Universidade do Minho
Escola de Engenharia
Masters Informatics Engineering

Cloud Computing Applications and Services Course Unit

Academic Year 2025/2026

Final report

Diogo Rodrigues (PG60244) Miguel Machado (PG60284)
Vicente Martins (PG58812) Simão Alves (PG58810) Tiago Diogo (PG60308)

January 29, 2026

ASCN

Index

List of Figures

1 Introduction

This report describes the work developed within the scope of the *Cloud Computing Applications and Services* course, the objective of which was the installation, automatic configuration, monitoring, and optimization of the AirTrail web application, using the Google Cloud platform, the Google Kubernetes Engine (GKE) service, and the Ansible tool.

The AirTrail application allows users to track completed flights, maintain a travel history, and analyze statistics associated with their flight habits. Throughout this project, the application was adapted for a cloud-native environment, analyzing its performance, scalability, and resilience, as well as applying optimizations based on metrics collected during experimental testing.

1.1 Features

The AirTrail application provides a set of features oriented towards the management and analysis of air travel. In particular, it is possible to:

- Visualize completed flights on an interactive world map, as well as consult the full flight history.
- Add new flights, requiring data such as date, time, origin, and destination.
- Export and import flight data from various external sources, such as MyFlightRadar24, App in the Air, JetLog, and CSV files.
- View and manage visited countries, both automatically (based on registered flights) and manually.
- Utilize OAuth authentication, enabling integration with external identity providers (e.g., Azure AD, Google), ensuring centralized and secure authentication.
- Consult statistics regarding the user's flight habits.
- Responsive design allowing the application to be used across a wide range of devices.

1.2 Architecture and Components

The AirTrail application features a monolithic architecture based on the client-server model:

- **Backend (API Server):** Implemented in Node.js with TypeScript, it provides REST endpoints used by the frontend and supports features such as authentication, flight data importation, and database access. The backend supports authentication via OAuth/OIDC, allowing integration with external identity providers, as well as authentication through an API key used as a *Bearer token*.
- **Frontend (Web Interface):** An SPA (*Single Page Application*) developed with SvelteKit, a modern framework based on Svelte and TypeScript. This component provides a responsive and intuitive interface for flight visualization and management, communicating with the backend via HTTP/JSON.
- **Database:** AirTrail utilizes a PostgreSQL database for persistent data storage, including information on users, flights, tokens, and settings. Interaction with the database is performed through the Prisma ORM.

It is noteworthy that the application is distributed using Docker and *docker-compose*, allowing the orchestration of the different application components, which facilitates automated deployment in cloud environments such as Google Kubernetes Engine (GKE).

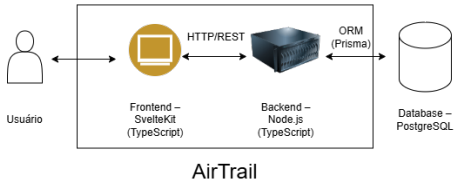

Figure 1.1: Illustration of the AirTrail application's monolithic architecture.

1.3 APIs

The AirTrail application provides a REST API used by the frontend for managing user flights. Among the main endpoints provided, the following stand out:

- `GET /flight/list` — Displays a list of all the user's flights.
- `GET /flight/get/{id}` — Retrieves the details of a specific flight by *id*.
- `POST /flight/save` — Creates a new flight or updates an existing one if the *id* is present.
- `POST /flight/delete` — Removes an existing flight using its respective *id*.

2 Potential Performance Bottlenecks and Single Points of Failure

2.1 Potential Performance Bottlenecks

The analysis of the AirTrail application, considering its base architecture, allows for the identification of several components susceptible to becoming performance bottlenecks as the workload increases.

Firstly, the **application server**, implemented in Node.js, constitutes a potential bottleneck. Since the backend is responsible for processing HTTP requests, executing business logic, and interacting with the database, an increase in the number of simultaneous requests can lead to resource contention such as CPU and memory. This situation is particularly relevant in scenarios involving frequent database access or calls to external APIs, which can result in an increase in the application's global latency.

