



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Trabalho Prático

Relatório de Desenvolvimento

Mestrado em Engenharia Informática

Aplicações e Serviços de Computação em Nuvem

*Desenvolvido pelo **grupo 28**:*

Eduardo Silva	pg47167
Ivo Baixo	pg47271
José C. Magalhães	pg47355
Jorge Vieira	pg47349
Paulo R. Pereira	pg47554

11 de janeiro de 2022

Conteúdo

1	Introdução	5
2	Arquitetura Base do Sistema	6
2.1	Descrição da Arquitetura	6
2.2	Identificação dos SPOF's	7
2.3	Implementação na <i>Google Cloud Platform</i>	7
2.3.1	Ferramentas de configuração automática	7
2.3.2	Observações acerca da GCP	8
2.4	Testes de Desempenho	8
2.4.1	Método de Testes	8
2.4.2	Teste 1: Página Inicial	9
2.4.3	Teste 2: Registrar um novo Utilizador	9
2.4.4	Teste 3: <i>Login</i>	10
2.4.5	Teste 4: <i>Login</i> + Criar Página	10
2.4.6	Nota relativa aos resultados	11
3	Proposta de Arquitetura	12
3.1	Proposta de resolução dos SPOF's	12
3.1.1	Web-Server	12
3.1.2	Base de Dados	13
3.2	Implementação na <i>Google Cloud Platform</i>	13
3.2.1	Observações acerca da GCP	13
3.2.2	Ferramentas de configuração automática	14
3.2.3	Regras de <i>Firewall</i>	15
3.2.4	Balanceador de Carga	15
3.2.5	Grupo de Instâncias	17

3.2.6	SQL <i>instance</i> e respetivas réplicas	18
3.2.7	PROXY SQL - PGpool	19
3.3	Visualização da Arquitetura	20
3.4	Ferramentas de Monitorização	21
3.4.1	Web-servers	21
3.4.2	Base de Dados	22
3.5	Testes de Desempenho	23
3.5.1	Teste 1: Página Inicial	23
3.5.2	Teste 2: Registar um novo Utilizador	23
3.5.3	Teste 3: <i>Login</i>	24
3.5.4	Teste 4: <i>Login</i> + Criar Página	24
3.6	Análise de Resultados	25
4	Conclusão	28
A	Anexos	29
A.1	Ficheiro JSON relativo a registar um novo Utilizador	29
A.2	Ficheiro JSON relativo ao <i>login</i>	30
A.3	Ficheiro JSON relativo à criação de uma página	31

Lista de Figuras

2.1	Arquitetura simples da plataforma <i>Wiki.js</i>	6
2.2	Resultados estatísticos do Teste 1 - Arquitetura simples	9
2.3	Resultados estatísticos do Teste 2 - Arquitetura simples	10
2.4	Resultados estatísticos do Teste 3 - Arquitetura simples	10
2.5	Resultados estatísticos do Teste 4 - Arquitetura simples	11
3.1	Plataforma <i>Wiki.js</i> a funcionar	12
3.2	Regras de <i>Firewall</i> definidas	15
3.3	Balanceador de carga HTTP	17
3.4	<i>Instance Group</i> associado ao balanceador HTTP	18
3.5	Instâncias SQL	18
3.6	Proxy pgpool - Arquitetura idealizada mas por implementar	19
3.7	Arquitetura proposta	20
3.8	Gráfico referente a <i>Egress/Ingress bytes</i>	22
3.9	Utilização da ferramenta <i>Query Insights</i>	23
3.10	Resultados estatísticos do Teste 1 - Arquitetura complexa	23
3.11	Resultados estatísticos do Teste 2 - Arquitetura complexa	24
3.12	Resultados estatísticos do Teste 3 - Arquitetura complexa	24
3.13	Resultados estatísticos do Teste 4 - Arquitetura complexa	25
3.14	Utilização de CPU da BD no Teste 4	25
3.15	Utilização de CPU dos servidores no Teste 4	26
3.16	Utilização de CPU dos servidores no Teste 1	27
3.17	Variação de memória usada de um dos servidores	27

1. Introdução

No âmbito da unidade curricular de Aplicações e Serviços de Computação em Nuvem foi proposto efetuar a caracterização, análise, instalação, monitorização e avaliação da aplicação *Wiki.js*. Esta plataforma é *open-source* e tem como funcionalidades principais a autenticação de utilizadores, a criação de páginas e a adição de comentários às mesmas.

Uma parte fulcral da análise da aplicação é o estudo das suas possíveis falhas. Obviamente, é desejado um bom serviço, de alta qualidade, que garanta um fluxo ininterrupto de informação, i.e., um fluxo sem falhas (identificação dos **SPOFs**, *Single Point of Failure*). Para tal, é necessário criar um plano que impeça a indisponibilidade da aplicação sempre que um dos seus recursos falhe, bem como, o uso de balanceadores de carga para garantir um uso racional dos mesmos.

Assim, o presente relatório descreve a arquitetura proposta pelo grupo, em que os **SPOFs** identificados são devidamente resolvidos e implementados. Além disto, e de modo a comprovar que de facto se encontram resolvidos, foram realizados testes à plataforma.

Acrescenta-se que o relatório se encontra bem documentado e que a ordem do mesmo foi fiel à que nos é apresentada no enunciado.

2. Arquitetura Base do Sistema

2.1 Descrição da Arquitetura

A primeira etapa do enunciado prático passou pelo estudo e compreensão de vários pontos, entre eles qual a arquitetura e componentes da plataforma Wiki.js. Esta segue, então, a arquitetura típica de um *web-server*, sendo composta essencialmente por três grandes componentes:

- **Front-End:** Contém toda a interação com o utilizador, apresentada em **HTML** e **JavaScript**. Os pedidos são enviados, ao servidor *back-end*, utilizando **GraphQL**.
- **Back-End:** É constituído por um grande número de serviços, desde a simples autenticação de um cliente até à incorporação de outros serviços. De relevantes para o enunciado em questão destacam-se o servidor, em **Node.js**, a base de dados que, embora não haja propriamente uma limitação no que toca à escolha desta, decidiu-se seguir a sugestão da própria plataforma utilizando, assim, **PostgreSQL**. Por último, e para efeitos de testes, é incorporado o serviço referente ao **GraphQL**.

De seguida, podemos ver uma representação do que foi descrito anteriormente:

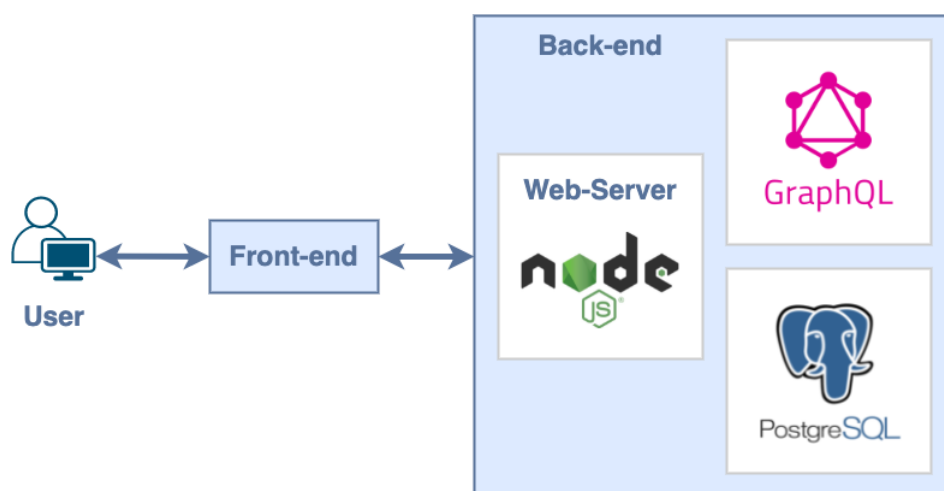


Figura 2.1: Arquitetura simples da plataforma Wiki.js

2.2 Identificação dos SPOF's

Um determinado componente do sistema é definido como um *Single Point of Failure* se, no caso de falhar, todo o sistema colapse, levando, assim, a que toda a aplicação deixe de funcionar.

