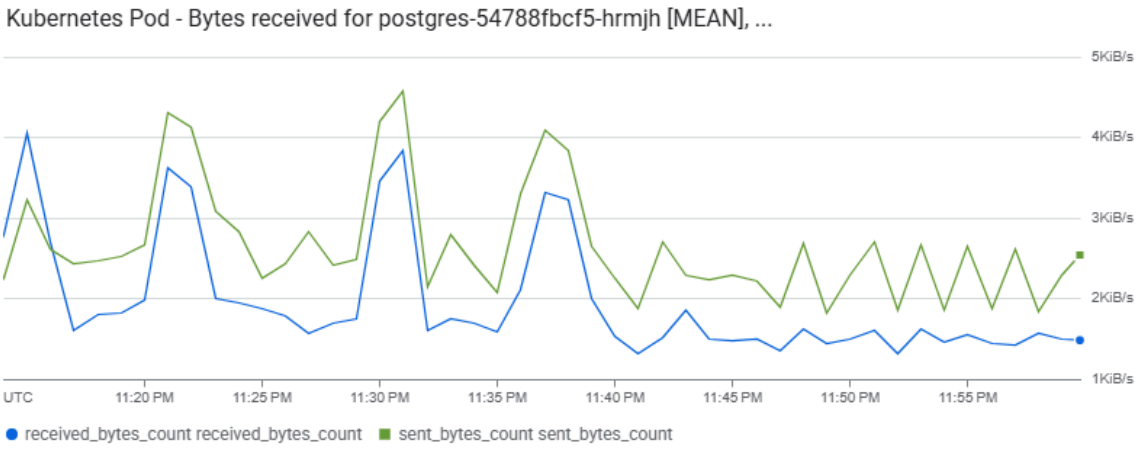


Figure 9: PostgreSQL container memory usage during baseline load test

Overall, the network traffic analysis shows predictable and stable behavior at the application level, while the PostgreSQL pod exhibits higher and more variable network activity. This pattern indicates that network I/O is primarily driven by database operations, reinforcing the database layer as the most resource-intensive component during the load test.

## 4.5. Diagnosis and Confirmed Bottlenecks

### 4.5.1. Identified Bottlenecks

Based on the baseline performance evaluation, it was possible to confirm the initial hypothesis that the PostgreSQL database constitutes the primary performance bottleneck of the system.

Load tests executed with increasing numbers of concurrent clients (1.000 and 5.000 threads) revealed a clear degradation in system performance as the workload increased. While the **AirTrail application** container maintained relatively **stable CPU and memory** usage across all load levels, the **PostgreSQL** container exhibited sustained **high CPU utilization and steadily increasing memory consumption** under higher concurrency.

Monitoring data shows that, as the number of concurrent users increased, the **database layer** experienced **significantly higher processing pressure**. CPU usage remained consistently elevated during the tests, and memory consumption increased progressively, indicating intensive query execution and buffer usage. In contrast, the application layer did not show signs of saturation, suggesting that request handling and business logic execution were not the limiting factors.

Additionally, network traffic metrics for the PostgreSQL pod exhibited recurrent peaks that correlated with periods of high CPU usage, indicating intensive data access and transfer between the application and the database. Although direct disk I/O metrics were not available, the combined evidence from CPU usage, memory consumption, and network activity strongly suggests that database operations dominate system resource utilization under high workloads.

Overall, these results demonstrate that the **application's performance** under different numbers of clients and workloads is **primarily constrained by the database layer**, confirming **PostgreSQL** as the **main bottleneck** prior to optimization.

### 4.5.2. The Single Point of Failure

The analysis also identified the PostgreSQL database as a single point of failure (SPOF) in the initial system architecture.

The database was deployed as a single instance, meaning that any failure or unavailability of this component would render the entire application unusable. Furthermore, this single-instance setup limits scalability, as all read and write operations are handled by the same database node, increasing contention under high load.

Nonetheless, both components can represent SPOF since, prior to optimization, it only exist on instance of each. The work and analysis provided us with an answer to the most important and likely to fail component.

