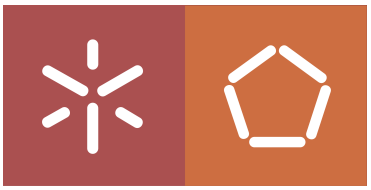

Universidade do Minho
Escola de Engenharia

Ano Letivo 2025/2026

AirTrail: Implementação Kubernetes

Aplicações e Serviços de Computação em Nuvem

Grupo 11

MEI - 1º Ano - 1º Semestre

Trabalho realizado por:

PG61130 - Ana Margarida Pires
PG58805 - Diana Dinis
PG61522 - Francisco Mota
PG59966 - João Silva
PG61551 - Manuel Fernandes

Braga, 17 de janeiro de 2026

Índice

1. Introdução

A *Cloud* está cada vez mais presente na vida de todos os utilizadores de aplicações, uma abordagem preferida pela maior facilidade de configuração pelo lado do cliente e melhor controlo de versões.

A utilização de *scripts* no desenvolvimento de aplicações em nuvem é fundamental para se garantir que os testes realizados sejam repetíveis e reproduzíveis, garantindo uma maior consistência em todo o processo.

O presente relatório descreve o projeto *Airtrail*, uma aplicação *web* com *endpoints* HTTP para gestão de utilizadores e voos, e a respetiva automação de *deployment* e testes, utilizando Kubernetes e Ansible.

1.1. Enquadramento

Este relatório final pertence ao trabalho prático de Aplicações e Serviços de Computação em Nuvem, da Universidade do Minho.

1.2. Estrutura

O documento divide-se em quatro capítulos principais:

- **Aplicação**, onde se discutirá a aplicação *Airtrail* e tudo que a compõe;
- **Arquitetura**, onde será mostrada a implementação em *scripts* de Ansible;
- **Testes**, onde se mostrará alguns resultados de *benchmark*
- **Conclusão**, onde se discutirá a globalidade do projeto, assim como trabalho futuro.

2. Aplicação Airtrail

AirTrail é uma aplicação que permite aos seus utilizadores rastrear voos e consultar o próprio histórico de viagens.

Pela sua descrição, as funcionalidades da aplicação são as presentes na Tabela 1.

Tabela 1: Funcionalidades da aplicação AirTrail

Funcionalidade	Descrição
Mapa Mundial	Ver os voos todos num mapa interativo
Histórico de Voo	Manter o histórico dos voos num lugar apenas
Estatísticas	Obter percepções do histórico
Múltiplos utilizadores	Gerir múltiplos utilizadores, partilha de dados entre estes, com autenticação e integração com OAuth
Importar voos	Importar voos de várias fontes

A aplicação destaca-se pela sua forma dinâmica de adicionar voos, permitindo que os utilizadores registem facilmente novas viagens de forma rápida e intuitiva. Também é de notar o *design* criativo de rastrear as viagens pelo mapa do mundo interativo, que torna o uso da aplicação mais apelativo.

2.1. Requisitos

A aplicação tem como requisitos:

- **Git** → Gestor de versões;
- **Bun** → Gestor de pacotes de JavaScript;
- **Node.js** → Ambiente de JavaScript;
- **PostgresSQL** → Motor de base de dados;

2.2. Arquitetura e Principais Componentes

Na Figura 1 está presente um esquema dos principais componentes da aplicação AirTrail:

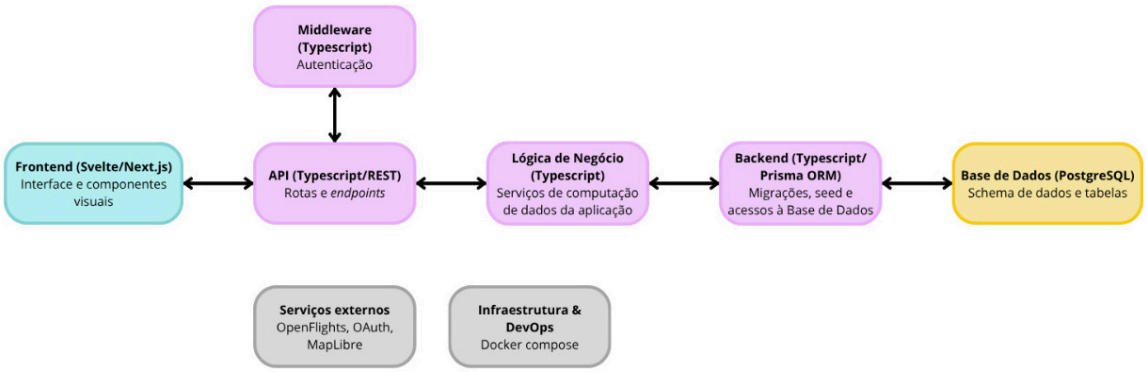

Figura 1: Arquitetura e principais componentes

Fundamentalmente, pode ser resumido em:

- **Frontend:** Interface em Svelte
- **Backend/API:** Implementado em TypeScript com Prisma ORM
- **Base de Dados:** PostgreSQL, gerido num container isolado (airtrail_db)
- **Docker Compose:** Orquestra os containers definidos em **docker/production** e **docker/development**

2.3. APIs

As rotas da API estão localizadas em **src/lib/server/routes/**

Os principais *endpoints* são os seguintes, na Tabela 2:

Tabela 2: Principais *endpoints* da aplicação AirTrail

Categoria	Endpoint	Método	Descrição
Aircraft	/api/aircraft/save/form	POST	Inserção um avião pelo formulário
Airlines	/api/airline/save/form	POST	Criação de companhia aérea por formulário
Airports	/api/airport/save/form	POST	Inserção de aeroportos por formulário
Flights	/api/flight/get/:id	GET	Retorna voo por ID
Flights	/api/flight/list	GET	Lista de voos
Flights	/api/flight/delete	DELETE	Remoção de voo
Flights	/api/flight/save/form	POST	Inserção de um novo voo
User	/api/users/login	POST	Autenticação de utilizadores
User	/api/users/add	POST	Criação de novo utilizador
Statistics	/api/visited-countries	GET	Consulta de países visitados

Estas APIs são invocadas internamente pelo *frontend*, garantindo uma separação entre a interface do utilizador e a lógica de negócio, como já visto na Figura 1.

3. Arquitetura Solução

A implementação da aplicação foi com recurso ao *Google Kubernetes Engine* (GKE) da *Google Cloud*, através da utilização de Ansible, permitindo automatizar os processos de *deploy* e *undeploy*, conseguindo-se tornar em testes repetíveis e reproduzíveis, uma das bases de DevOps.

3.1. Estrutura

A estrutura do repositório é a seguinte:

```
GrupoTP-11
├─ airtrail-deploy.yml                # deploy (DB + WebApp)
├─ airtrail-undeploy.yml              # undeploy (delete_data="true" apaga DB/PVC)
├─ gke-cluster-create.yml             # cria cluster GKE
├─ gke-cluster-destroy.yml            # destrói cluster GKE
├─ test-all.yml                       # testes end-to-end
├─ monitoring-setup.yml               # configura monitorização
├─ run-benchmark.yml                  # executa testes jmeter + relatorios
│
├─ inventory/
│  ├─ gcp.yml         # vars GCP/GKE + endpoint (app_ip/app_port) + vars de teste
│  └─ group_vars/all.yml # vars globais (imagem/porta webapp, DB_URL/credenciais)
│
├─ roles/
│  ├─ airtrail-app/tasks/main.yml # deploy webapp + espera IP externo + atualiza
│  │                              # inventory
│  ├─ airtrail-db/tasks/main.yml # deploy PostgreSQL (PVC + deployment + service)
│  ├─ gke_cluster_create/tasks/main.yml        # cria cluster
│  ├─ gke_cluster_destroy/tasks/main.yml       # apaga cluster
│  ├─ jmeter_local_setup/tasks/main.yml        # instala o Java + motor do JMeter
│  ├─ jmeter_local_setup/vars/main.yml         # versões + caminhos de instalação
do JMeter
│  ├─ jmonitoring_setup/tasks/main.yml         # dashboards + métricas e
integração com Google Cloud Monitoring
│  ├─ test_airtrail/test_service/tasks/main.yml # teste HTTP ao serviço (GET /)
│  ├─ test_airtrail/account_setup/tasks/main.yml # setup/login + cria API key
│  ├─ test_airtrail/flight_creation/tasks/main.yml # cria voo + guarda flight_id
│  ├─ test_airtrail/flight_info/tasks/main.yml # consulta voo por ID
│  ├─ test_airtrail/templates/jmeter/login_test.jmx.j2 # teste carga login
│  ├─ test_airtrail/templates/jmeter/mixed_load.jmx.j2 # teste fluxo completo
│  └─ test_airtrail/templates/jmeter/search_flights.jmx.j2 # teste consulta voos
│
└─ templates/                         # Kubernetes (Deployment/Service/PVC)
   ├─ airtrail-deployment.yml         # Deployment webapp (imagem/porta/env DB_*)
   ├─ airtrail-hpa.yml                # Implementação HPA no webserver
   ├─ airtrail-service.yml            # Service webapp (LoadBalancer → IP externo)
   ├─ postgres-deployment.yml         # Deployment PostgreSQL (POSTGRES_* + volume)
   ├─ postgres-service.yml            # Service PostgreSQL (ClusterIP interno)
   └─ postgresql-pvc.yml              # PVC (storage persistente)
```