- **Web-Server:** Toda a plataforma Wiki.js é apresentada ao seu utilizador através de uma interface gráfica. Caso esta, por algum motivo falhe, leva a que não haja interação possível, pois todos os restantes serviços na **back-end** dependem da *front-end*.
- **Base de Dados:** Os setores referentes ao armazenamento são outro componente fundamental de toda a aplicação. No caso de terem uma avaria, a camada de acesso à persistência deixará de conseguir ler e escrever as respetivas mudanças de estado, levando a que o sistema se torne incoerente. Ou seja, neste caso concreto, se o servidor *Node.js* vir interrompida ou quebrada a sua conexão com a base de dados *PostgreSQL*, então todo o sistema fica comprometido.

2.3 Implementação na *Google Cloud Platform*

2.3.1 Ferramentas de configuração automática

Para a implementação da arquitetura básica, foi decidido que todos os componentes da plataforma iriam ser instalados na mesma instância de Máquina Virtual, de forma a conseguir retirar as melhores conclusões acerca dos vários pontos de *Bottleneck*.

O processo de instalação da aplicação foi simples e, conforme pretendido pelo enunciado, foi feito de forma automática usando, para isso, a poderosíssima ferramenta *Ansible*. Os passos da configuração, detalhadamente apresentados no respetivo *playbook*, passaram essencialmente pelo seguinte:

1. Criação da *gcp compute instance*, bem como do respetivo *gcp compute adress*, onde será feito o *deployment* do Wiki.js;
2. Instalação de componentes e dependências da plataforma;
3. Configuração da base de dados, tendo sido criada uma base de dados única ("*wiki*") apenas com um utilizador, ("*wikijs*"), e a respetiva *password* ("*wikijsrocks*"). Acrescenta-se que estes dados são os dados *default* presentes no ficheiro "*config.yml*" e, por isso, não foi necessário

mudar este último, não havendo problemas de conexão à base de dados por parte do servidor *Node.js*.

Por fim, e de modo a comprovar a correta instalação, basta colocar nos pedidos HTTP do *browser* do utilizador o IP da máquina em questão, bem como a porta 3000 (IP:3000).

2.3.2 Observações acerca da GCP

A interação com a *Google Cloud Platform* foi relativamente simples. Além da habitual "burocracia de autenticação", foi apenas necessário decidir que máquina se iria utilizar nas instâncias do projeto, tendo sempre em conta o crédito fornecido. Optou-se, então, por uma máquina com as seguintes características:

- **Localização:** europe-west1
- **Sistema Operativo:** Ubuntu 18.04
- **Tipo:** e2-medium
- **CPU:** 2vCPU
- **Memória** 4GB
- **Armazenamento:** 10GB

2.4 Testes de Desempenho

2.4.1 Método de Testes

A componente de testes dividiu-se essencialmente em duas partes: o uso da API de dados em *GraphQL* para a realização de todos os testes que necessitem de operações CRUD (*Create, Read, Update and Delete*) na *back-end* da aplicação, bem como a utilização do programa *Apache JMeter*, abordado nas aulas práticas da disciplina.

Antes de iniciar os testes, foi necessário seguir o protocolo da plataforma e, por isso, todos os pacotes HTTP criados com o objetivo de aceder ao *endpoint GraphQL*, têm de ter, obrigatoriamente, no cabeçalho uma **API-KEY**, sendo esta gerada, pelo administrador, nas definições do servidor em questão. Então, o formato, inserido no respetivo campo no *JMeter*, passa pelo seguinte:

- **Authorization:** Bearer API-KEY

Estando tratadas as devidas autorizações, foram então executados os testes utilizando, para isso, a já falada aplicação *Apache JMeter*, ferramenta que permite manipular várias vertentes para executar diversos testes, desde a criação de *threads* (para simular, por exemplo, vários utilizadores), a pedidos HTTP, bem como permitir efetuar o processo de *benchmarking*.

De forma a conseguir os melhores resultados possíveis, decidiu-se efetuar todos os testes no computador de um dos elementos do grupo.

Por fim, acrescenta-se que há testes em que foi necessário aceder à camada API exposta pelo *GraphQL*, tendo sido utilizado o *endpoint* por defeito, em */graphql*. Mais se informa que todos os ficheiros JSON correspondente ao *Body Data* utilizados nos diversos testes, para fins distintos, estão presentes em **Anexos**.

2.4.2 Teste 1: Página Inicial

Este teste, que é o cenário mais simples e envolve a requisição da página inicial, tem como objetivo entender o funcionamento da aplicação quando um número específico de utilizadores (que vai aumentando) tentam, em simultâneo, aceder à mesma. Para a sua concretização, basta enviar o seguinte pacote HTTP:

1. GET /

Os resultados obtidos encontram-se sintetizados na seguinte tabela:

Threads	Average	Min	Max	Std. Dev	Error %	Throughput
500	376	93	1004	282,78	0,00%	50,20
1000	159	94	1208	121,57	0,00%	49,80
5000	28898	107	57318	13700,49	0,00%	70,50
10000	23190	120	47591	14336,74	0,00%	69,90

Figura 2.2: Resultados estatísticos do Teste 1 - Arquitetura simples

2.4.3 Teste 2: Registrar um novo Utilizador

Este teste consiste em aceder à página de registo da plataforma e criar uma nova conta. Para isso, utilizou-se uma funcionalidade do *JMeter* que permite criar variáveis aleatórias, para que cada pedido *HTTP* registasse um novo utilizador.

É composto pelos seguintes passos:

1. GET /

2. POST /graphql

NOTA: *Body Data* presente em Anexos ??;

Threads	Average	Min	Max	Std. Dev	Error %	Throughput
500	47406	107	103757	16423,02	0,00%	14,60
1000	18295	159	35438	9991,12	1,05%	15,50

Figura 2.3: Resultados estatísticos do Teste 2 - Arquitetura simples

2.4.4 Teste 3: *Login*

Este teste consiste em fazer *login* com uma conta de um utilizador já existente. É composto pelas seguintes operações:

1. GET /login

2. POST /graphql

NOTA: *Body Data* presente em Anexos A.2;

Threads	Average	Min	Max	Std. Dev	Error %	Throughput
500	88	81	124	5,00	0,00%	49,70
1000	89	80	326	9,56	0,00%	49,80
5000	89	79	1148	29,44	0,00%	50,00
10000	89	78	1111	28,24	0,00%	50,00

Figura 2.4: Resultados estatísticos do Teste 3 - Arquitetura simples

2.4.5 Teste 4: *Login + Criar Página*

Este teste consiste em fazer *login* com uma conta já existente e posteriormente criar uma nova página. A criação de uma nova página é feita utilizando variáveis aleatórias, aplicando assim o mesmo método que foi utilizado no Teste 2. Este teste, mais complexo, é composto pelas seguintes operações:

1. GET /login

2. POST /graphql

NOTA: *Body Data* presente em Anexos A.2;

3. GET /e/en/new-pages

4. POST /graphql

NOTA: *Body Data* presente em Anexos A.3;

Threads	Average	Min	Max	Std. Dev	Error %	Throughput
500	42585	129	252454	76889,47	11,00%	7,80

Figura 2.5: Resultados estatísticos do Teste 4 - Arquitetura simples

2.4.6 Nota relativa aos resultados

Como se pode verificar, algumas tabelas podem parecer incompletas (por exemplo, o Teste 2 só foi efetuado para dois valores de *threads*: 500 e 1000) pois, nestes casos, o grupo achou que não traria qualquer vantagem do ponto de vista da análise. Por exemplo, no último cenário, só foi testado para 500 *threads* pois este número foi suficiente para mandar todo o sistema abaixo. Assim, testar para mais *threads* não traria qualquer benefício.