The observed performance bottlenecks, combined with the lack of redundancy, highlight the need for database-level optimizations and architectural improvements. Addressing this single point of failure is essential not only to improve performance, but also to enhance the overall resilience and availability of the system

# 5. Optimization and Comparative Results

Following the initial deployment and analysis of the application, the identified bottlenecks guided our optimization strategy. **The primary target for optimization was the Database.**

## 5.1. Optimization Strategy

We determined that a standard Kubernetes `Deployment` is unsuitable for scaling databases because all replicas share a single Persistent Volume. Multiple instances attempting to write to the same filesystem simultaneously would cause fatal metadata collisions and data corruption.

To address this, our strategy focused on a **Primary/Replica (Master/Slave)** architecture using Kubernetes **StatefulSets**. This ensures data consistency while allowing us to scale read operations using the **Horizontal Pod Autoscaler (HPA)**. These changes resulted in the architecture represented by the following figure:

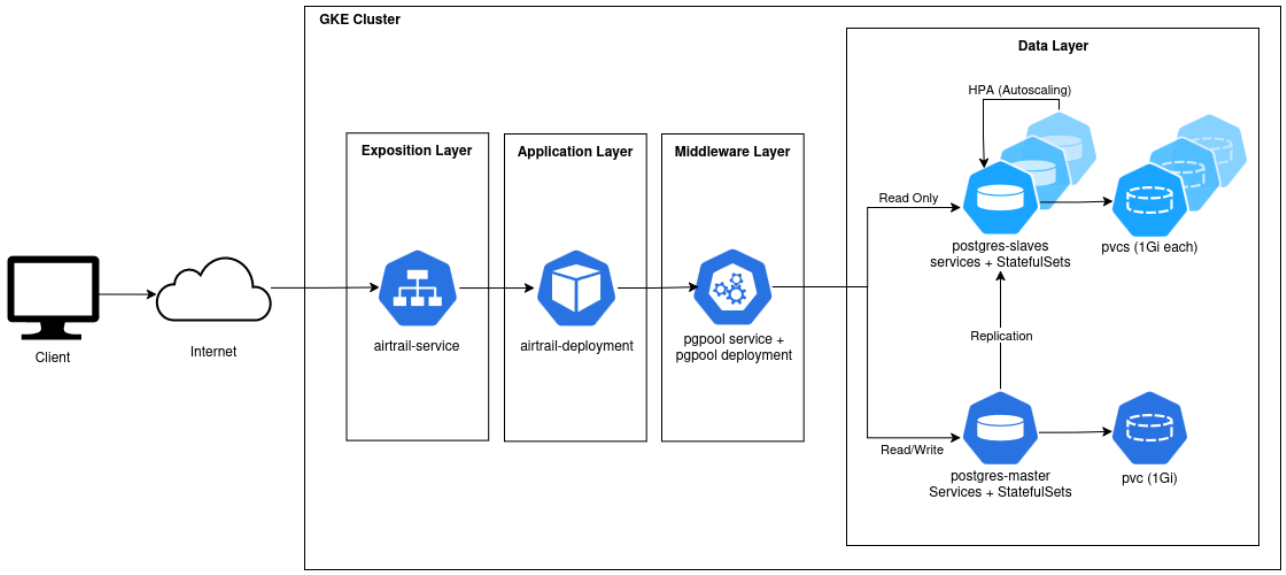


Figure 10: Optimized Deployment Architecture

### 5.1.1. Horizontal Scaling

Our horizontal scaling approach was built upon three key components: the **Primary/Replica architecture** for data redundancy, **Pgpool-II** for query routing, and **HPA** for dynamic load adaptation.

#### 5.1.1.1. Primary/Replica (Master/Slave)

Our solution separates the database into two distinct functional tiers:

1. **The Write Tier (Primary):** We deployed a single-node StatefulSet (postgres-master). Using `volumeClaimTemplates` ensures this node has a dedicated, persistent disk. This node acts as the exclusive "Source of Truth" for all write operations, eliminating the risk of concurrent write corruption.
2. **The Read Tier (Replicas):** We deployed a secondary StatefulSet (postgres-slave) specifically for read-only traffic. Each replica in this set receives its own independent disk via `volumeClaimTemplates` and synchronizes data from the Primary node over the network.

This division enabled safe scaling by decoupling the write and read tiers.

#### 5.1.1.2. Pgpool-II - The Facilitator

To implement this separation, we utilized Pgpool-II. Since the application logic is treated as a black box and cannot be modified, the responsibility for routing different query types had to be delegated to an external component.