3.2. Avaliação inicial de escalabilidade

Inicialmente, a aplicação Airtrail vai ser desenvolvida com arquitetura *multi-tier*, contendo apenas o *service*, do tipo *Load Balancer*, o *web app* em um *pod* e a base de dados em outro *pod*, como demonstra a Figura 2:

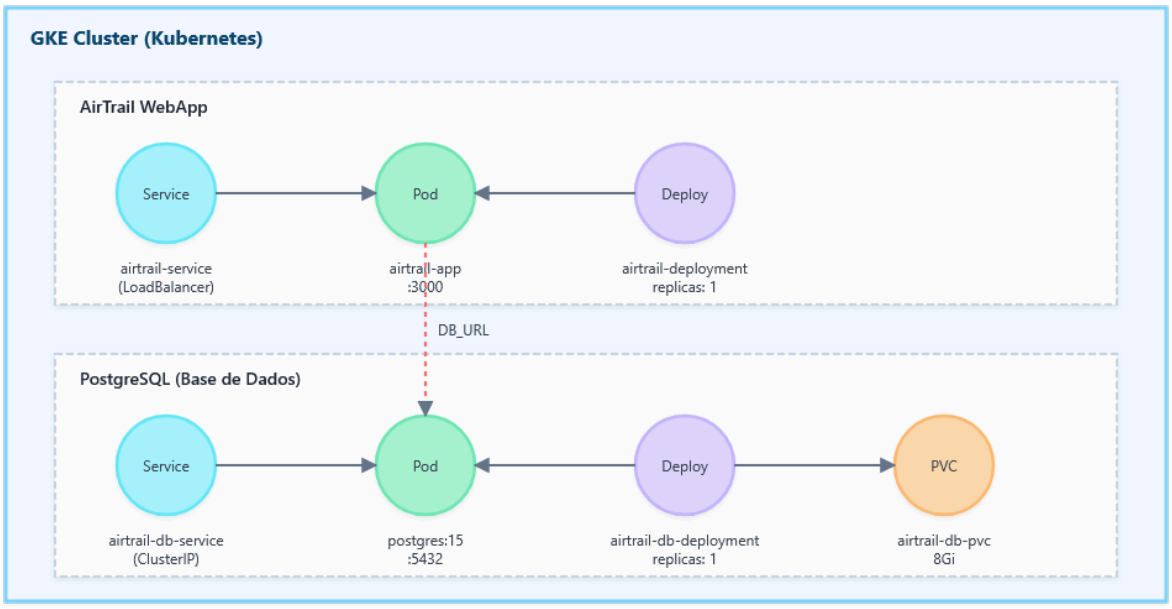

Figura 2: Arquitetura em Kubernetes com HPA

Posto isto, há vários problemas que podem ser encontrados quando se usaria uma solução estática como a demonstrada, estando divido em gargalo de desempenho e ponto único de falha:

Gargalo de desempenho

Web App:
→ Apenas 1 réplica fixa, não aguentando picos
→ CPU limitada
→ Saturação sob carga elevada

Base de Dados:
→ Limitações de I/O do disco
→ Conexões limitadas

Ponto único de falha

Web App:
→ Se o pod falhar, aplicação fica indisponível
→ Restart manual necessário
→ Dependência de um único nó

Base de Dados:
→ *Downtime* durante falhas
→ Sem alta disponibilidade

3.3. Solução escalável

Com a quantidade de pedidos de utilizadores, é de esperar que a CPU comece a ficar sobrecarregada.