3. Proposta de Arquitetura

3.1 Proposta de resolução dos SPOF's

3.1.1 Web-Server

A plataforma segue o modelo geral de um *web-server*: é desenhada em *Node.js* e é considerada **stateless**, pois não guarda qualquer estado do utilizador em memória. Qualquer operação de CRUD necessária é processada diretamente na base de dados, que já se encontra devidamente separada do servidor. Desta forma, conclui-se que é possível criar tantas instâncias deste servidor quanto pretendido, sendo apenas necessário conhecer o IP da base de dados externa. Por fim, é necessário um mecanismo que não obrigue a que o cliente teste que servidores estão ativos, passando a solução por um balanceamento de carga. Este, que será analisado mais à frente, alberga todos os servidores disponíveis e resultará num único IP externo, facilitando assim todo o processo. No caso concreto da arquitetura proposta, foi estabelecido que seriam usadas **três** instâncias de servidores, um número considerado aceitável e plausível, tendo em conta também as limitações de crédito.

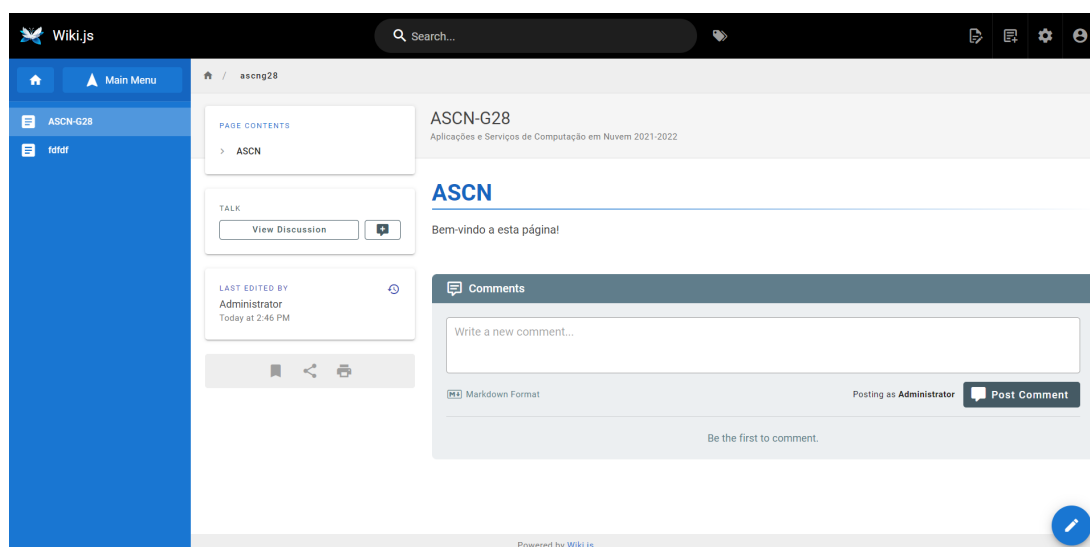


Figura 3.1: Plataforma *Wiki.js* a funcionar

3.1.2 Base de Dados

De modo a que a base de dados não fosse um SPOF, decidiu-se tirar proveito da configuração de Alta Disponibilidade da GCP, serviço esse disponibilizado pela *Cloud SQL instance*, utilizando assim esta ferramenta para alojar a nossa base de dados *PostgreSQL*.

Assim, estando a instância principal configurada para HA (*High Availability*), na ocorrência de um *failover*, a *Cloud SQL* altera automaticamente a disposição, passando esta a fornecer dados através da *standby instance*, protegendo então o sistema de um possível colapso, pois, num pequeno intervalo de tempo, serão novamente garantidas as opção de leitura e escrita por parte do servidor.

Na diagrama da arquitetura proposta é possível verificar, de uma forma geral, a relação entre todos estes componentes.

O processo de *failover*, bem como a reação da instância a este, pode ser sucintamente explicado da seguinte forma:

- A instância primária falha. Através de mecanismos próprios (*heartbeats*, por exemplo) é detetado que esta se encontra não-responsiva e, por isso, o *failover* é iniciado;
- De seguida, a instância *standby* prossegue a uma reconexão, passando este agora a fornecer dados da zona secundária. O endereço IP é estático e igualmente partilhado com a instância primária, não sendo então necessário alterar nada em todo o sistema.

3.2 Implementação na *Google Cloud Platform*

3.2.1 Observações acerca da GCP

De forma a garantir uma coerência no que toca aos recursos usados, tendo em vista a limitação de crédito, todas as máquinas virtuais usadas na proposta de solução são iguais à instância usada na análise inicial, que se encontra detalhada na subsecção 2.3.2. Estas máquinas, por sua vez, são todas geradas a partir de um único *instance template* definido pelo grupo.

Acrescenta-se que, conforma sugerido pelo enunciado, bem como para garantir a melhor solução do ponto de vista de alta disponibilidade, foram utilizados vários serviços extra da *Google Cloud Platform*, que serão devidamente detalhados de seguida.

3.2.2 Ferramentas de configuração automática

A implementação desta arquitetura divide-se em dois momentos: uma parte toda automática, utilizando, mais uma vez, a ferramenta *Ansible*, e outra parte mais manual, passando esta pela interface gráfica da *Google Cloud Platform*.

As fases inteiramente automáticas podem ser sintetizadas no seguinte:

1. A instalação e correta configuração da Base de Dados na GCP, juntamente com o seu gestor relacional *PostgreSQL*, bem como das respetivas *Read-Replicas*, é feita utilizando apenas um único *playbook*;
2. O *template* que é utilizado nas máquinas virtuais presentes nesta solução é, também ele, instalado na GCP a partir de um *playbook*;
3. Uma vez criadas as máquinas virtuais, o processo de instalação e devida configuração do *Wiji.js* nas mesmas é feito através de um *playbook*. De modo a simplificar bastante este processo de *deployment*, tirou-se partido da *script* do inventário dinâmico, uma das soluções do *Ansible* para quando se está a trabalhar em ambientes de computação em nuvem. Desta forma, o "problema" de não ter IP's estáticos fica resolvido, pois o *script* encontrar-se-á não só sempre atualizado, mas também devidamente dividido em grupos previamente definidos, para fins de eficiência no que toca a divisão de *hosts*.

Acrescenta-se ainda que, para uma maior legibilidade do código, bem como facilitar mudanças futuras no que toca a campos como nome, tipo de máquina a ser criada, bem como todas as respetivas características, foi criada uma pasta "vars" que agrupa, em diversos ficheiros, todas as variáveis usadas nos demais *playbooks*.

Embora este enunciado tivesse como objetivo uma instalação e configuração automática de todos os componentes o grupo assume que, devido a vários fatores, não foi possível atingir esta "plenitude de automação". Um dos fatores deve-se, por exemplo, à dificuldade em encontrar em encontrar informação, ou até mesmo exemplos sobre a conjugação das ferramentas *Ansible* e GCP. Outro dos problemas é que, na perceção do grupo, a própria documentação se encontra desatualizada, faltando campos importantes para uma correta configuração daquilo que se pretendia.

Desta forma, foram então realizadas na interface gráfica da *Cloud* os seguintes passos manuais:

- A criação de um grupo de instâncias, a partir do *template* definido pelo grupo, cujos campos se encontram devidamente configurados como, por exemplo, a política de *autoscaling*;

- A criação de um balanceador de carga HTTP, associado ao grupo anterior, em que são definidos os campos *front-end* e *back-end*. Neste último é ainda definido o critério de verificação de integridade (*Health Check*).

3.2.3 Regras de *Firewall*

Visto que os diversos serviços utilizados comunicam por portas diferentes, é necessário abrir estas na *Firewall* da rede virtual da GCP, pois, por *default*, estas encontram-se bloqueadas por questões de segurança.

