Universidade do Minho
Escola de Engenharia

Projeto de System Deployment & Benchmarking

Instalação Zulip

Relatório de Desenvolvimento

José André Martins Pereira
(a82880@alunos.uminho.pt)

Ricardo André Gomes Petronilho
(a81744@alunos.uminho.pt)

Miguel Raposo Dias
(pg41089@alunos.uminho.pt)

Ricardo Cunha Dias
(pg39295@alunos.uminho.pt)

30 de Dezembro de 2019

Conteúdo

Capítulo 1

Introdução

Na unidade curricular de **Systems Deployment & Benchmarking** foi-nos proposto um trabalho prático de análise, deployment e benchmarking de uma aplicação.Ao longo deste relatório apresentam-se os processos efetuados para a concretização do projeto.

Inicialmente faz-se uma contextualização do problema e apresentação dos objetivos, descreve-se a arquitetura e componentes da aplicação, identifica-se e apresenta-se as ferramentas utilizadas na instalação e deployment da mesma.

Por fim, e não menos importante, identificam-se as ferramentas utilizadas na monitorização e benchmark, bem como se faz uma análise dos resultados obtidos.

Capítulo 2

Contextualização do Problema

2.1 Objetivos

Os objetivos centrais deste projeto são analisar a arquitetura de uma aplicação, para se perceber os possíveis pontos críticos, e que soluções existem, para reduzir o impacto dos mesmos. Com a resolução destes problemas, consegue-se uma maior disponibilidade da aplicação. Do mesmo modo, monitorizar e analisar o desempenho da aplicação, para diferentes configurações dos componentes, para se entender o impacto que estas tem no desempenho da aplicação.

2.2 Aplicação

Figura 2.1: Logotipo do Zulip.

Assim, a aplicação escolhida pela equipa foi o **Zulip**, na medida em que segue os requisitos do trabalho prático, bem como, tem uma boa documentação.
O **Zulip** é uma aplicação de chat em tempo real, onde o principal objetivo é oferecer um boa experiência a organizações, empresas e projetos voluntários, de pequenas equipas de amigos, até dezenas de milhares de utilizadores.

Capítulo 3

Arquitetura da aplicação Zulip

A arquitetura da aplicação **zulip** contém vários **componentes**, sendo que, se consegue dividir os mesmos em diferentes subsistemas. No entanto, é perceptível pela figura 3.1, que existe uma separação entre o componente **Client** com os restantes, pois são subsistemas distintos.

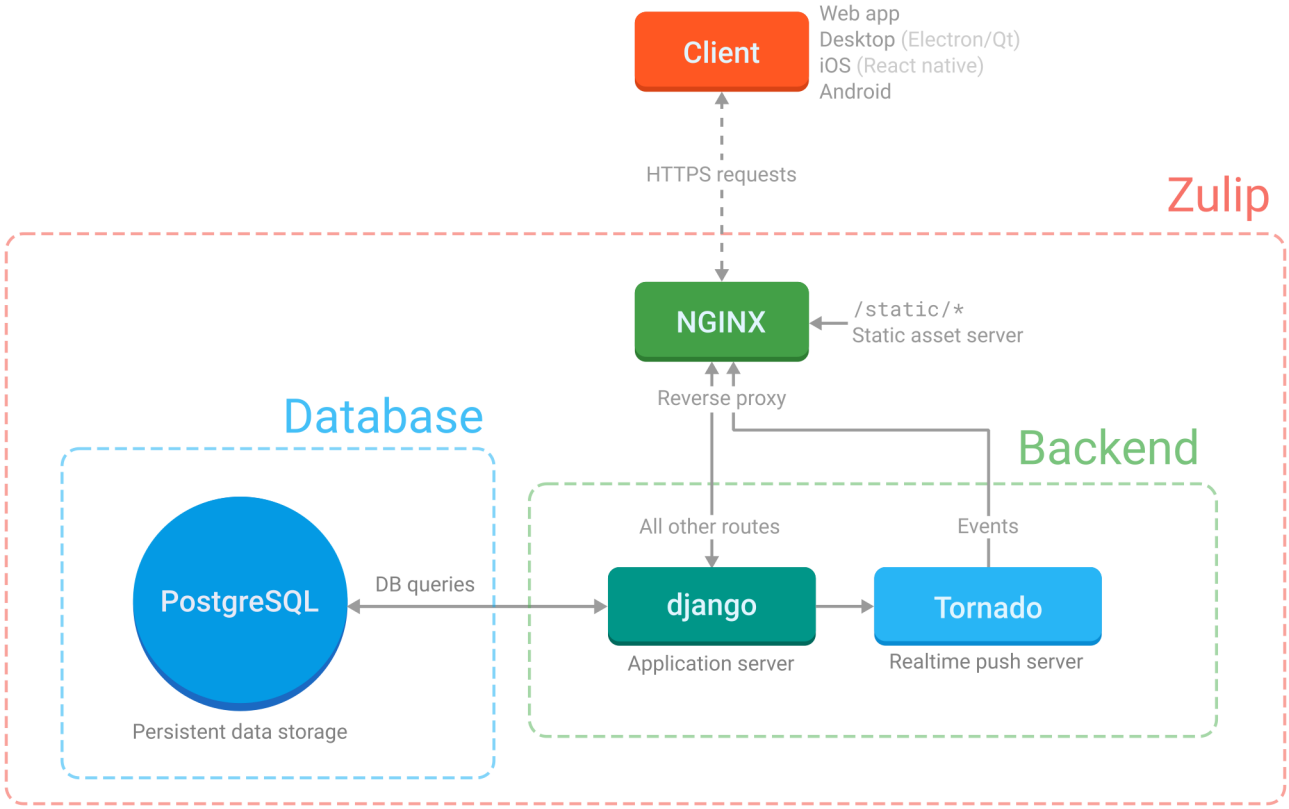

Figura 3.1: Arquitetura da aplicação Zulip.