Pgpool-II is an open-source middleware that sits between PostgreSQL clients and servers. It acts as a proxy to enhance database performance, availability, and scalability. Crucially, it handles **load balancing** (distributing read queries across replicas) and connection pooling while remaining transparent to the application. This allowed us to separate and balance queries between the Primary and the Replicas without modifying the application code.

#### 5.1.1.3. HPA

The implementation of the Horizontal Pod Autoscaler (HPA) for the Replica tier allows the cluster to automatically scale the number of read-only pods based on CPU demand. This approach is safe because:

- **Storage Isolation:** Each new replica gets its own unique PVC, preventing filesystem conflicts.
- **Traffic Integrity:** The HPA only affects the read-replicas, ensuring that the critical Primary node remains stable and unaffected by the churn of scaling events.

In an application such as AirTrail, flight search operations create a high volume of read traffic, consequently, the HPA effectively manages this increased pressure on the Replica Tier while maintaining performance.

### 5.1.2. Vertical Scaling

In addition to the scale-out (horizontal) strategy, we also implemented vertical scaling to improve the baseline performance of the underlying infrastructure. The node pool configuration was upgraded from the restricted `e2-small` machine type to the more capable `e2-standard-2`. This upgrade provides dedicated vCPUs and increased memory, reducing the likelihood of resource throttling on the database Master.

## 5.2. Optimized Performance

This section presents the performance evaluation results obtained after applying the optimization strategy described in Section 5.1 (Vertical Scaling, Master/Slave Architecture, and HPA). The test window covers two distinct phases: the 1000-thread stabilization test and the 5000-thread stress test, the analysis focuses on the behavior of the AirTrail application and the PostgreSQL database cluster (Master and Replicas) under these high concurrency scenarios.

To provide a consolidated overview, Table 3 summarizes the main resource utilization metrics. It presents CPU usage, memory consumption, and network traffic for both the AirTrail application and the Database tier. This table serves as a reference for the detailed analysis of the 5000-thread stress test discussed in the following subsections. The tests of the optimized deployment, also focus on an optimized machine (vertical scaling) which changes some CPU and memory conceptions.

| Number of Threads | CPU Usage (Total: 6vCPU) | | Memory usage(Total: 24GiB) | | Network Traffic | |
|---|---|---|---|---|---|---|
| | Airtrail | Database | Airtrail | Database | Airtrail | Database |
| 1.000 | Low (0.2 cores) | Low ( 0.9 cores) | Stable (500–600 MiB) | Stable (100-135 MiB) | Moderate | Moderate |
| 5.000 | Stable (0.2–0.4 cores) | Low (0.8–1.1 cores) | Oscillating (750–1000 MiB) | Growing (Master: 160 MiB) | High (1 MiB/s peaks) | High (180 KiB/s peaks) |

Table 3: Optimized Tests Results Summary

### 5.2.1. Application Container (AirTrail)

Figure 11 illustrates the CPU usage of the AirTrail application container during the stress test. The usage profile provides clear evidence of the benefits of vertical scaling. Instead of the erratic behavior often seen in resource-starved environments, the CPU usage remained stable and well within the limits of the upgraded node capacity. This stability indicates that the application logic was not throttled by the underlying infrastructure, allowing it to process incoming requests efficiently without queuing delays or context-switching overhead.