De seguida podemos verificar as portas configuradas pelo grupo:

<input type="checkbox"/>	Name	Type	Targets	Filters	Protocols / ports	Action
<input type="checkbox"/>	porta3000	Ingress	Apply to all	IP ranges: 0.0.0.0/0	tcp:3000	Allow
<input type="checkbox"/>	porta5432	Ingress	Apply to all	IP ranges: 0.0.0.0/0	tcp:5432	Allow
<input type="checkbox"/>	porta5601	Ingress	Apply to all	IP ranges: 0.0.0.0/0	tcp:5601	Allow
<input type="checkbox"/>	porta80	Ingress	Apply to all	IP ranges: 0.0.0.0/0	tcp:80	Allow
<input type="checkbox"/>	porta9200	Ingress	Apply to all	IP ranges: 0.0.0.0/0	tcp:9200	Allow

Figura 3.2: Regras de *Firewall* definidas

De forma sucinta, acrescenta-se que as portas abertas correspondem a:

- **3000:** onde o servidor *Node.js* da plataforma se encontra à escuta;
- **5432:** onde o *daemon* de *PostgreSQL* se encontra à escuta;
- **5601:** necessária para o bom funcionamento do Kibana;
- **80:** embora possa ser omitida no pedido HTTP, por ser a *default*, esta porta é usada na *Front-End* do *LoadBalancer*;
- **9200:** necessária para o bom funcionamento do ElasticSearch.

3.2.4 Balanceador de Carga

Todas as instâncias de máquinas virtuais são controladas pela rede interna da *Google* pelo que, cada vez que estas são criadas, é-lhe atribuído um endereço IP. Tendo em conta a inviabilidade de trabalhar com estes IP's efémeros, bem como o facto de cada instância *web-server* ter o seu

endereço IP, foi necessário recorrer a uma ferramenta da GCP, que permite a configuração fixa de um endereço IP para ser utilizado, "aproveitando-se" dos IP's efêmeros implementados na sua rede.

Esta ferramenta trata-se, então, dos **balanceadores de carga HTTP**, presentes em *Network Services*, cujo objetivo passa por associar um IP virtual fixo, que atua como "endereço de gestor" para os grupos de instâncias de máquinas virtuais, levando assim a que toda a gestão do balanceamento de carga seja feito pela *Google*. Os benefícios que levaram à utilização desta ferramenta passam pelo seguinte:

- A responsabilidade de uma boa distribuição de carga, para uma determinada configuração, fica a cargo da *Google*;
- *Front-end* possui IP fixo, resolvendo desta forma a questão da volatibilidade de IP's das várias máquinas que se encontram na *back-end*;
- A existência de uma opção *Health-Check*, responsável pela verificação de integridade da porta fornecida, usando, para isso, determinados intervalos de tempo. Este verificador permite identificar se uma máquina está a ter erros e, caso aconteça, esta última é removida do *back-end*, mantendo as restantes máquinas funcionais, garantindo assim a alta disponibilidade (*High availability*) do sistema.

Acrescenta-se que este balanceador de carga está associado um grupo de instâncias (agrupamento de várias máquinas virtuais) do servidor *Node.js*, que será visto em detalhe na subsecção seguinte. Note-se, analisando a figura seguinte, que existem **3/3 instâncias** íntegras e, desta forma, garante-se que existam **3 instâncias** do servidor a correr de forma simultânea.

lb-wiki-ascn

Faster web performance and improved web protection with Cloud CDN and Cloud Armor. [Learn more](#)

DETAILS

MONITORING

CACHING

Frontend

Protocol ↑	IP:Port	Certificate	SSL Policy ↑	Network Tier ?
HTTP	34.120.92.66:80	-		Premium

Host and path rules

Hosts ↑	Paths	Backend
All unmatched (default)	All unmatched (default)	wiki-ascn-backend

Backend

Backend services

1. wiki-ascn-backend

Endpoint protocol	Named port	Timeout	Health check	Cloud CDN
HTTP	http	30 seconds	hc-backend	Disabled

ADVANCED CONFIGURATIONS

Name ↑	Type	Zone	Healthy	Autoscaling	Balancing mode	Selected ports ?	Capacity
ig-webs-ascn	Instance group	europe-west1	3 of 3	On: Target LB capacity fraction 80%	Max backend utilization: 80%	3000	100%

Figura 3.3: Balanceador de carga HTTP

3.2.5 Grupo de Instâncias

A decisão de aplicar um balanceador de carga só faz sentido se se pensar num agrupamento de várias máquinas virtuais (Grupo de Instâncias), algo facilmente aplicável na *Google Cloud Platform*.

Por isso, foi criado, a partir do *template* definido pelo grupo e já detalhado nesta secção, um grupo de instâncias, contendo todas as instâncias do servidor *Node.js*. Este grupo terá, então, um balanceador de carga HTTP associado, que já foi explicado ao detalhe.

Note-se que o número de instâncias de máquinas virtuais que este grupo abrange é facilmente escalável, podendo ir até dez máquinas, o que, em casos mais extremos a nível de afluência poderia ser concretizado bastando, para isso, ter um valor de crédito maior. O grupo decidiu que, para efeitos deste projeto, três instâncias do servidor bastariam para alcançar os objetivos pretendidos.

Na figura seguinte é possível ver, na interface gráfica da GCP, o resultado da sua configuração:

Instances by status
3 instances
 3

Instance by health
Not configured
Autohealing off. [Configure](#)

Autoscaling
On (min 3, max 4)
Predictive autoscaling off. [Change](#)

Status Ready
Creation Time Jan 6, 2022, 5:57:59 PM UTCZ
Description
Number of instances 3
Template templateteste
Location europe-west1
In use by [lb-teste](#)

Instance Group Members REMOVE FROM GROUP DELETE INSTANCE


Filter Enter property name or value

<input type="checkbox"/> Status	Name	Creation Time	Template	Per instance config	Internal IP	External IP	Health Check Status	Connect
<input type="checkbox"/>	wiki-interface-ig-4n9w	Jan 6, 2022, 5:58:11 PM UTCZ	templateteste		10.132.0.18 (nic0)	34.79.68.230		SSH
<input type="checkbox"/>	wiki-interface-ig-dqsf	Jan 6, 2022, 5:58:14 PM UTCZ	templateteste		10.132.0.19 (nic0)	35.195.171.231		SSH
<input type="checkbox"/>	wiki-interface-ig-zx0x	Jan 6, 2022, 5:58:11 PM UTCZ	templateteste		10.132.0.17 (nic0)	34.140.231.196		SSH

Figura 3.4: *Instance Group* associado ao balanceador HTTP

3.2.6 SQL *instance* e respectivas réplicas

Estando o processo automático de criação terminado, é possível verificar na interface gráfica da GCP todas as informações relativas não só à base de dados *master* (como o IP público de acesso, por exemplo), mas também das respectivas *Read Replicas*, o que se pode confirmar na figura seguinte.




SQL

Instances


[+ CREATE INSTANCE](#)


[⇄ MIGRATE DATA](#)

[SHOW INFO PANEL](#)

 Filter

Enter property name or value



















<input type="checkbox"/>	Instance ID  	Type	Public IP address	Private IP address	Instance connection name	High availability	Location	Actions
<input type="checkbox"/>	  wikijs-db	PostgreSQL 13	35.196.77.33		g28ascn2021:us-eas... 	ENABLED	us-east1-d	
<input type="checkbox"/>	 wikijs-db-readreplica1	PostgreSQL read replica	34.135.219.96 		g28ascn2021:us-cen... 	N/A	us-central1-k	
<input type="checkbox"/>	 wikijs-db-readreplica2	PostgreSQL read replica	35.223.156.56 		g28ascn2021:us-cen... 	N/A	us-central1-k	

Figura 3.5: Instâncias SQL

Acrescenta-se que, do ponto de vista de escalabilidade, as *Read Replicas* representam um componente que é bastante escalável, podendo assim serem facilmente replicadas, tornando assim as operações de leitura mais eficientes. Quanto maior o número de réplicas, menor é o tempo necessário para efetuar uma operação de leitura.

