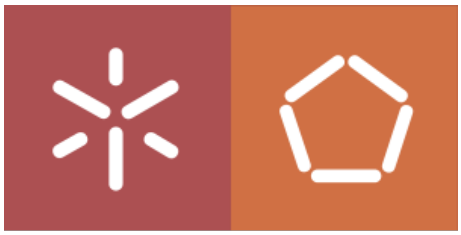

Universidade do Minho
Escola de Engenharia

Aplicações e Serviços de Computação em Nuvem

Trabalho Prático

Grupo 19

pg60249	pg60268	pg60007	pg60291	pg60309
Edgar Pinto	João Brito	Luís França	Pedro Matos	Tomás Oliveira

2025/2026

Índice

1. Introdução

Este trabalho foi realizado no âmbito da unidade curricular de Aplicações e Serviços de Comunicação em Nuvem, tratando-se da instalação, configuração, monitorização e avaliação da aplicação **AirTrail**, que permite o rastreio e consulta de histórico de voos.

2. Arquitetura da aplicação AirTrail

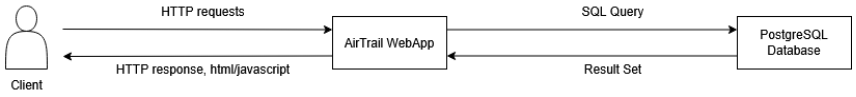

Figura 1: Arquitetura da aplicação AirTrail

Como a Figura 1 indica a aplicação está dividida em dois componentes principais, o servidor da *web app* e a base de dados **PostgreSQL**.

A interface do cliente é responsável pelo conteúdo que é visualizado ao inicializar a aplicação, ou seja, a página de *login* para introduzir as suas credenciais e a página principal com o mapa-mundo para poder adicionar os voos e visualizá-los.

O servidor da *web app* tem duas funções principais: transmitir a interface de utilização para o cliente e comunicar com a base de dados através de uma ORM. Ao aceder à aplicação através de um *browser* o servidor fornece o código HTML e JavaScript que permite que o cliente visualize a aplicação trocando mensagens HTTP, de modo com que o cliente interaja com o *backend* da aplicação. A partir destas mensagens são realizadas as operações desejadas pelo cliente (adicionar voos, estatísticas) pelo seu *input*, resgatando a informação necessária à base de dados. Do lado do servidor esta interação é realizada utilizando a ORM **Kysely**.

A base de dados é responsável por armazenar valores relacionados às diversas classes da aplicação utilizando PostgreSQL que tem uma melhor eficiência em aplicações que necessitam de várias operações de escrita e consultas complexas.

3. Funcionalidades e API

As seguintes funcionalidades são proporcionadas pela AirTrail:

- *Add Flight:* Ao clicar no primeiro botão da barra de navegação o utilizador pode adicionar um novo voo, preenchendo de seguida diversos campos, como origem/destino, número do voo e hora de chegada/partida;
- *Export / Import:* A aplicação permite aos utilizadores exportarem os dados de voo para o formato CSV ou JSON, e permite importar dados do *website* **MyFlightradar24** e das aplicações **AppInTheAir**, **jetlog**, **TripIt** e **Flightly**;
- *OAuth Authentication:* Autenticação via **OpenID Connect**, um protocolo criado com base na estrutura **OAuth 2.0**.

A aplicação disponibiliza uma série de estatísticas acessadas pela barra de navegação:

- Número total de voos completos
- Distância total viajada
- Tempo total em voo
- Número de aeroportos visitados
- Distribuição de voos por classe de assento
- Distribuição de voos por assento
- Distribuição de voos por motivo
- Quantidade de visitas por continente
- Quantidade de voos por mês
- Quantidade de voos por dia da semana.

A API utilizada é a **adsbdb** (*Automatic Dependent Surveillance-Broadcast Database*), uma API pública que permite obter informações sobre aeronaves, companhias aéreas e rotas de voo. Esta API acede às seguintes fontes para obter as informações sobre os voos e as aeronaves:
- **PlaneBase** - dados técnicos
- **Airport-Data.com** - imagens e informações visuais

Para utilizar a API, existem os seguintes métodos de obter e alterar dados:
- Para listar todos os voos feitos pelo utilizador na aplicação, utiliza-se **`GET /flight/list`**
- Para listar os detalhes de um voo específico, faz-se **`GET /flight/get/{id}`**, em que **`{id}`** é o número de identificador do voo
- É possível criar um novo voo ou alterar um voo pré-existente através de **`POST /flight/save`**
- Finalmente, pode-se eliminar um voo da lista local com **`POST /flight/delete`**

Estes métodos são requisitados através de HTTP com o endereço real do serviço do utilizador, por exemplo, "your-instance.com", no formato https://your-instance.com/api.