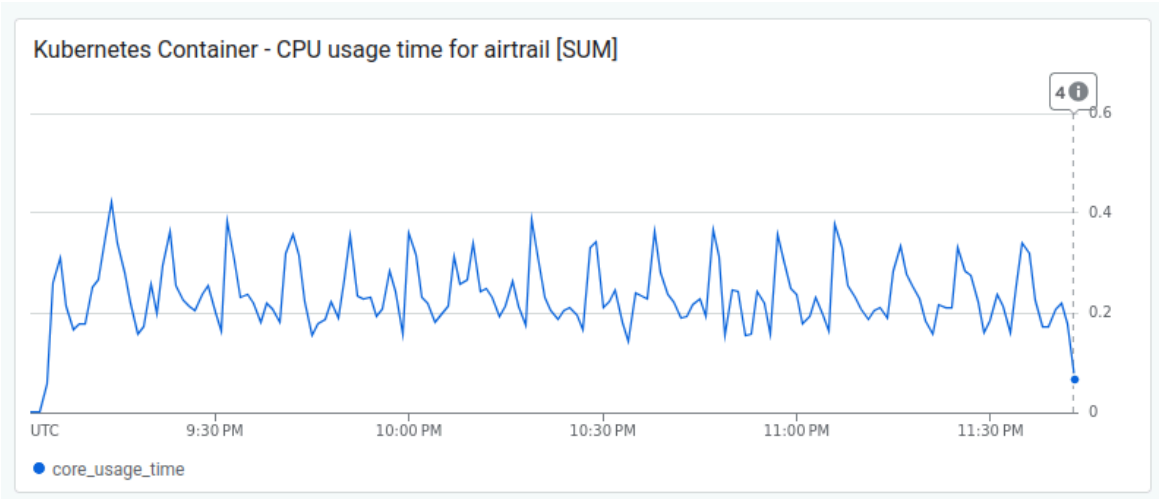


Figure 11: AirTrail container CPU usage during optimized load test

The memory usage, shown in Figure 12, exhibits a distinct oscillating pattern throughout the test duration. This behavior represents the healthy lifecycle of memory management within the application: objects are dynamically allocated to handle user requests and subsequently released when tasks are completed. Crucially, the graph demonstrates that the application utilized a significant amount of RAM to maintain this throughput. In the non-optimized environment, this necessary memory footprint would have exceeded the physical limits of the node, leading to immediate service crashes. The availability of extra memory in the `e2-standard-2` instance ensured continuous operation without forcing the operating system to kill the process.

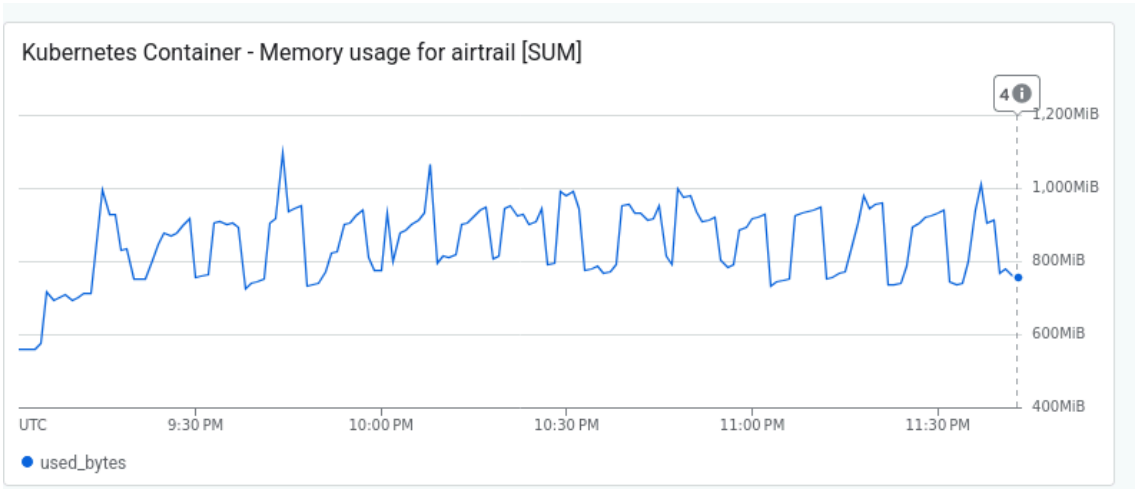


Figure 12: AirTrail container memory usage during optimized load test

### 5.2.2. Database Container (PostgresSQL)

The database layer analysis highlights the successful decoupling of workload types via the Master/ Slave architecture. As shown in Figure 13, both the Primary node and the initial Replica maintained high utilization rates simultaneously. This indicates that the system effectively parallelized the workload: while the Master was fully engaged in processing transactional writes and replication data, the Replica absorbed the heavy lifting of read queries. Had this been a single-node deployment, the combined CPU demand would have saturated the core, creating a severe bottleneck and increasing query latency.