O *Horizontal Pod Autoscaler* (HPA) é uma ferramenta de *Kubernetes* que permite ajustar automaticamente o número de réplicas de um *pod* para uma *workload* específica, tendo em conta métricas observáveis, como a utilização da CPU. Permite regular os recursos, garantindo o suporte durante muitos acessos e poupando recursos quando não é preciso tanto poder computacional.

Foi utilizado HPA para replicas da *web app*, de forma a conseguir garantir o desempenho conforme o aumento do número de utilizadores da aplicação, tendo como *threshold* a média de utilização da CPU acima dos 75%. Isto permitirá resolver o problema da *web app* ser um gargalo de desempenho, quando há muitos acessos de utilizadores, sem cargas elevadas para a base de dados.

Na Figura 3 está presente a arquitetura, ainda *multi-tier*, no caso de haver a necessidade para dois pods da *web app*:

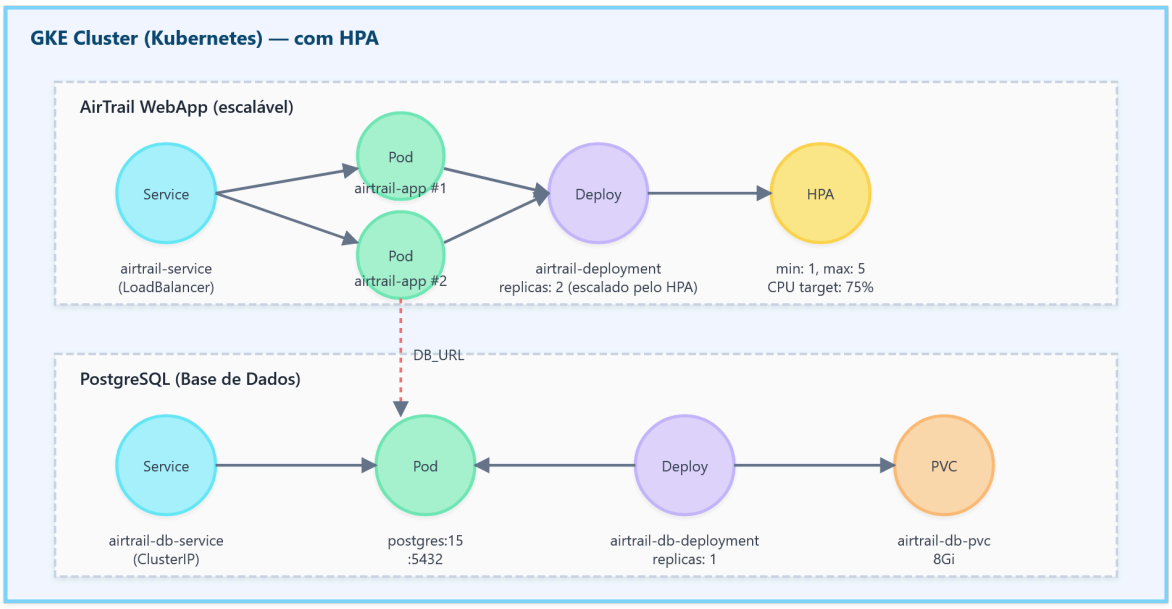

Figura 3: Arquitetura em Kubernetes com HPA

4. Testes

Os testes de carga foram realizados utilizando a ferramenta JMeter, lecionada durante as aulas práticas.

Foram testados os seguintes *endpoints*:

- `/api/users/login`
- `/api/flight/list`

Nos seguintes casos:
1- Teste individual ao Login: Avaliação da capacidade de entrega da página inicial de autenticação.
2- Teste individual à Procura de Voos: Medição da *performance* na consulta de registos de voos.
3- Teste de sequência (*Mixed_Load*): Teste de fluxo de utilizador que executa três pedidos em ordem para cada utilizador: entrada na *home*, navegação para *login* e consulta de dados. Fluxo completo onde o utilizador fz *login* e consulta a lista de voos.