Todos estes comandos têm que passar por uma verificação de segurança feita através de uma chave API que é gerada para cada utilizador que serve como um *token* de autenticação (*bearer token*) e impede que os dados estejam disponíveis para qualquer pessoa que use a aplicação.

4. Instalação local da aplicação

Primeiramente instalamos a *app* seguindo o guia no *website*. Desta forma, são criados dois *containers*, um para a base de dados e outro para a *web app*. Isto foi possível utilizando o comando `docker compose` que permite inicializar múltiplos *containers* simultaneamente.

De seguida procedemos à instalação manual através do `bun` para executar o código da aplicação. Este processo, embora relativamente simples, exigiu a instalação prévia de algumas dependências, nomeadamente **git**, **nodejs**, **bun** e PostgreSQL. Realizamos esta instalação com duas máquinas virtuais, uma para a execução da aplicação, outra para a base de dados.

Para configurar a aplicação tivemos que criar um ficheiro `.env`, utilizando como base o exemplo já fornecido pelos docentes no repositório. Os campos que tivemos de mudar

são `ORIGIN(S)` e `DB_URL` onde colocamos o endereço IP da VM da aplicação e da VM da base de dados respetivamente; para os valores de `DB_PASSWORD`, `DB_DATABASE_NAME` e `DB_USERNAME` garantimos que a VM contendo a base de dados era configurada com exatamente os mesmos valores já fornecidos no `.env.example`.

Durante a instalação, foram detetados problemas de desempenho na máquina virtual, relacionados com limitações de memória RAM. Estes erros de memória aconteceram durante a execução do comando `bun run build`, e sugerem que o projeto requer uma quantidade mínima de recursos para correr de forma estável. Utilizamos 4096 MB para concluir a instalação, pois o valor inicial de 1024 MB resultou num tempo de *build* excessivo. Também alteramos as configurações do PostgreSQL na VM da base de dados de forma a aceitar outras conexões além de `localhost`.

5. Instalação e configuração da aplicação na *cloud*

5.1. Instalação base

Para fazer a instalação base pedida no segundo *checkpoint* do projeto, foi necessário criar três componentes essenciais para realizar o *deployment* correto da aplicação: a criação do *cluster* com a **Google Kubernetes Engine** (GKE), a criação de uma base de dados para garantir a existência da camada de dados e a própria camada aplicacional que foca no AirTrail em si.

A criação e eventual destruição dos *clusters* é controlada pelos playbooks `gke-cluster-create` e `gke-cluster-destroy`. Durante a criação do *cluster* é também criado um *node pool* constituído por duas máquinas virtuais.

A base de dados é tratada pelo *role* `k8s_postgres` que cria um *Persistent Volume Claim* (PVC) de 10 GB para garantir que os dados persistam quando se reiniciam os *pods*. A base de dados em si é uma instância PostgreSQL que está acessível internamente no *cluster* por uma porta específica disponibilizada pelo serviço **ClusterIP**.

Para realizar o *deployment* do servidor aplicacional criou-se a *role* `k8s_airtrail` que associa a base de dados e outros valores essenciais através de variáveis ambiente e cria um *deployment* e um *service*. O *deployment* `airtrail-deployment.yml`, instancia os *pods* utilizando a imagem configurada e define os *requests* de recursos necessários para a estabilidade do sistema. O `airtrail-service.yml`, cria um *Service* do tipo *LoadBalancer* que permitindo a comunicação interna e externa da aplicação, garantindo que o tráfego seja distribuído de forma eficiente e que o serviço permaneça resiliente a falhas individuais de instâncias.