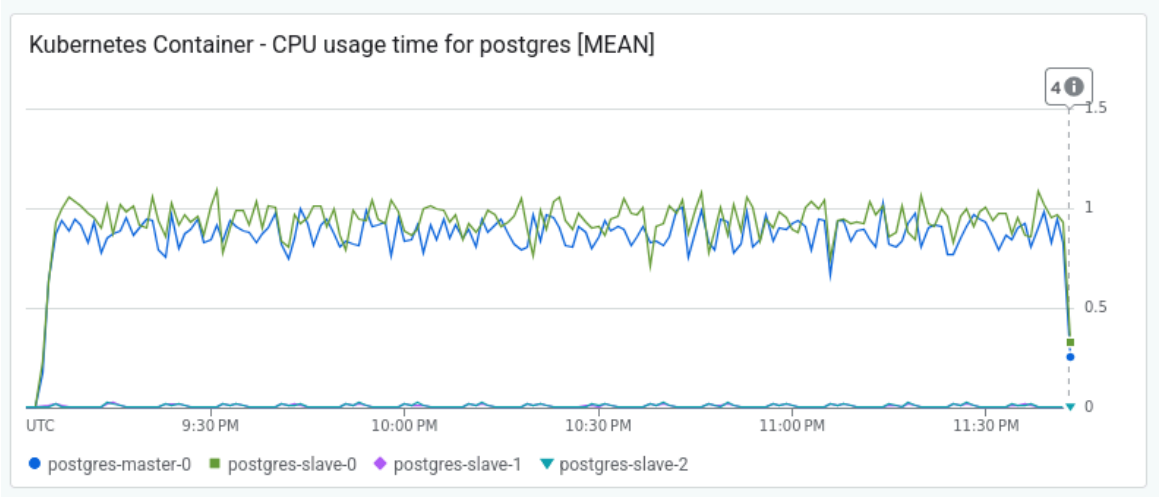


Figure 13: PostgreSQL Cluster CPU usage per pod

Figure 14 serves as the definitive visual confirmation of the Horizontal Pod Autoscaler (HPA) in operation. The graph shows a “stepped” progression where new colored lines (representing additional Slave pods) appear as the test continues and demand increases. This elasticity ensures that the read-replica tier expands dynamically to match the traffic volume. Meanwhile, the Master node shows a steady accumulation of memory usage, reflecting its role in caching writes and managing connections to the expanding fleet of slaves. This separation of concerns protected the Master from being overwhelmed by the high volume of read operations.

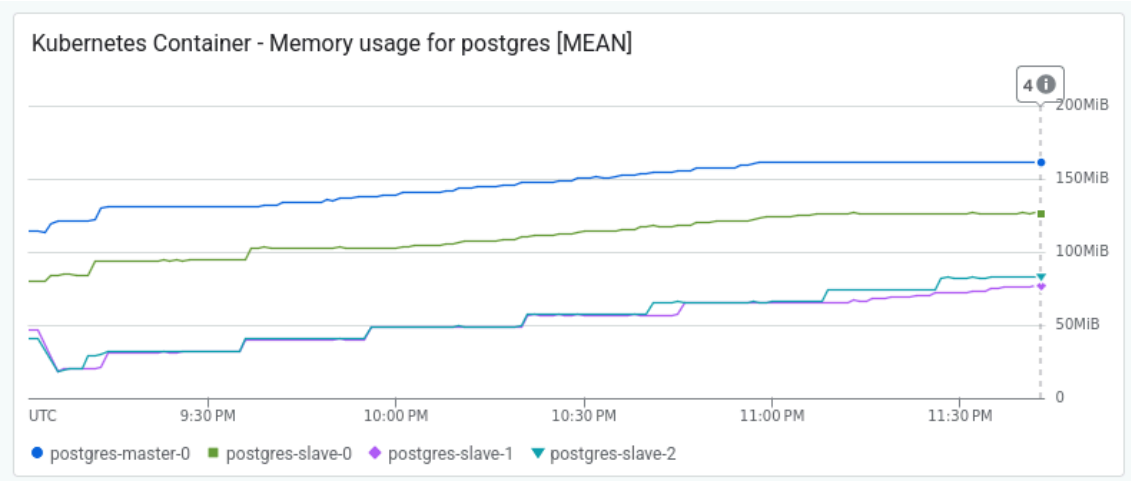


Figure 14: PostgreSQL Cluster memory usage demonstrating HPA scale-out

### 5.2.3. Network Traffic Analysis

Network metrics act as a critical proxy for system availability and I/O throughput. Figure 15 displays the traffic for the AirTrail application, showing a continuous stream of data without interruptions. The periodic peaks correspond to the batched arrival of user requests from the load testing tool. The absence of sudden drops to zero confirms that the application pods remained healthy and responsive throughout the stress test, successfully handling the network load without connection timeouts or resets.