Para implementar esta eficiência entre pedidos de leitura/escrita e o encaminhamento destes para o complexo "base de dados *master* + réplicas de leitura" é necessário um mecanismo PROXY que faça toda esta gestão de forma eficiente, algo que infelizmente o grupo, apesar de ter uma ideia e de a ter implementado, não conseguiu que esta ficasse funcional.

3.2.7 PROXY SQL - PGpool

De forma a tornar o sistema mais eficiente, surgiu a ideia de criar um balanceador de pedidos à base de dados que, por um lado, encaminhasse os pedidos de leitura às bases de réplica e, por outro, os pedidos de escrita à base de dados primária. Assim, o processo passaria por criar um *instance group* onde tivessem várias máquinas a correr o *pgpool*, que, por sua vez, encaminhariam os pedidos de leitura e escrita na base de dados para a base de dados correspondente.

Esta solução tornaria o sistema mais eficiente porque aliviaria alguma carga da base de dados primária, tirando proveito das réplicas. Nunca seria um **SPOF** uma vez que o *instance group* se asseguraria de garantir que as máquinas estivessem sempre disponíveis. Infelizmente, não foi possível pôr esta solução a funcionar corretamente, apesar de estar praticamente totalmente implementada.

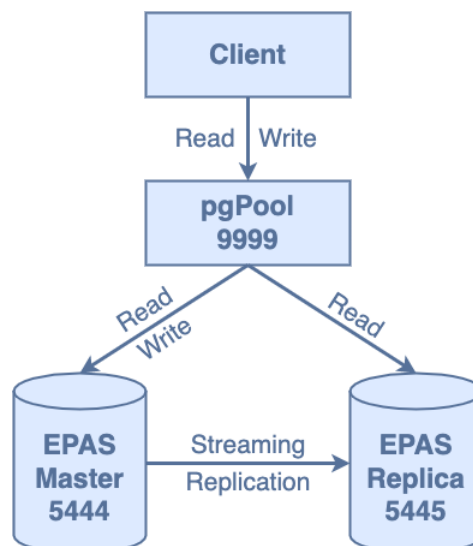


Figura 3.6: Proxy pgpool - Arquitetura idealizada mas por implementar

3.3 Visualização da Arquitetura

De seguida, é possível verificar a arquitetura proposta:

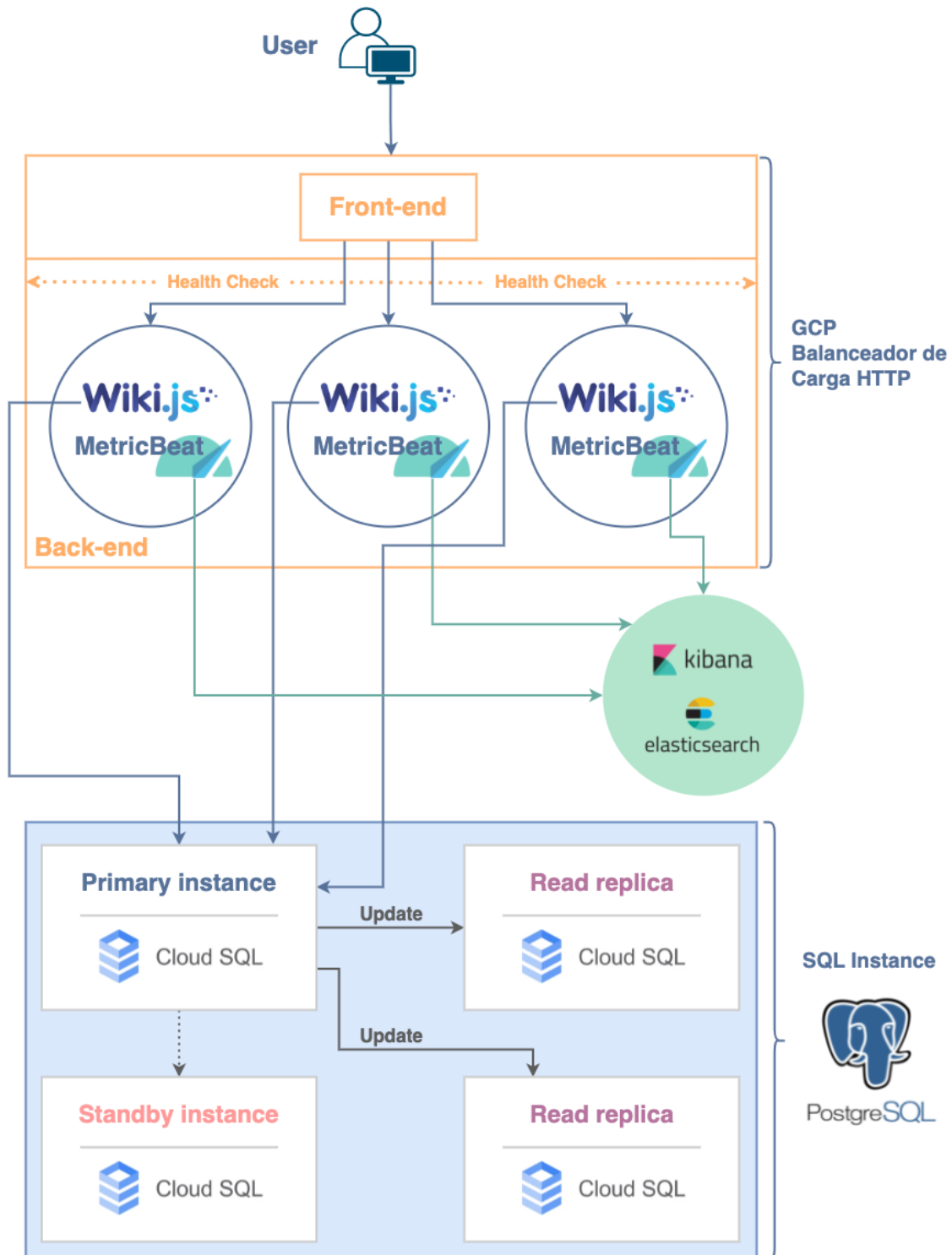


Figura 3.7: Arquitetura proposta

3.4 Ferramentas de Monitorização

A terceira etapa consistia em dotar a aplicação de ferramentas de monitorização, a fim de poder retirar conclusões de um ambiente de testes. O grupo decidiu utilizar duas estratégias distintas para monitorizar toda a aplicação: para os *web-servers* foi implementada uma *ELK Stack* (tal como nos guiões práticos), já para a componente da Base de Dados foram utilizadas as ferramentas de monitorização disponibilizadas pela *Google Cloud Platform*.

3.4.1 *Web-servers*

A instalação de todos os componentes necessários ao bom funcionamento desta subsecção foi feita de forma automática, pelo que, conforme se pode visualizar na **figura X**, a configuração das ferramentas de monitorização passa essencialmente por duas partes: a primeira passa pela instalação, numa máquina virtual (chamada "monitor") criada para o efeito, do *ElasticSearch* e do *Kibana*. A segunda parte, por sua vez, consiste em dotar todos os servidores *Node.js* com a ferramenta *MetricBeat*, de forma a que seja possível, no monitor, consultar a todas as métricas pretendidas relativas às diversas instâncias *Wiki.js*.

Quanto aos componentes ***ElasticSearch*** e ***Kibana***, o respetivo *playbook* da sua instalação tem como pontos importantes o seguinte:

1. A criação da máquina virtual na GCP, bem como das respetivas configurações;
2. Tanto para a instalação do *ElasticSearch* como do *Kibana* foi utilizado o repositório de ***Ansible Roles***, mais conhecido como ***Ansible Galaxy***, permitindo, assim, uma rápida incorporação dos respetivos *Roles* no *playbook*. Esta decisão não só permitiu que o código em questão fosse mais legível, mas também permitiu diminuir a complexidade da automatização completa deste processo de monitorização.