5.2. Instalação final

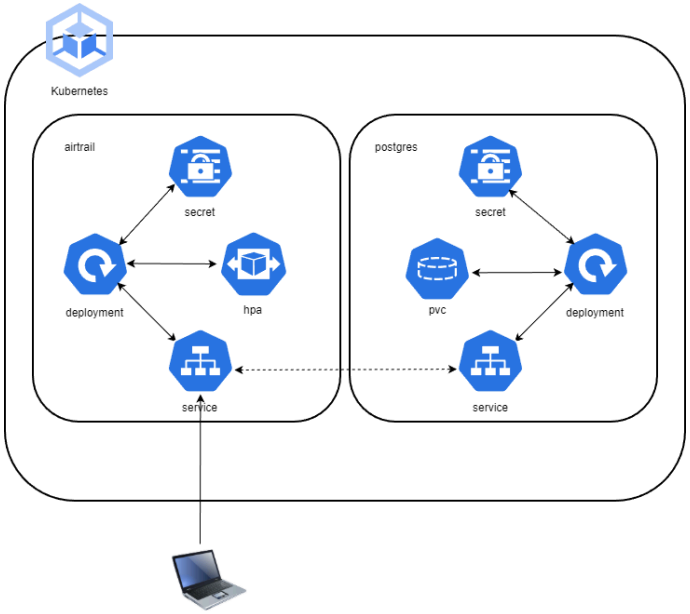

Figura 2: Arquitetura da Solução Final

Para a instalação final, foram adicionadas diversas otimizações à instalação base mencionada anteriormente, ilustradas na Figura 2. Optamos por priorizar o servidor *web*/aplicacional porque as melhorias aqui introduzidas têm um impacto muito mais visível para o utilizador final – como o servidor aplicacional não precisa de guardar o estado, os ganhos provenientes da replicação são mais acentuados.

Para tornar a aplicação mais escalável para um número crescente de clientes implementou-se um *Horizontal Pod Autoscaler* (HPA) que permite ganhar escalabilidade horizontal na aplicação AirTrail através da criação de réplicas. Configurou-se o HPA de modo a utilizar a percentagem de *CPU request* para determinar quando criar ou destruir novas réplicas. Além disso, é possível definir o número mínimo e máximo de réplicas que podem ser geradas a qualquer ponto. Na configuração final, foi definido o valor mínimo de réplicas igual a 2 de modo a garantir que não existe apenas 1 réplica funcional a qualquer momento (o que representaria um ponto único de falha). O número máximo de réplicas é igual a 5 pois concluiu-se que seria o número mais proveitoso tendo em conta os testes realizados na secção 9. Ainda, implementou-se uma funcionalidade de *Session Affinity*, garantindo que, num cenário com múltiplas réplicas ativas, o tráfego de um utilizador específico seja direcionado consistentemente para a mesma réplica durante a sua sessão.

Para aumentar a segurança da aplicação foram implementados *Ansible Vaults* e *Kubernetes Secrets* que melhoram a segurança exponencialmente no repositório e no *cluster* respetivamente.

O *Ansible Vault* baseia-se no ficheiro `vault.yml`, cujo conteúdo se encontra encriptado com AES-256 e só pode ser desencriptado através de uma chave armazenada localmente (`vault_pass.txt`). Estes valores desencriptados podem depois ser aproveitados durante a execução do *playbook*.

Por sua vez, os *Kubernetes Secrets* tratam da segurança dentro do *cluster*. O ficheiro principal de *deployment* da aplicação (`airtrail-deployment.yml`) não contém nem acede diretamente à palavra-passe da base de dados; em vez disso, obtém-na a partir do cofre `airtrail-secret`, sendo esta disponibilizada temporariamente como uma variável de ambiente (`DB_PASSWORD`). Esta abordagem em camadas assegura que as credenciais nunca são expostas em *plaintext*, nem no código-fonte nem nos ficheiros de configuração do *cluster*.

5.3. Teste de funcionalidades

Para testar se a aplicação AirTrail está completamente funcional, executamos o *playbook* `test_all.yml` que ambos *deployment* e *undeployment* ocorrem sem erros, se as APIs são devidamente chamadas e se há persistência dos dados depois de um *undeploy*. A instalação final passa todos estes testes, pelo que a aplicação disponibiliza as funcionalidades corretamente.

6. Monitorização, métricas e visualizações

A implementação de um serviço de monitorização é fundamental para qualquer instalação sobre uma infraestrutura *cloud*, pois permite observar o estado interno do sistema, como a sua saúde e performance, contribuindo para novas decisões de otimizações de parâmetros de configuração. Para este projeto escolhemos como ferramenta de monitorização o **Google Cloud Monitoring** nativo do motor de *cloud* já utilizado no restante trabalho. Como esta ferramenta possui integração nativa com o **Google Cloud** é capaz de informar em tempo real as métricas observadas pelo sistema, sem ter de passar por um agente externo de monitorização.