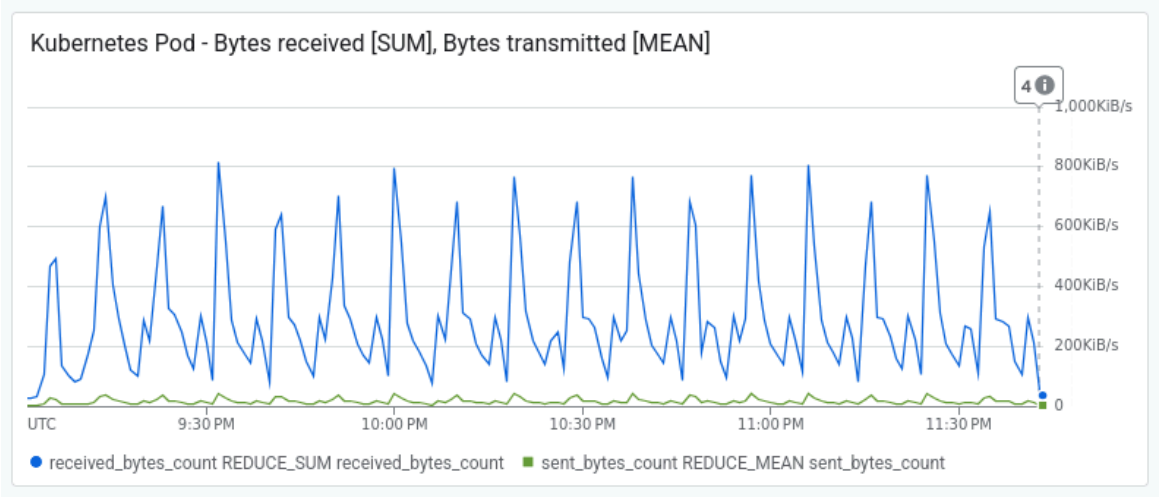


Figure 15: Network traffic for AirTrail pod during optimized load test

Figure 16 provides a granular view of the traffic distribution within the database cluster, validating the routing logic of Pgpool-II. The Master node exhibits the most intense traffic pattern, driven by write operations and the synchronization of data to the replicas. In contrast, the Slave nodes show a more moderate but consistent flow associated with read-only queries. This clear separation confirms that the middleware successfully intercepted and routed queries to the appropriate tier, preventing the Primary node from being saturated by read traffic and ensuring data consistency across the cluster.

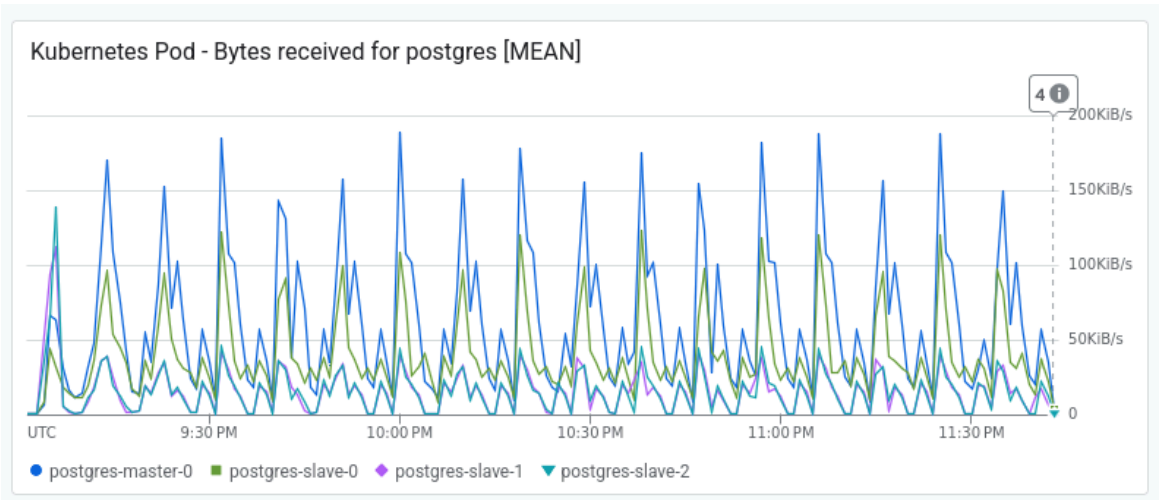


Figure 16: Network traffic per PostgreSQL pod showing Write/Read distribution

## 5.3. Comparative Analysis: Optimized vs. Non-Optimized

After conducting the stress tests on both the baseline (non-optimized) and the final architecture (optimized), we can draw definitive conclusions regarding the impact of the implemented strategies. This section compares the system behavior under the peak load scenario (5000 threads).