The **PostgreSQL database** represents another potential performance bottleneck. As a single instance, the application relies heavily on read and write operations performed on this component. Under peak loads, CPU and disk I/O saturation may occur, as well as locks resulting from concurrent operations, leading to performance degradation and increased response times.

Finally, **communication with external APIs** used to obtain flight-related information can negatively impact application performance. The latency or temporary unavailability of these APIs directly affects the response time of certain features, especially in the absence of caching mechanisms or fault control.

2.2 Single Points of Failure

In addition to performance bottlenecks, the application's base architecture also presents several single points of failure that compromise its resilience.

The existence of a **single instance of the application server** constitutes a critical point, as any failure at the Node.js process level, the node where it is executing, or during maintenance and update operations, results in the total unavailability of the application for users.

Similarly, the **PostgreSQL database** sets up a significant single point of failure. In the absence of replication or *failover* mechanisms, a failure in this component prevents data persistence and recovery, compromising the functioning of the application as a whole.

Additionally, the dependence on **external APIs** introduces vulnerabilities related to the availability of third-party services. Temporary failures in these APIs can compromise specific features of the application. Although the application core may remain functional, the absence of mechanisms such as *circuit breakers* or local caching limits tolerance to external failures.

3 Installation and Automatic Configuration

3.1 Tools Used

The installation and configuration of the AirTrail application were carried out using a set of cloud computing and automation tools. Specifically, the following were used:

- **Google Cloud Platform (GCP)** served as the cloud infrastructure and provided monitoring mechanisms that were utilized in later stages.
- **Google Kubernetes Engine (GKE)** allowed for the creation and orchestration of containers to host the application and the various components required for its operation. For the installation and configuration, different Deployment and Service pods were created; the Deployment pods contain the containers necessary for the program's execution.
- **Ansible** was used to automate the provisioning, configuration, and deployment process of the application through communication with the GKE tool. Additionally, Ansible Vault was utilized to add a layer of security to our solution by encrypting and protecting passwords.
- **Docker** was used for the containerization of the different components; we utilized Docker Images for both PostgreSQL and AirTrail itself.
- **Postman** was used to test and build HTTP requests for AirTrail.
- **JMeter** was used to test AirTrail's behavior under multiple simultaneous HTTP requests.

3.2 Automation Approach

The Ansible playbooks provided by the teaching staff were used as a foundation and extended to automate the creation of the Kubernetes cluster, the deployment of the different application components, and their configuration in a cloud environment.

To support this architecture, the automation was structured into distinct components orchestrated by Ansible, beginning with the provisioning of the data layer. This configuration ensures that critical user and flight data survive component restarts or failures, with database access managed internally via a `ClusterIP` service, restricting communication to cluster components.

Next, the AirTrail application is deployed, which includes the configuration of a Horizontal Pod Autoscaler (HPA) to ensure automatic system scalability under processing load, and the creation of a `LoadBalancer` service. Managing the `ORIGIN` variable presented a particular challenge, as it depends on the public IP address dynamically assigned by Google Cloud, which is unknown at the initial moment of deployment.

To resolve this dependency, our playbooks implement an active polling mechanism that waits for the external IP to be assigned to the LoadBalancer. Once this IP becomes available, it is automatically captured and injected into the application configuration through a new rollout. This step ensures that the `ORIGIN` variable correctly reflects the public address, enabling the proper functioning of the web application's security and CORS mechanisms. Finally, the automation process concludes with the configuration of monitoring, using the Google Cloud CLI to generate custom dashboards that allow for immediate visualization of the system status.

Thus, the solution ensures a "zero-touch" installation where the entire environment is correctly configured and interconnected with a single command, from the persistent database to the publicly accessible scalable application.

3.3 Data Persistence and Service Exposure

To ensure the durability of critical application data against the volatile lifecycle of containers, we utilized the `PersistentVolumeClaim` (PVC) mechanism. We defined a 10Gi storage request with `ReadWriteOnce` access mode, which delegates the dynamic provisioning of a Persistent Disk to Google Kubernetes Engine (GKE). This volume is mounted to the PostgreSQL container, ensuring that all user and flight records survive restarts, failures, or updates of the database Pods.