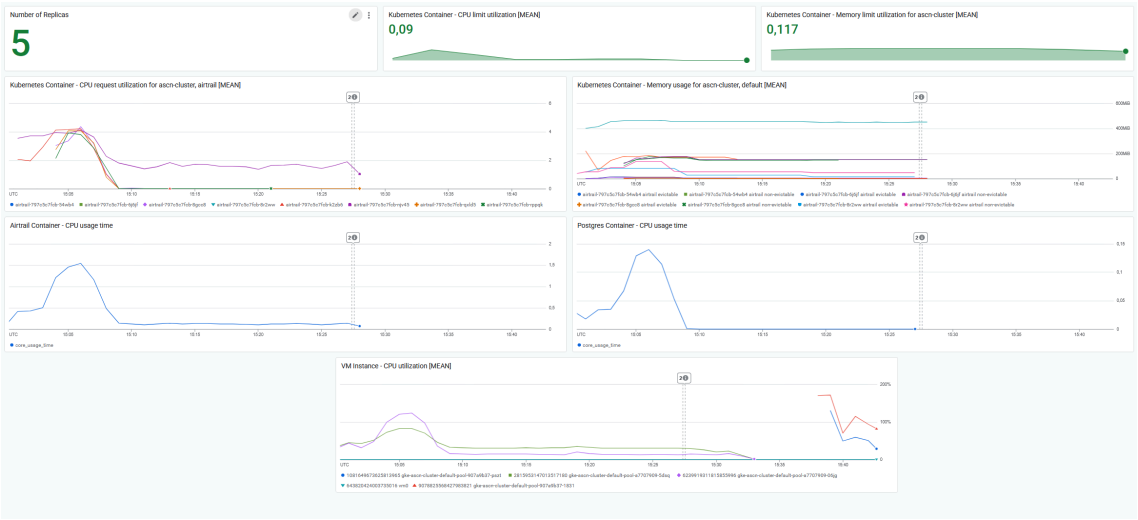

Figura 3: *Dashboard* desenvolvido para a monitorização da aplicação AirTrail

Na Figura 3 podemos observar o *Dashboard* desenvolvido para realizar a monitorização da aplicação. As métricas escolhidas foram:

- Número de réplicas

- Utilização do limite de CPU do *cluster*
- Utilização do limite de memória do *cluster*
- Utilização do pedido de CPU para os *pods* AirTrail
- Utilização de memória do *cluster*
- Tempo de uso do CPU para os *pods* AirTrail
- Tempo de uso do CPU para os *pods* PostgreSQL
- Uso do CPU das várias máquinas virtuais

Estas métricas permitem uma visão geral do estado do *cluster* permitindo a visualização do uso de CPU e memória dos vários *pods*, destacando as métricas relacionadas aos *pods* AirTrail pois é esse o componente que pretendemos escalar e otimizar.

7. *Benchmarking*

Como ferramenta de *benchmarking* aplicamos **Apache JMeter**, que permite automatizar e simular testes de carga sobre uma aplicação *web* capturando estatísticas relevantes.

Para testar a aplicação, desenvolvemos dois *scripts* de testes – `aitrail_test` e `airtrail_test_long`. O primeiro simula vários acessos ao sistema num curto período de tempo. Este conecta 100 clientes, testando assim o nível de adaptação do *site* a acréscimos rápidos de débito.

O teste `airtrail_test_long` avalia a estabilidade e o limite de concorrência da aplicação através de duas fases: na primeira metade, o número de utilizadores cresce gradualmente até atingir os 1000 clientes simultâneos (*ramp-up*). Na segunda metade, essa carga máxima é mantida constante para testar a resiliência do AirTrail sob stress prolongado, garantindo que o sistema sustenta o desempenho sem degradação ao longo do tempo.

Para executar os testes criamos uma máquina virtual do Google Cloud dedicada. Esta abordagem permite uma melhor consistência e reprodutibilidade dos testes, pois garante-se que todos executam com os mesmos recursos. É também importante salientar que todos os testes correram com a *Session Affinity* desligada já que esta funcionalidade obrigaria que corressem sobre o mesmo *pod*, tornando impossível testar a replicação do sistema.