The comparison highlights how the combination of Vertical Scaling, Master/Slave replication, and HPA resolved the critical bottlenecks identified in the initial deployment.

### 5.3.1. Resource Contention vs. Sufficient Headroom

The most immediate improvement resulted from the vertical scaling strategy (moving from `e2-small` to `e2-standard-2`).

- **Non-Optimized Scenario:** The baseline environment suffered from severe memory pressure. With only 2GiB of shared RAM for the OS, and the Database, the system operated at the edge of an OOM (Out-Of-Memory) crash. The CPU usage for the database peaked at 1.7 cores, nearly saturating the 2 vCPUs available, leaving no room for system processes.
- **Optimized Scenario:** As evidenced in the results, the optimized application consumed between 750–1000 MiB of RAM. This confirms that the baseline infrastructure was physically less capable of sustaining the application's actual needs. The upgrade provided the necessary headroom for the application to perform healthy cycles without impacting the database performance.

### 5.3.2. Write-Bottleneck vs. Load Distribution

The architectural shift from a single `Deployment` to a `StatefulSet` with Pgpool-II fundamentally changed the database performance profile.

- **Non-Optimized Scenario (Single Node):** In the baseline test, the single PostgreSQL instance was responsible for processing 100% of the traffic (Writes + Reads). This created a single point of failure and a performance bottleneck, as indicated by the high CPU saturation ( 1.7 cores) on a single pod.
- **Optimized Scenario (Master/Slave):** The optimized results show a clear distribution of labor. While the Master node remained active (managing writes), the read-load was effectively offloaded to the Slaves. The network traffic analysis confirms this split: the Master handled the "heavy" write traffic, while the Slaves handled the high-volume read traffic. This separation prevented the write-lock contention that plagued the non-optimized version.

### 5.3.3. Static vs. Dynamic Scalability

The final differentiator is the system's ability to adapt to unexpected load.

- **Non-Optimized Scenario:** The baseline was static. Had the traffic exceeded 5000 threads, the single database node would have become unresponsive, causing a cascading failure across the application.
- **Optimized Scenario:** The inclusion of the HPA introduced resilience. As observed in the memory and CPU graphs of the optimized test, the cluster automatically provisioned new replicas (`postgres-slave-1`, `postgres-slave-2`) as demand increased. This proves that the optimized architecture is not just faster, but also scalable, capable of maintaining performance levels even as the user base grows.

# 6. Final Reflexion

The implementation of this project has demonstrated that automating infrastructure with Ansible is essential for achieving operational reliability. By resolving the hardware limitations (Vertical Scaling) and architectural constraints (Horizontal Scaling), transformed a fragile, resource-starved deployment into a resilient, high-availability system capable of sustaining high concurrency with zero downtime and successfully addressed the core risks of database scaling. This transition was critical in preventing the filesystem corruption that occurs when multiple instances attempt to write to shared storage, ensuring that each node operates with a dedicated, isolated volume while maintaining data consistency across the cluster. The optimization strategy proved successful.

Despite these architectural improvements, our critical analysis identified a significant limitation regarding High Availability (HA). Because the roles of Master and Slave are manually defined within the configuration, the system lacks an automated mechanism for leader election. In a production failure scenario where the Primary node becomes unavailable, the current cluster cannot promote a Slave to the primary position without manual intervention. This “static” role assignment represents a single point of failure that prevents the system from being truly self-healing.

This project serves as a sophisticated bridge between basic container orchestration and production-grade database management. We have built a platform that is stable under load and secure from data corruption; however, the next evolutionary step would be the implementation of a Kubernetes Operator. By moving to an operator-pattern, we could automate the failover process and configuration management, transforming our current manual architecture into a fully autonomous, cloud-native database cluster.