Regarding application exposure, we adopted a two-tier approach to maximize security and accessibility. The database is exposed exclusively within the cluster via a `ClusterIP` service, preventing any direct access from the public internet. Conversely, the AirTrail application is exposed externally through a `LoadBalancer` service. This service not only provides a dedicated public IP address but also performs port mapping, redirecting traffic from the standard HTTP port (80) to the application's internal port (3000), simplifying final access for the user.

4 Monitoring and Metrics

The monitoring of the AirTrail application was carried out using the tools provided by the Google Cloud platform, specifically Google Cloud Monitoring. This tool provides metrics, events, and metadata presented through the collection of information from the components involved. It identifies issues and uncovers patterns, helping to evaluate the user experience and facilitating the analysis of the implementation's quality.

For our solution, we chose to implement a custom dashboard in Google Cloud Monitoring (Stackdriver), utilizing line charts and numerical scorecards. This approach allows for real-time visualization of the cluster status and critical AirTrail application components.

The selected metrics primarily focus on two levels:

- `Cluster Level:` Visualization of the global CPU and memory utilization by cluster nodes, total number of running pods, and log volume (throughput).
- `Application Level:` Granular monitoring of the `airtrail` (web application) and `postgres-container` (database) containers. For each, we compare actual CPU and Memory usage against the requests and limits defined in Kubernetes (`core_usage_time` vs. `request_cores/limit_cores` and `used_bytes` vs. `request_bytes/limit_bytes`).

We opted to create this dedicated dashboard since standard GCP dashboards (GKE Dashboard) present excessive information density and require constant manual filtering to isolate our project's components.

The automation of this process was ensured via Ansible in the monitoring role. This role defines a task that invokes the `gcloud monitoring dashboards create` command, utilizing a JSON file that describes the entire layout structure and the necessary MQL (Monitoring Query Language) queries.

In this way, monitoring allowed us not only to validate the correct provisioning of resources (verifying that limits were not being reached) but also to proactively detect potential performance bottlenecks in the database and web server components during load testing.

5 Experimental Evaluation

The objective of the experimental evaluation was to analyze the behavior of the AirTrail application under different load levels. To this end, we implemented a benchmarking mechanism using the JMeter tool and the monitoring provided by GCP.

5.1 Postman

To test the application's endpoints, we used the Postman API platform to understand the structure of the HTTP requests. We began by manually creating several flight records in AirTrail and then executed the following GET request: `"http://{airtrail_ip}/api/flight/list"`. Through this, we obtained relevant information such as: `"fromId"`, `"toId"`, `"from"`, `"to"`, `"icao"`, and `"userId"`. These fields are crucial for the creation test of a flight via an HTTP POST request: `"http://{airtrail_ip}/api/flight/save"`.

5.2 JMeter

In our solution, we used the JMeter tool to develop a synthetic set of HTTP requests for the page `http://{airtrail_ip}:{airtrail_port}/`.

We started by defining a *Thread Group* representing the benchmark clients that perform HTTP POST requests. After defining the benchmark clients, we implemented a timer for the requests. By varying the number of threads in the *Thread Group* and/or the interval (ms) between requests, we aimed to visualize the behavior and variation of the monitored cluster metrics through the previously defined monitoring tools.

For the first evaluation, we defined that a given benchmark client would execute 150 requests to the aforementioned URL every 300ms for 1 minute. During the observed period, we verified an increase in the CPU usage percentage across all nodes and, as expected, an increase in logging throughput.