7.1. Teste *baseline*

De início, executamos um teste *baseline*, isto é, um teste ao *site* inicial sem nenhuma alteração quanto às otimizações.

Realizamos o `airtrail_test_long`, em que a média final do tempo de resposta foi 7613.79 ms e o débito atingiu 96.00 *req/sec* (pedidos por segundo). A performance desempenhada é muito fraca para o número de clientes proposto.

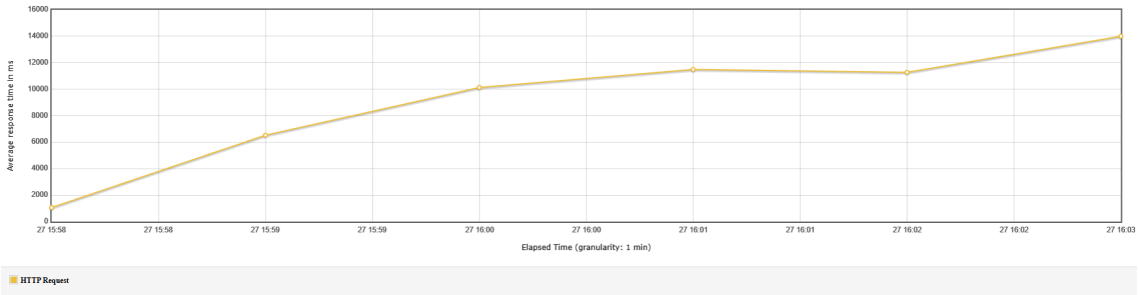

Figura 4: Tempo de resposta ao longo do tempo.

Na Figura 4 observamos que o tempo de resposta começa a aproximadamente 1000 ms, aumentando progressivamente até ao ponto onde a carga de clientes é mantida; no seu pico a aplicação atingiu um atraso de 14000 ms.

Além das métricas já referidas, durante este teste o *site* sofreu falhas em cerca de 19.21% das respostas.

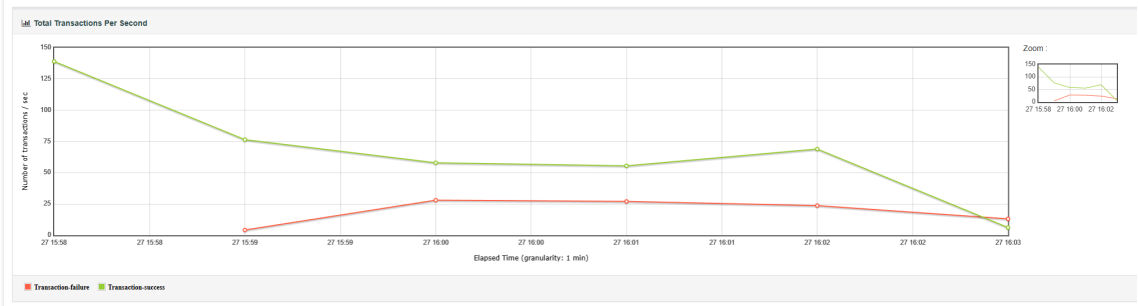

Figura 5: Testes sucedidos e falhados ao longo do tempo.

Na Figura 5 conseguimos identificar a evolução do número de transações bem- e mal--sucedidas ao longo do tempo. Verificamos que a taxa de sucesso piora com um número maior de *threads*, o que constitui mais uma evidência que o desempenho atingido pela instalação base é indesejável.

7.2. Testes de escabilidade

Para testar a capacidade do *cluster* de suportar um número crescente de réplicas, realizamos o `airtrail_test` sobre um número de réplicas crescente, de 1 até 5.

Réplicas	**1**	**2**	**3**	**4**	**5**
***Mean Response Time* (ms)**	66.02	34.08	26.93	69.55	86.79
***Throughput* (*req/sec*)**	253.16	296.54	295.87	268.98	249.91

Tabela 1: Resultados de *Benchmarking* para a Instância *e2-small*

Como podemos observar na Tabela 1, a performance do *cluster* melhora progressivamente até degradar com as 4 e 5 réplicas. Isto indica que o *cluster* com a máquina *e2-small* não é muito viável para o processo de escalabilidade. Para colmatar esta falta, decidimos melhorar o tipo da máquina para *e2-medium* e repetir este teste de *benchmark* para maiores valores de réplicas.