O ficheiro de automação referente ao ***MetricBeat*** segue o estilo do que foi feito nas aulas práticas da disciplina, tendo na sua composição, de forma detalhada, todos os passos necessários ao seu funcionamento. Acrescenta-se que, de modo a que este *playbook* seja executado apenas uma vez é feito usufruto das propriedades do inventário dinâmico.

3.4.2 Base de Dados

Para monitorizar toda a componente relativa à base de dados foram aproveitadas todas as ferramentas que a *Google Cloud Platform* fornece para o efeito, na interface gráfica, pois estas são bastante completas e permitem fazer uma análise rigorosa:

- Para avaliar a *performance* da Base de Dados, baseada em diversas métricas (CPU, I/O, etc.) e de modo a detetar possíveis anomalias, foi feito usufruto da ferramenta **Monitoring** que, nos momentos dos testes, nos permitiu avaliar cuidadosamente todos estas métricas de forma clara. De seguida podemos ver um exemplo desta ferramenta a ser utilizada:

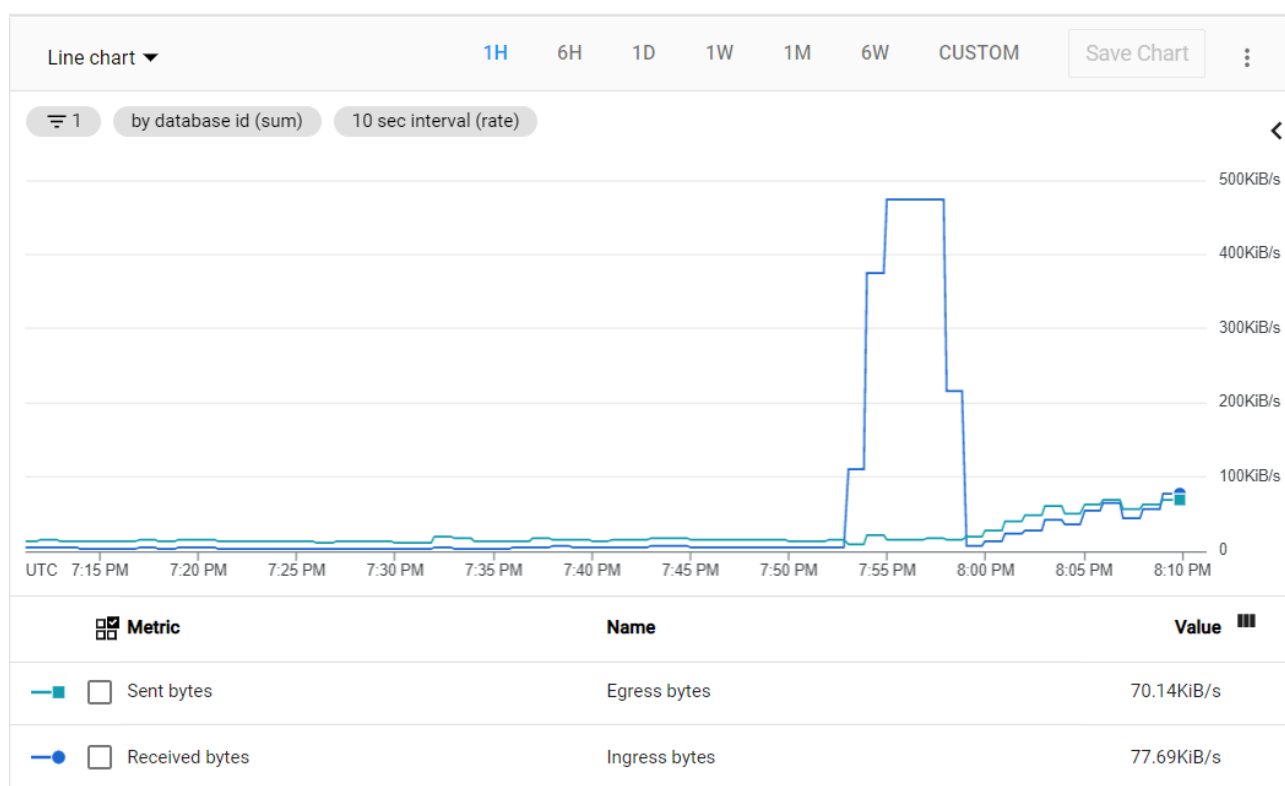


Figura 3.8: Gráfico referente a *Egress/Ingress bytes*

- Para avaliar toda a componente que envolve as diversas *queries* realizadas, foi utilizada também a ferramenta **Query Insights**, presente no menu referente à instância SQL. De seguida pode-se verificar um pequeno excerto da utilização desta ferramenta:

QUERIES

TAGS

Filter

Filter queries

?

Query	Database	Load by total time	Avg execution time (ms)	↓ Times called	Avg rows returned
select "pages"."id", "pages"."path", "pages"."hash", "pages"."title", ...	ivo	<div></div>	0.08	1,279	0
UTILITY COMMAND	ivo	<div></div>	9.64	813	0
select * from information_schema.columns where table_name...	ivo	<div></div>	3.13	416	13

Figura 3.9: Utilização da ferramenta *Query Insights*

3.5 Testes de Desempenho

De forma a poder estabelecer uma comparação com a análise inicial à arquitetura simples, aplicaram-se exatamente os mesmos testes à arquitetura proposta.

3.5.1 Teste 1: Página Inicial

Este teste é composto pela operação:

1. GET /

Os resultados obtidos encontram-se sintetizados na seguinte tabela:

Threads	Average	Min	Max	Std. Dev	Error %	Throughput
500	79	69	391	23,00	0,00%	49,70
1000	79	67	236	14,29	0,00%	49,90
5000	90	45	1255	71,86	0,43%	43,30
10000	79	66	393	17,59	0,00%	35,50

Figura 3.10: Resultados estatísticos do Teste 1 - Arquitetura complexa

3.5.2 Teste 2: Registrar um novo Utilizador

Este teste é composto pelas operações:

1. GET /

2. POST /graphql

NOTA: *Body Data* presente em Anexos ??;

Os resultados obtidos encontram-se sintetizados na seguinte tabela:

Threads	Average	Min	Max	Std. Dev	Error %	Throughput
500	5866	84	12763	3013,43	0,23%	27,70
1000	6239	84	17619	4190,52	0,59%	22,90
5000	10472	81	30047	10713,20	19,24%	11,30
10000	13487	86	30068	11798,49	31,74%	29,30

Figura 3.11: Resultados estatísticos do Teste 2 - Arquitetura complexa

3.5.3 Teste 3: *Login*

Este teste é composto pelas operações:

1. GET /login

2. POST /graphql

NOTA: *Body Data* presente em Anexos A.2;

Os resultados obtidos encontram-se sintetizados na seguinte tabela:

Threads	Average	Min	Max	Std. Dev	Error %	Throughput
500	64	55	200	19,29	0,00%	49,80
1000	64	55	374	17,86	0,00%	49,90
5000	61	53	362	11,81	0,00%	50,00
10000	62	53	378	12,57	0,00%	50,00

Figura 3.12: Resultados estatísticos do Teste 3 - Arquitetura complexa

3.5.4 Teste 4: *Login + Criar Página*

Este teste é composto pelas operações:

1. GET /login

2. POST /graphql

NOTA: *Body Data* presente em Anexos A.2;

3. GET /e/en/new-pages

4. POST /graphql

NOTA: *Body Data* presente em Anexos A.3;

Os resultados obtidos encontram-se sintetizados na seguinte tabela:

Threads	Average	Min	Max	Std. Dev	Error %	Throughput
500	9094	41	31378	12167,18	21,63%	29,20
1000	11093	38	31392	13134,05	30,23%	34,30
5000	11767	39	30608	11841,06	27,41%	44,30
10000	11290	85	30333	11736,68	23,65%	36,40

Figura 3.13: Resultados estatísticos do Teste 4 - Arquitetura complexa

3.6 Análise de Resultados

Após efetuar todos os testes, bem como uma cuidada monitorização, decidiu-se analisar mais a fundo os resultados obtidos com o Teste 4, nomeadamente tentar justificar o porquê daqueles valores de erros.