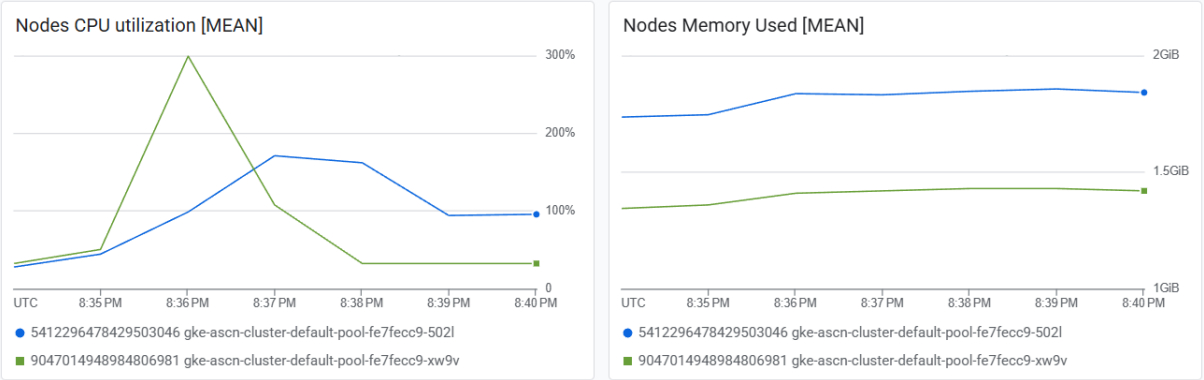

Figure 5.1: Cluster metrics during the JMeter test.

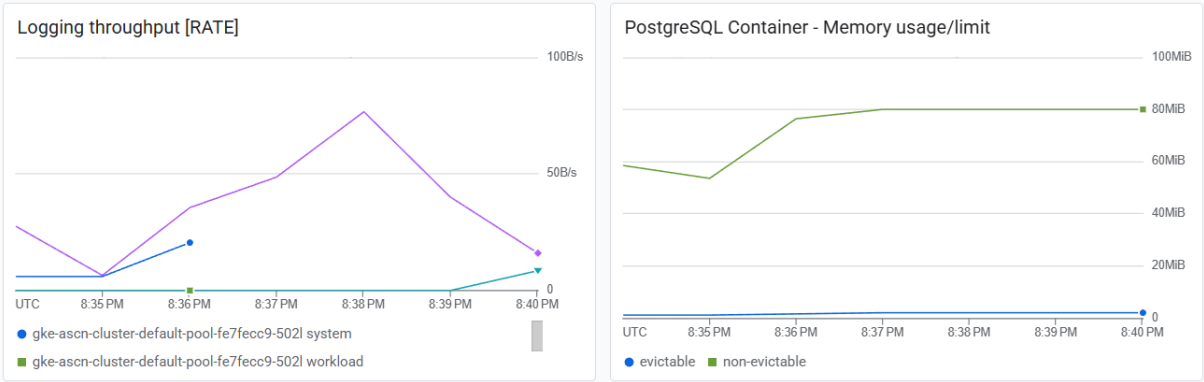

Figure 5.2: Logging throughput and database memory usage with requests performed by JMeter.

By observing the amount of CPU utilized by the `airtrail` container, we noticed that the requests forced the utilization of the total CPU allocated to the container. Thus, it is possible to conclude that requests accumulated in the queue during the iterations, which consequently led to a drastic increase in the CPU required for processing.

We also noticed that during the benchmark execution, the `airtrail` container became unstable and restarted cyclically. This occurs because upon reaching 100% CPU utilization, the application can no longer respond in a timely manner to Kubernetes *Liveness Probes*. When the orchestrator detects that the Pod is unhealthy, it forces its termination (observed in the `Terminating` state) and automatically creates a new instance (`Running` state) to ensure service availability, which is a *self-healing* mechanism. However, as the test load remained constant, the new container would quickly saturate, repeating the process.

Therefore, we can determine that to support this load, it would be necessary to increase the CPU values requested by the container (*requests* and *limits*) or implement horizontal scalability (HPA). Although this stress test with JMeter pushes the system to its limit artificially, it was essential to validate GKE's automatic recovery capability in the face of critical resource failures.

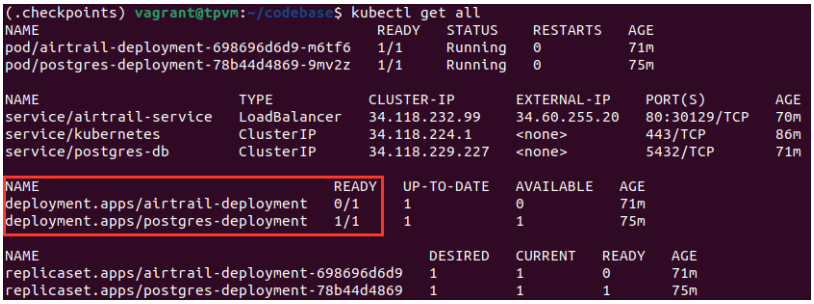

Figure 5.3: Pod termination mechanism due to Liveness Probes.