Réplicas	1	3	5	7	9
***Mean Response Time* (ms)**	72.67	47.99	31.44	49.02	96.08
***Throughput* (*req/sec*)**	96	273.55	290.41	259.35	241.69

Tabela 2: Resultados de *Benchmarking* para a Instância *e2-medium*

Com os resultados da Tabela 2 pode-se observar que as métricas apenas começam a deteriorar a partir de 5 réplicas. Logo, como configuração para o HPA determinamos que o melhor número máximo de réplicas seria 5, aproveitando da melhor maneira as capacidades das máquinas virtuais escolhidas.

7.3. Testes com escalonamento automático

Como teste final realizamos o teste `airtrail-test-long` sobre a configuração final com HPA. Definimos como mínimo de réplicas 2 e máximo 5.

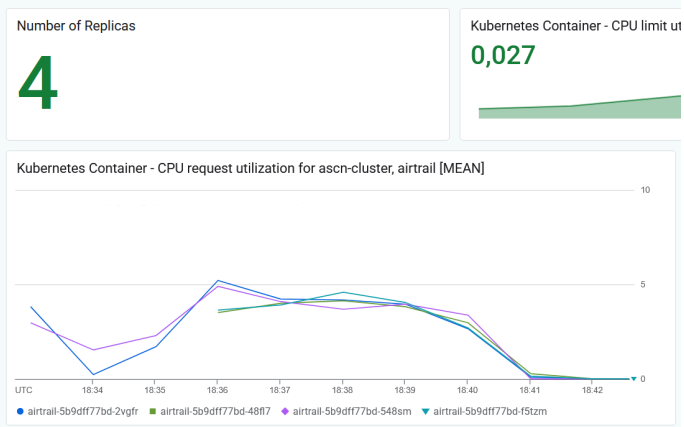

Figura 6: Valores de réplicas e *CPU request* para o teste inicial com HPA

Observamos através do gráfico apresentado na Figura 6 que o HPA não cria um número acima de 4 réplicas, mesmo excedendo o valor de CPU *request*. Isto pode ser explicado pelo facto dos *pods* anteriormente criados ocuparem demasiados recursos do *cluster*, tornando impossível criar mais réplicas além das existentes. Para resolver este problema introduziu-se um valor de CPU *limit* igual a 300 *millicores* de modo a garantir que as máquinas não utilizem recursos excessivos.

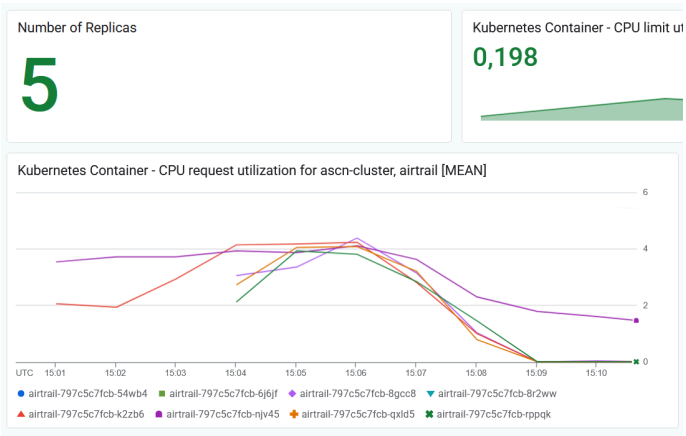

Figura 7: Valores de réplicas e *CPU request* para o teste inicial com HPA

No gráfico da Figura 7 constatamos que as 5 réplicas foram corretamente criadas, assim que a carga aumenta, desaparecendo depois da carga diminuir.