Começando por avaliar a utilização do CPU na base de dados, facilmente se verifica que os seus valores são bastante baixos e, por isso, o que nos leva a acreditar que o problema se encontra nos servidores.

A figura seguinte corresponde à variação da utilização do CPU nos momentos antes e durante o referido teste.

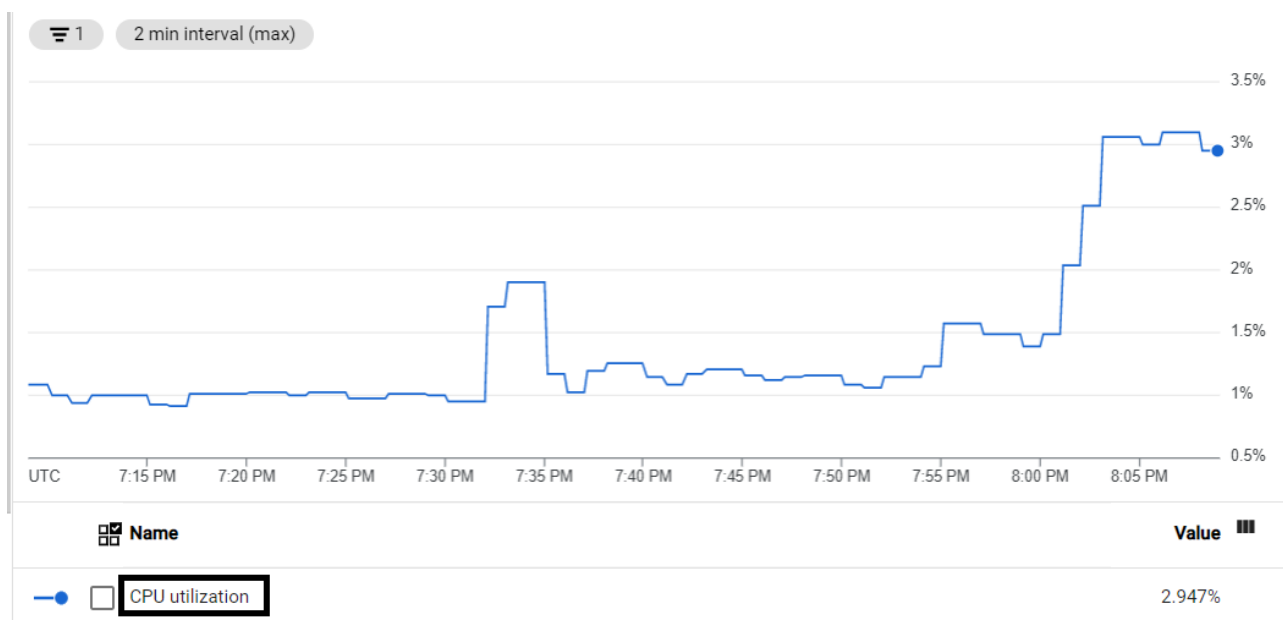


Figura 3.14: Utilização de CPU da BD no Teste 4

De seguida, consultou-se os resultados provenientes da monitorização constante dos servidores *Node.js*, durante a realização do Teste 4:

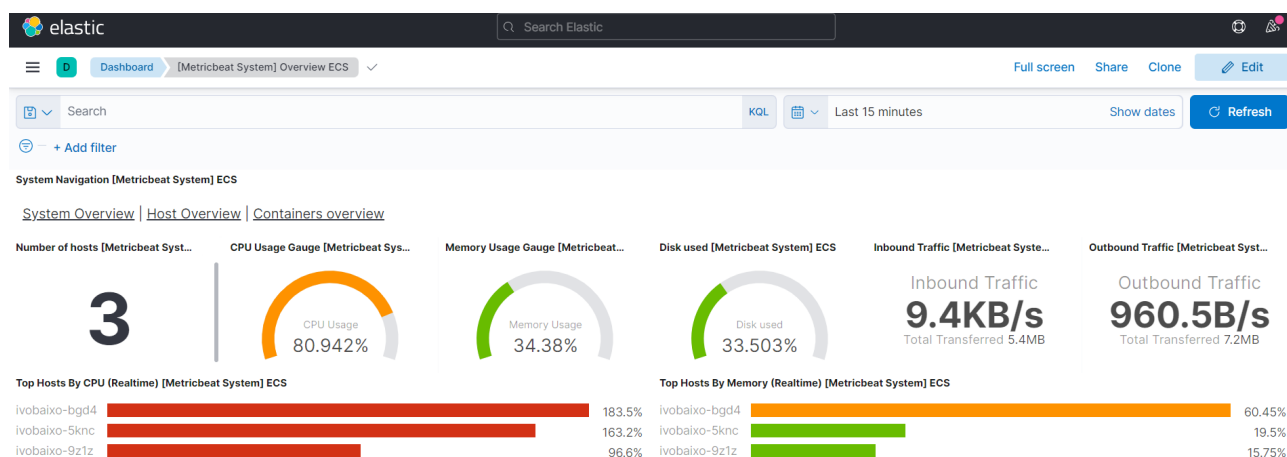


Figura 3.15: Utilização de CPU dos servidores no Teste 4

São apresentadas várias métricas, embora a que salte mais à vista é, sem dúvida, a utilização do CPU, em que esta atinge valores bastante altos em todas os servidores chegando a atingir, pelo menos em duas máquinas, um valor superior a 100% do mesmo, entrando assim em situações de *overload*. Com isto, é fácil perceber que isto causa a que os pacotes sejam descartados e, por isso, se tenha obtido valores consideráveis no campo *Error*.

Concluindo, o CPU, por estar a ser totalmente utilizado, não consegue lidar com um número tão elevado de pedidos em simultâneo o que leva a que muitos pacotes sejam descartados da fila de espera.

Por último, e de forma a estabelecermos um contraste no que toca a estas métricas, decidiu-se recorrer às mesmas para avaliar a performance dos servidores aquando a execução do Teste 1, o mais simples dos quatro. Sem surpresa, verifica-se que os valores de utilização de CPU, em todas as máquinas, são significativamente menores, não passando os 5% da sua utilização, o que vai de encontro aos pressupostos do grupo.

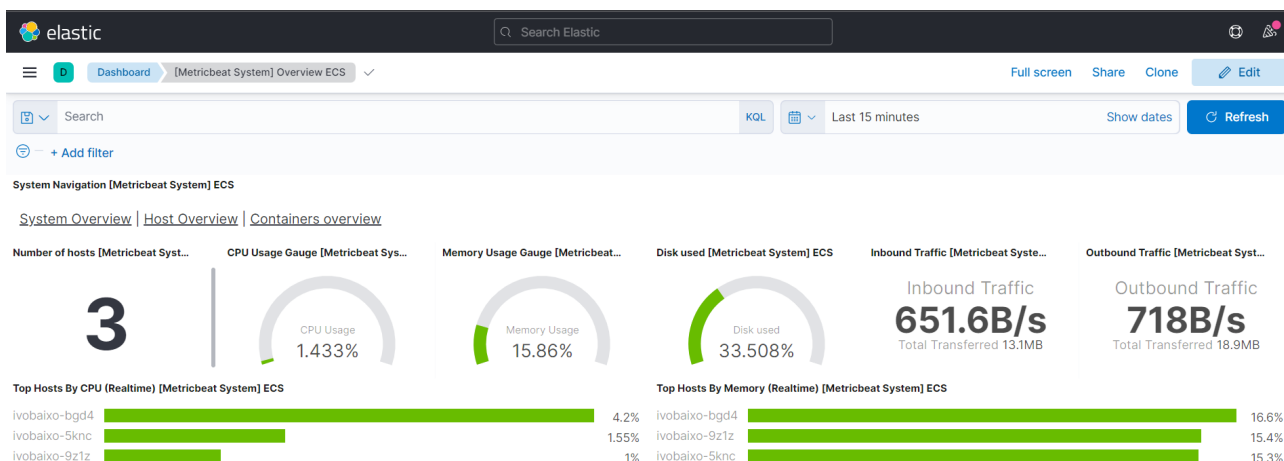


Figura 3.16: Utilização de CPU dos servidores no Teste 1

Por último, e por curiosidade, foi também analisada a variação do uso da Memória numa das instâncias de *web-servers* nos momentos em que é realizado o Teste 4 (até 20:12:00), e de seguida o Teste 1 (a partir de 20:12:00). Mais uma vez, facilmente se vê o diferente impacto que ambos os testes têm nas instâncias.