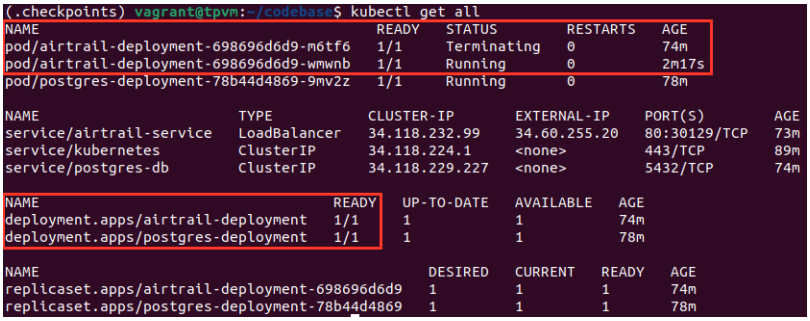

Figure 5.4: Kubernetes self-healing mechanism terminating and creating new Pods.

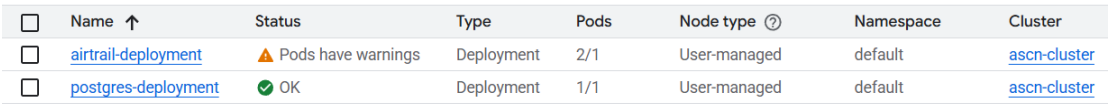

Figure 5.5: Kubernetes self-healing mechanism via Google Cloud Console.

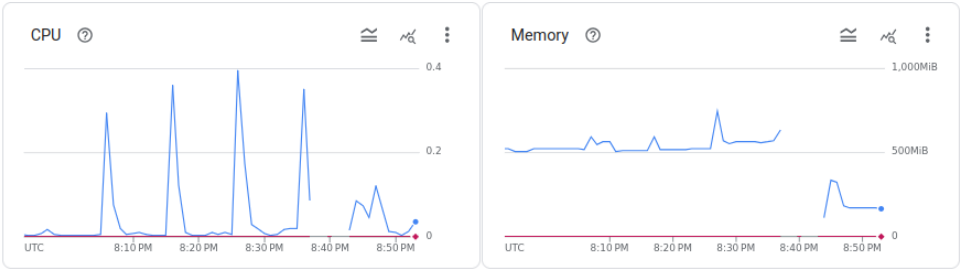

Figure 5.6: Graphical visualization of the self-healing mechanism.

6 Results Analysis

The analysis of the obtained results demonstrates that the applied optimizations had a positive impact on the performance and resilience of the AirTrail application. Compared to the base installation, the optimized version showed a better load distribution among the backend replicas and a more efficient utilization of available resources.

Furthermore, a reduction in response time variability and greater tolerance to load spikes were observed, highlighting improvements in the application's horizontal scalability.

This behavior was ensured by the implementation of the *Horizontal Pod Autoscaler* (HPA), which allowed for the dynamic adjustment of the number of replicas in real-time, maintaining the average CPU consumption close to the defined target of 75%. During the load tests with JMeter, the HPA proved to be fundamental in absorbing sudden traffic spikes, scaling the application up to a maximum of 4 replicas whenever demand required.

Additionally, the physical infrastructure was reinforced by changing the machine type of the GKE *cluster* nodes from *e2-small* to *e2-medium*. This transition was decisive, as the *e2-small* instances had severe memory (RAM) and CPU limitations, making it impossible to simultaneously schedule multiple application replicas and the PostgreSQL database.

After introducing the Horizontal Pod Autoscaler (HPA), new load tests were conducted to evaluate the application's behavior under high levels of concurrency and to validate the effectiveness of automatic scaling. To this end, two distinct scenarios were used in JMeter: a test with 300 threads executing POST requests and a second test with 5000 threads executing GET requests.