Há três cenários diferentes de *benchmark*, sendo o *light*, *medium* e *heavy*. Os valores *default* estão presentes na :

	Light	***Medium***	***Heavy***
Threads	10	50	100
Duration	60	120	180
Ramp_Time	10	30	60

A dashboard realizada no *Monitoring* da *Google Cloud* está presente na Figura 4, que contém a percentagem média de pedidos de CPU, a utilização de memória, o número total de replicas no momento do `airtrail-app` e gráficos de pedido de CPU e utilização de memória ao longo do tempo:

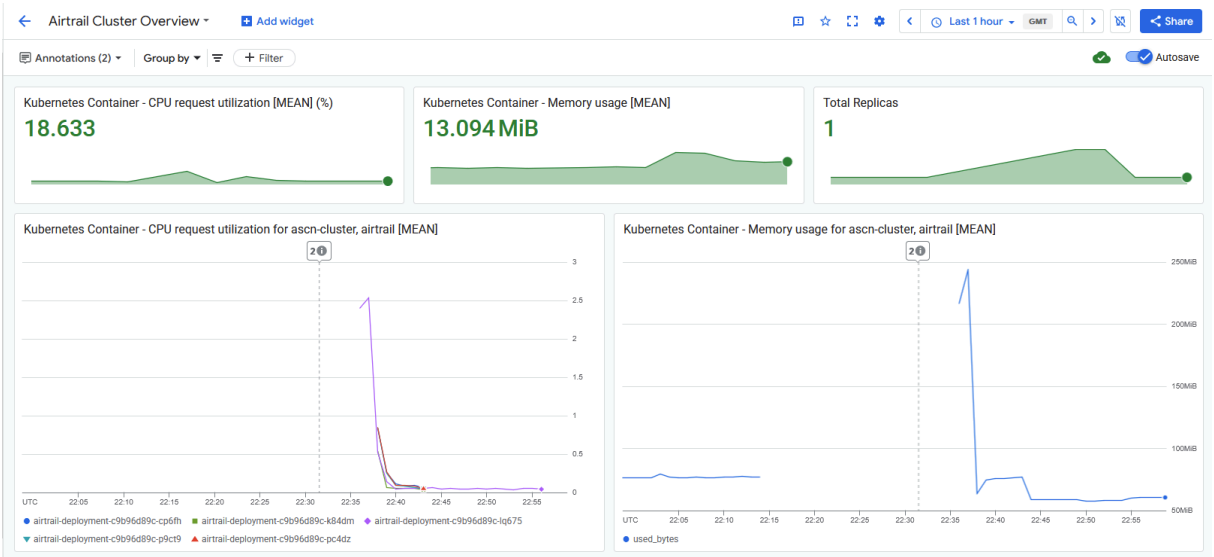

Figura 4: *Dashboard*

4.1. Solução inicial

Inicialmente, realizou-se o *benchmark* à aplicação inicial, de modo a conseguir obter dados que comprovem a necessidade de redundância e tolerância a faltas.

Foi logo identificado a necessidade de escalar verticalmente as máquinas dos *clusters*. Os pods da aplicação entravam em *CrashLoopBackOff*, onde o serviço se torna indisponível, derivado à utilização máxima da memória das máquinas, passando-se, assim, de máquinas com 2 vCPUs e 2GB de memória (`e2-small`) para máquinas com 2 vCPUs e 8GB de memória (`e2-standard-2`).

4.2. Solução escalável

A solução escalável, a utilizar o HPA, durante o *deploy*, iniciou o máximo de máquinas definidas, pois há um pico no pedido de CPU, demonstrado na Figura 5:

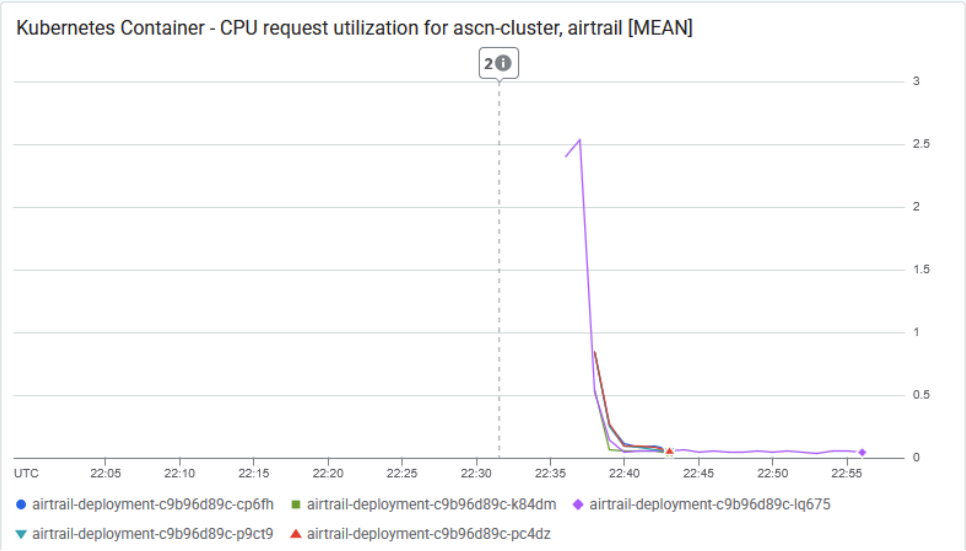

Figura 5: Gráfico pedido de CPU ao *deploy*

Passados cinco minutos, volta a estar apenas um *pod* em *deploy*. Está demonstrado na Figura 6:

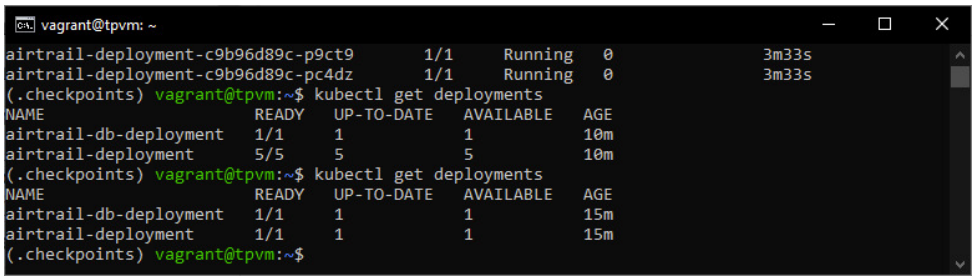

Figura 6: `get deployments` imediatamente após *deploy* e após cinco minutos

Com isto, é demonstrado que o HPA consegue moderar a quantidade de recursos que é utilizada, conforme a necessidade e a métrica estabelecida.

4.2.1. *Light*

Com o cenário *light*, não houve novas réplicas, pois a carga não era suficiente. O pedido de CPU também não sofreu grandes alterações.

5. Conclusão e Trabalho Futuro

O presente trabalho prático teve como objetivo a orquestração e automação da aplicação *AirTrail* em ambiente *cloud*, utilizando o **Google Kubernetes Engine (GKE)** e a ferramenta de configuração **Ansible**.

5.1. Síntese do Trabalho Desenvolvido

Através da elaboração dos *playbooks* de Ansible, foi possível atingir os objetivos propostos na Tarefa 1, nomeadamente a automação completa do ciclo de vida da aplicação:

- **Infraestrutura como Código :** Criação e destruição do *cluster* GKE de forma automática.
- **Deployment Automatizado:** Instalação da Base de Dados (com persistência via PVC) e da Aplicação Web, garantindo que os serviços ficam expostos corretamente.
- **Reprodutibilidade:** A estrutura modular dos *roles* (`airtrail-db`, `airtrail-app`) permite que o ambiente seja replicado facilmente para testes ou produção.

Relativamente à Tarefa 2, a análise de desempenho realizada através dos testes de carga com JMeter identificou a camada da Web App como um dos gargalos na utilização de CPU. Considerando o objetivo simplificador do projeto e o esforço envolvido em resolver limitações em outras camadas, optou-se por focar na resolução deste gargalo através da implementação do Horizontal Pod Autoscaler (HPA).

5.2. Limitações e Trabalho Futuro

Apesar do sucesso na implementação da escalabilidade na camada aplicacional, a solução atual apresenta ainda limitações, nomeadamente ao nível da base de dados:

- **Ponto Único de Falha :** A base de dados PostgreSQL reside num único *pod*. Se este nó falhar, a aplicação perde a capacidade de processar dados, embora o volume persistente garanta que os dados não sejam perdidos.
- **Escalabilidade de Dados:** O HPA não resolve problemas de I/O na base de dados.

Como trabalho futuro, sugeria-se a implementação de uma arquitetura de base de dados em *cluster*, utilizando *StatefulSets* do Kubernetes para alta disponibilidade. Para além disso, o desenvolvimento de um *pipeline* de CI/CD para executar os *playbooks* automaticamente mediante alterações no repositório de código.

5.3. Dificuldades

Com a sobreposição de diversos trabalhos e falta de coordenação entre o grupo, não foi possível testar mais a fundo a aplicação em *benchmark*, tendo resultado em dados incompletos e não satisfatórios.