Memory Usage [Metricbeat System] ECS

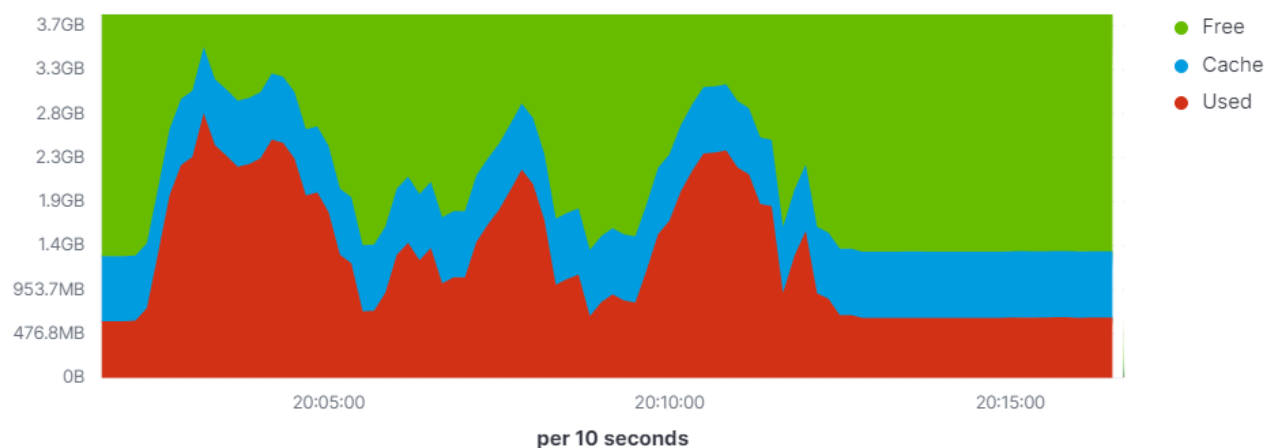


Figura 3.17: Variação de memória usada de um dos servidores

4. Conclusão

De um modo geral, o sentimento que passa é que este trabalho prático, além de extremamente desafiante, ajudou bastante a consolidar a matéria lecionada nas aulas da disciplina. Reconhecemos que foi muito enriquecedor para o nosso coletivo, pois permitiu que cimentássemos todas as competências não só da parte relativa à automatização de processos, mas também da parte relativa à importância de garantir a alta disponibilidade e desempenho de uma aplicação, bem como a prevenção de perdas de dados.

Assumimos que houve alguma dificuldade na ligação de todos os conceitos, pois, tendo o enunciado quatro etapas (e cada uma com conceitos distintos) estamos a falar de uma solução que envolve um grande número de componentes.

Todo o projeto foi implementado tendo em vista uma escalabilidade futura, com a garantia de que com mais posses financeiras seria facilmente possível a expansão do sistema em praticamente todos os seus componentes: *web-servers* e réplicas de leitura, bem como outros serviços aplicacionais.

Em forma de nota, o grupo pretende também referir a importância que um bom estudo da *Google Cloud Platform* teve pois, com este, foi possível utilizar um grande leque de ferramentas que facilitaram e tornaram possível a nossa solução. Além disto, foi também necessária uma conexão direta entre "como encaixar" o que foi abordado nas aulas práticas na GCP. Algo moroso, mas com bons resultados visíveis.

Todas as decisões tomadas pelo grupo, desde as estratégias escolhidas até traços fundamentais da implementação, foram sempre discutidas entre todos, pelo que o grupo acha que a solução final pode ser considerada bastante válida.

Por fim, falar de um sentimento de tristeza por, apesar de quase todos os objetivos terem sido cumpridos, também houve alguns que foram defraudados, muito por falta de tempo, como a não automatização completa de todos os componentes, bem como a não implementação do gestor de réplicas de leitura.

Em suma, o esforço foi grande com o intuito de garantir boas soluções para o enunciado proposto.

A. Anexos

A.1 Ficheiro JSON relativo a registar um novo Utilizador

```
1  [{
2    "operationName":null,
3    "variables":{"providerKey":"local",
4      "email":"${email}@mail.com",
5      "passwordRaw":"wikiwiki","name":"${name}",
6    "groups":[],"mustChangePassword":false,"sendWelcomeEmail":false
7  },
8  "extensions":{},
9  "query":"mutation ($providerKey: String!, $email: String!,
10    $name: String!,
11    $passwordRaw: String, $groups: [Int]!,
12    $mustChangePassword: Boolean, $sendWelcomeEmail: Boolean) {
13    users {
14      create(providerKey: $providerKey, email: $email, name: $name,
15        passwordRaw: $passwordRaw, groups: $groups, mustChangePassword:
16        $mustChangePassword, sendWelcomeEmail: $sendWelcomeEmail) {
17        responseResult {
18          succeeded
19          errorCode
20          slug
21          message
22          __typename
23        }
24        __typename
25      }
26      __typename
27    }
28  }"}
29  ]]
```

A.2 Ficheiro JSON relativo ao *login*

```
1  [{
2    "operationName":null,
3    "variables":{"username":"${username}","password":"wikiwiki",
4    "strategy":"local"},
5    "extensions":{},
6    "query":"mutation ($username: String!,
7    $password: String!, $strategy: String!)
8    {
9    authentication {
10    login(username:$username,password: $password, strategy: $strategy){
11
12        responseResult {
13            succeeded
14            errorCode
15            slug
16            message
17            __typename
18        }
19        jwt
20        mustChangePwd
21        mustProvideTFA
22        mustSetupTFA
23        continuationToken
24        redirect
25        tfaQRImage
26        __typename
27    }
28    __typename
29    }
30    }"
31  }]
```

A.3 Ficheiro JSON relativo à criação de uma página

```

1  [{
2      "operationName": null,
3      "variables": {"content": "<h1>Title</h1>\n\n<p>Some text here</p>",
4      "description": "descricao",
5      "editor": "code", "locale": "en", "isPrivate": false, "isPublished": true,
6      "path": "${path}", "publishEndDate": "", "publishStartDate": "",
7      "scriptCss": "", "scriptJs": "", "tags": [], "title": "Um titulo"},
8      "extensions": {}, "query": "mutation ($content: String!,
9      $description: String!, $editor: String!, $isPrivate: Boolean!,
10     $isPublished: Boolean!, $locale: String!, $path: String!,
11     $publishEndDate: Date, $publishStartDate: Date,
12     $scriptCss: String, $scriptJs: String,
13     $tags: [String]!, $title: String!) {
14         pages {
15             create(content: $content, description: $description,
16                 editor: $editor, isPrivate: $isPrivate,
17                 isPublished: $isPublished, locale: $locale,
18                 path: $path, publishEndDate: $publishEndDate,
19                 publishStartDate: $publishStartDate,
20                 scriptCss: $scriptCss, scriptJs:
21                 $scriptJs, tags: $tags, title: $title) {
22
23                 responseResult {
24                     succeeded
25                     errorCode
26                     slug
27                     message
28                     __typename
29                 }
30                 page {
31                     id
32                     updatedAt
33                     __typename
34                 }
35                 __typename
36             }
37             __typename
38         }
39     }
40 }]
```