In the scenario with 300 concurrent POST requests, the tests were run for a period of 1 minute, with a fixed interval of 300 ms between requests, as defined in the previous chapter. This configuration allows for generating a continuous and controlled load, simulating multiple users submitting flight creation requests almost simultaneously without causing an artificial instantaneous peak.

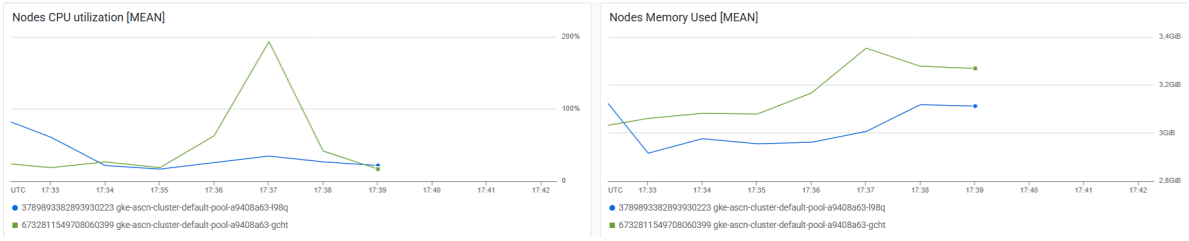

Figure 6.1: Cluster metrics during the 300-thread JMeter test.

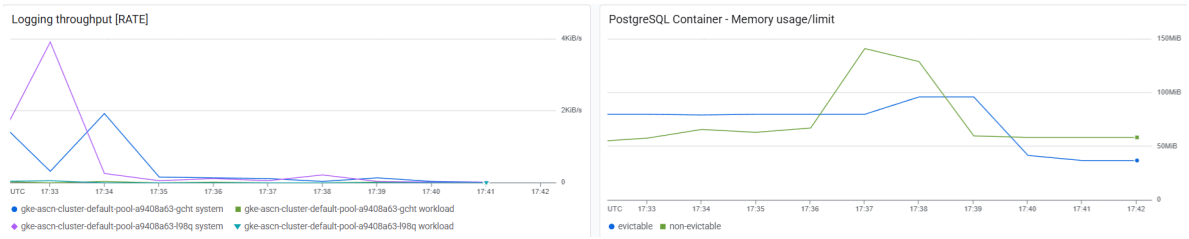

Figure 6.2: Logging throughput and database memory usage with requests performed by JMeter.

During the test execution, a progressive increase in CPU utilization was observed across the cluster nodes, as shown in the presented graphs. When the average CPU utilization reached the threshold configured in the Horizontal Pod Autoscaler (75%), Kubernetes automatically activated the horizontal scaling mechanism, proceeding to create new backend replicas.

This behavior allowed the write load to be distributed across several pods, reducing the pressure on each individual instance. Consequently, the instability previously observed in the tests without HPA was avoided, namely the frequent restarts caused by liveness probe failures. The CPU and memory utilization graphs show a controlled rise followed by stabilization, indicating that the system was able to adapt dynamically to the imposed load.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	656	27102	1010	49436	10331.30	45.73%	10.9/sec	14.30	5.73	1341.1
TOTAL	656	27102	1010	49436	10331.30	45.73%	10.9/sec	14.30	5.73	1341.1

Figure 6.3: JMeter 300-thread metrics.

In the second scenario, with 5000 threads performing GET requests for 1 minute, representing a pattern of intensive read operations, a very high volume of traffic directed at the application was observed. Despite this significant load, the system remained operational and stable. The LoadBalancer service distributed requests evenly among the various available replicas, while the HPA dynamically adjusted the number of pods based on the average CPU utilization.