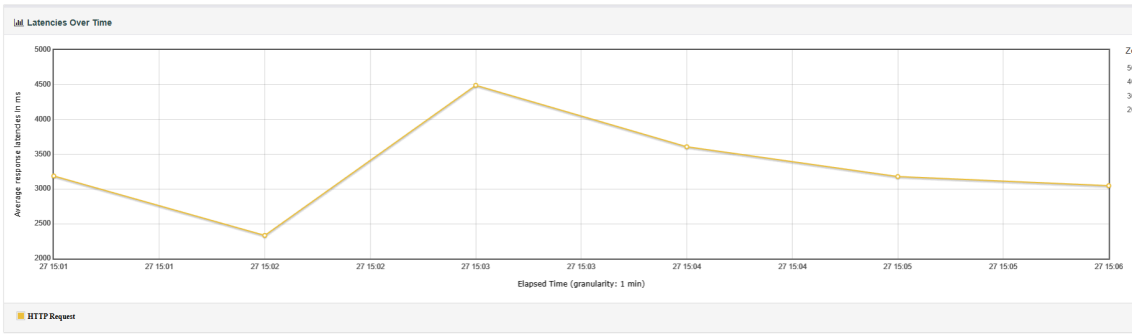

Figura 8: Tempo de resposta ao longo do tempo

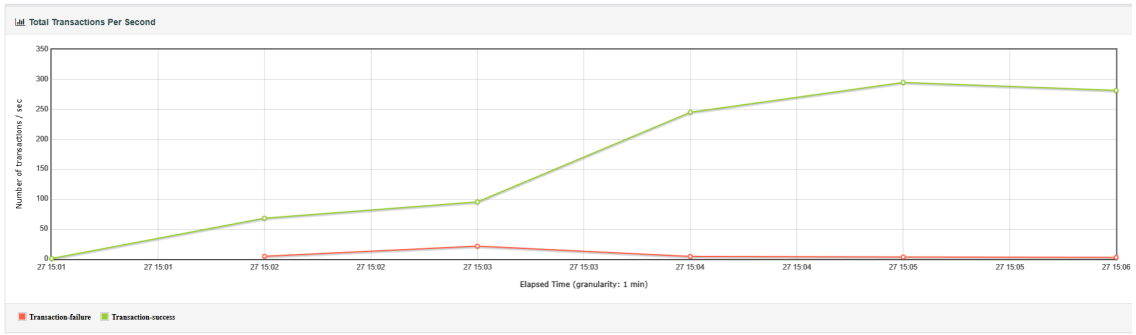

Figura 9: Número de transações com sucesso e com erros ao longo do tempo

A partir dos gráficos obtidos com o JMeter, verificamos uma grande melhoria em desempenho. Na Figura 8 vimos um aumento acentuado até às 15:03, que representa o tempo de resposta até serem criadas as réplicas. Após estas serem criadas, o tempo de resposta começa a diminuir novamente, mesmo com carga equivalente. Os dados obtidos a partir da Figura 9 reforçam este facto, pois podemos observar que a maior parte dos erros ocorre no período de tempo em que estão a ser arrancadas novas réplicas.

Testes	*baseline*	Configuração final
***Mean Response Time* (ms)**	7613.79	3593.95
***Throughput* (*req/sec*)**	96.00	200.99
***Error rate* (%)**	19.21	3.55

Tabela 3: Comparação entre a *baseline* e o resultado final.

Apesar de erros e alguns aumentos do tempo de resposta durante a inicialização das replicas, é possível observar pela Tabela 3 que comparativamente à configuração sem otimizações há um aumento em performance da configuração final. Esta melhoria refere-se não só às métricas destacadas pela Tabela 3 mas também a um aumento da quantidade de clientes suportados e uma melhor estabilidade da aplicação.

8. Questões da Tarefa 2

1. Considerando a instalação base proposta pelo grupo para a Tarefa 1:

a. Para um número crescente de clientes, que componentes da aplicação poderão constituir um gargalo de desempenho?

Um componente da aplicação que constitui um gargalo de desempenho é o próprio *cluster* GKE. Na instalação base, o *cluster* tem um limite de dois nodos *e2-small* com 100 GB de memória, o que restringe a capacidade total de processamento e memória disponível para os diferentes *pods*. Além disso, todo o tráfego externo entra por um único ponto de acesso, o **LoadBalancer**, o que também pode levar a um gargalo com o aumento do número de clientes.

Existe também o problema oposto na parte do *deployment* da aplicação AirTrail e da base de dados, pois não há limites de CPU ou memória definido para cada *pod*, ou seja, se um *pod* for particularmente custoso pode indiretamente incapacitar todos os outros.

Outro ponto que constitui um gargalo de desempenho é a falta de escalabilidade horizontal nas componentes do servidor *web*/aplicacional e da base de dados PostgreSQL e, por isso, à medida que o número de clientes aumenta, a carga aplicada a estas duas componentes vai também aumentar, convergindo todas as operações para a única réplica existente.

Em suma, para um número crescente de utilizadores, o desempenho da aplicação é limitado não só pela base de dados mas também por restrições impostas ao nível do *cluster* e da arquitetura global da solução.

b. Qual o desempenho da aplicação perante diferentes números de clientes e cargas de trabalho?