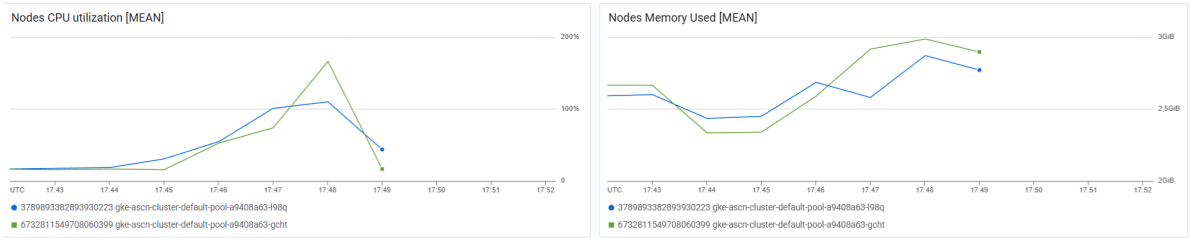

Figure 6.4: Cluster metrics during the 5000-thread JMeter test.

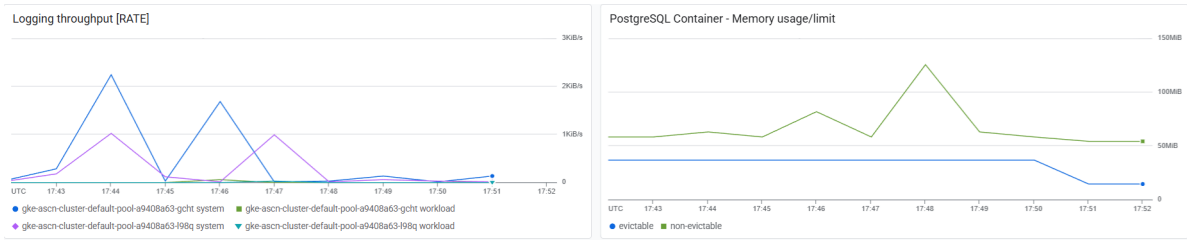

Figure 6.5: Logging throughput and database memory usage with requests performed by JMeter.

The analysis of the graphs reveals a temporary increase in CPU and memory utilization on the cluster nodes, followed by stabilization, indicating that the scaling mechanism responded adequately to the imposed load. An increase in logging throughput is also observed, consistent with the high number of requests processed in a short period.

Despite the intensity of the test (5000 requests in one minute), no failures, pod restarts, or significant service degradation occurred. This behavior confirms that the combination of LoadBalancer and HPA effectively absorbs high traffic spikes, ensuring service continuity and stable response times even under heavy loads.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	34017	9158	1	15552	2281.26	44.56%	526.0/sec	4052.99	563.44	7890.5
TOTAL	34017	9158	1	15552	2281.26	44.56%	526.0/sec	4052.99	563.44	7890.5

Figure 6.6: JMeter 5000-thread metrics.

7 Final Reflection

The work developed allowed for the practical application of the fundamental cloud computing concepts covered in the course, namely containerization, orchestration with Kubernetes, deployment automation, monitoring, and the performance evaluation of distributed applications.

The main strengths highlighted include the complete automation of the installation and configuration process of the AirTrail application using Ansible and Google Kubernetes Engine. This resulted in a reproducible, scalable solution aligned with *cloud-native* best practices. The implementation of monitoring mechanisms through Google Cloud Monitoring, combined with load testing using JMeter, allowed for the analysis of application behavior under stress scenarios, showcasing mechanisms such as *self-healing*, automatic scalability, and recovery from resource saturation.

Regarding areas for improvement, the application's monolithic architecture and the use of a single PostgreSQL database instance remain as limitations in terms of independent scalability and resilience. This was particularly evident during load testing, where the database remained a non-horizontally scalable component. The introduction of a microservices-based architecture and database replication mechanisms constitute potential future developments.

Regarding deviations from the initial plan, it was necessary to adjust the automation strategy, specifically concerning the dynamic management of the public IP address assigned by the LoadBalancer and the configuration of environment variables dependent on that address. Despite these deviations, the final solution proved to be more robust and technically complete. These adjustments reflected the need to adapt to the real-world challenges of dynamic cloud environments, which were not fully predictable during the initial planning phase.

In summary, the project contributed to the consolidation of technical knowledge and to the understanding of the inherent challenges in developing and operating applications in cloud environments.