Baseado nos testes efetuados através do JMeter, conseguimos determinar o desempenho teórico da aplicação com um número crescente de clientes. Com um número reduzido de clientes, o desempenho é estável com latências aproximadas do mínimo registado e há baixo consumo de recursos já que o *cluster* não está sob pressão. Por outro lado, para um número mais elevado de clientes, o desempenho acaba por degradar progressivamente

com latências de resposta a aumentar, tornando a aplicação praticamente inutilizável para números muito grandes de clientes. A instalação base possui um fator de escalabilidade baixo.

c. Que componentes da aplicação poderão constituir um ponto único de falha?

Como já foi mencionado na questão 1.a., há principalmente dois componentes da aplicação que poderão constituir um ponto único de falha, a base de dados PostgreSQL e o servidor *web/aplicacional*. Estes dois pontos precisam de ser otimizados para garantir a sua escalabilidade, porque se ocorrer um aumento de clientes, a aplicação base ficará sobrecarregada e uma das instâncias não será capaz de suportar a carga acrescida de clientes.

2. Com base nas respostas dadas para as questões anteriores:

a. Que otimizações de distribuição/replicação de carga podem ser aplicadas à instalação base?

Uma possível otimização é o *Horizontal Pod Autoscaler* (HPA) que é um mecanismo do Kubernetes que permite implementar escalabilidade horizontal da aplicação com base no uso de CPU ou memória. A criação de mais nós em qualquer uma das camadas permite corrigir o erro de "único ponto de falha" mencionado anteriormente.

Embora seja tecnicamente possível escalar horizontalmente tanto a *web app* quanto a base de dados PostgreSQL através de réplicas no Kubernetes, optamos por priorizar a *web app*. Esta escolha deve-se ao facto de as aplicações *stateless* (sem estado) poderem ser replicadas instantaneamente via HPA para responder a picos de tráfego, sem complexidade adicional. Já o escalonamento do PostgreSQL exige uma arquitetura mais sensível de replicação de dados, onde dividimos a carga entre um *pod* mestre para operações de escrita e múltiplas réplicas dedicadas exclusivamente à leitura, garantindo que a persistência dos dados permaneça íntegra e sincronizada entre os nós.

Outra alteração é a definição de limite de recursos para cada *pod* para garantir que nenhum *pod* individualmente consuma demasiados recursos da restante infraestrutura.

b. Qual o impacto das otimizações propostas no desempenho e/ou resiliência da aplicação?

Estas otimizações elevam o desempenho e a resiliência da infraestrutura. Ao nível da performance, a disponibilização de mais réplicas permite aumentar o débito de pedidos e reduz a latência dos mesmos. Quanto à resiliência, a eliminação de pontos únicos de falha garante que, em caso de erro crítico, a recuperação ocorra muito mais rapidamente, minimizando drasticamente o tempo de inatividade.

9. Reflexão Final

A realização deste trabalho prático permitiu consolidar os conceitos teóricos de computação em nuvem, especificamente no uso de *clusters* através de Kubernetes e na automatização de uma infraestrutura através de Ansible. Com estas ferramentas, o

objetivo principal de automatizar a instalação da aplicação AirTrail foi atingido com sucesso.

Como pontos fortes do projeto, destaca-se que foi implementada uma abordagem onde todo o ambiente, desde a criação do *cluster* GKE até o *deployment* da aplicação e execução de testes, é realizado através de comandos Ansible, sem precisar de ajustes manuais. Além disso, foi feita a análise dos gargalos de desempenho através da monitorização por JMeter e os *dashboards* do Google Cloud e foram feitas otimizações para aprimorar a componente do servidor *web*/aplicacional.

A aplicação apresenta todas as suas funcionalidades operacionais, com o *website* a funcionar corretamente e de forma estável. Importa ainda salientar que a persistência de dados foi garantida, sendo possível mantê-la após a realização de um *undeploy* normal, assegurando a integridade e continuidade da informação.

Quanto aos pontos a melhorar, reconhecemos que a arquitetura atual, embora funcional, mantém a base de dados como um ponto único de falha. Como trabalho futuro, a implementação da replicação seria essencial para garantir a resiliência quanto a este problema. Adicionalmente, a monitorização poderia evoluir de testes pontuais para uma solução que utiliza ferramentas como **Prometheus** e **Grafana** para visualização de métricas em tempo real.