O **Client** é responsável pela apresentação (*view*) da aplicação, enquanto que o restante subsistema é de uma forma geral, responsável pela lógica e tratamento de dados da mesma, sendo este mais complexo e composto por subsistemas interiores (**Database** & **Backend**).

A divisão também deve-se ao facto do componente **Client** estar na máquina do utilizador, enquanto que os restantes estarão em máquinas administradas pela empresa de desenvolvimento.

O componente **NGINX** consiste num servidor *HTTP* intermediário (*middleware*) entre o componente **Client** com *HTTP requests* e o **Backend**. Na verdade, o **NGINX** permite uma independência entre os subsistemas, ou seja, o **Client**, pode ser definido para diferentes plataformas, nomeadamente: web, desktop, iOS e android.

Em relação à base de dados da aplicação, tem-se o componente **PostgreSQL**, um sistema de gerenciamento de dados persistente que apenas comunica com o servidor da aplicação **django**.

Em termos do sistema de **backend** da aplicação, este é composto por duas componentes, sendo elas **django** e **Tornado**, este primeiro **django** é de extrema importância tendo em conta que é a base da construção da aplicação **zulip**, e consiste num framework de web do lado do servidor de código aberto, desenvolvido em Python, fornecendo assim todo o framework necessário para ser mais tarde apresentado do lado do cliente, quanto ao **Tornado** é um servidor de web assíncrono e sem bloqueio de I/o que permite manter ativas milhares de ligações em tempo real, no caso do **zulip** é respónsavel pela entrega de mensagens e não muito mais.

Outros componentes secundários igualmente importantes no funcionamento do **zulip** são **Supervisor** que é um sistema cliente/servidor que permite monitorizar e controlar os processos em sistemas **unix**, no caso do **zulip** é utilizado para iniciar os processos do servidor, reinicializar-los em caso de falha e fazer *logging*, outro é **memcached** é um sistema distribuído de cache em memória, é usado para guardar em chache modelos de dados e invalidar-los caso sejam alterados.

Um outro componente secundário é a base de dados em memória **Redis** que armazena chaves com durabilidade opcional, é usado para guardar dados com tempo de vida bastante baixo.
O **RabbitMQ** é um software de mensagens com código aberto, que é usado para tratar de filas que requerem uma entrega fiável mas não são passíveis de fazer no sistema principal, é também usado para comunicações entre **django** e **Tornado**.
Por último, o componente **Thumbor** é um serviço de miniaturas de fotos open-source que tem como função administrar as fotos do sistemas ora via **upload** ora via **url**.

Importante realçar que a arquitetura que será elaborada no deployment da aplicação, varia consoante o número de replicas de cada componente. No entanto isto será abordado com mais detalhe nos capítulos seguintes.

Capítulo 4

Instalação/Configuração da aplicação na Google Cloud

De seguida apresenta-se o processo de instalação e configuração dos componentes da aplicação, onde foram utilizadas as ferramentas: **Docker**, **Kubernetes**.

4.1 Vantagens da utilização de Containers

A utilização de containers na infraestrutura da aplicação **zulip**, permite um isolamento dos componentes, bem como um melhor aproveitamento dos recursos de cada máquina, facilitando o processo de teste, *provisining* e migração dos mesmos, sendo estes importantes num processo de deployment.

4.2 Abordagens

Com uma análise da arquitetura da aplicação **Zulip**, percebeu-se a existência de pontos críticos (**SPOFS**). Na verdade, não se deve conter num sistema componentes únicos, que em sua falta, causem a falha de todo o sistema.
Assim, para uma redução da probabilidade de acontecimento destes, achou-se que a solução seria a replicação dos componentes onde estes acontecem. Operações que façam acessos à base dados, ou apenas ao servidor aplicacional, estão dependentes destes componentes, sendo crucial a fiablidade dos mesmos. Assim, todos os componentes devem ser replicados, excepto o **NGINX**, visto a impossibilidade de replicação deste.
A replicação destes componentes também contribuí para uma melhor disponibilidade, visto que, há uma distribuiçao de carga pelas várias réplicas.

4.3 Docker

A ferramenta **Docker** consiste numa plataforma para *developers*, com o objetivo de construir, partihar e executar aplicações com *containers*. O uso desta ferramenta facilita o processo de **deployment**.
As grandes vantagens do **Docker** são:

- Flexibilidade: a complexidade da aplicação não impede a possibilidade de ser colocada num container;
- Leves/eficientes: os recursos e kernel da máquina são partilhados por todos os containers, havendo um melhor aproveitamento dos recursos;

- Portabilidade: pode ser construído localmente e fazer-se o *deploy* para a cloud facilmente;
- Escabilidade: com facilidade, consegue-se aumentar e distribuir as replicas de containers;
- etc ...

Na figura 4.1 verifica-se uma das grandes vantagens da ferramenta **Docker** em relação à utilização de **VMs** para agregação de diferentes aplicações na mesma máquina.
Tal como já foi referido, na utilização de máquinas virtuais (caso da direita da figura 4.1), há uma repetição de sistemas operativos desnecessária para cada aplicação, e deste modo, recursos usados desnecessariamente para estes sistemas opearativos.
Por outro lado, no **Docker**, existe apenas um sistema operativo, partilhado por todas as aplicações, havendo assim, um melhor aproveitamento dos recursos, pois estes sao distribuídos pelos vários containers.

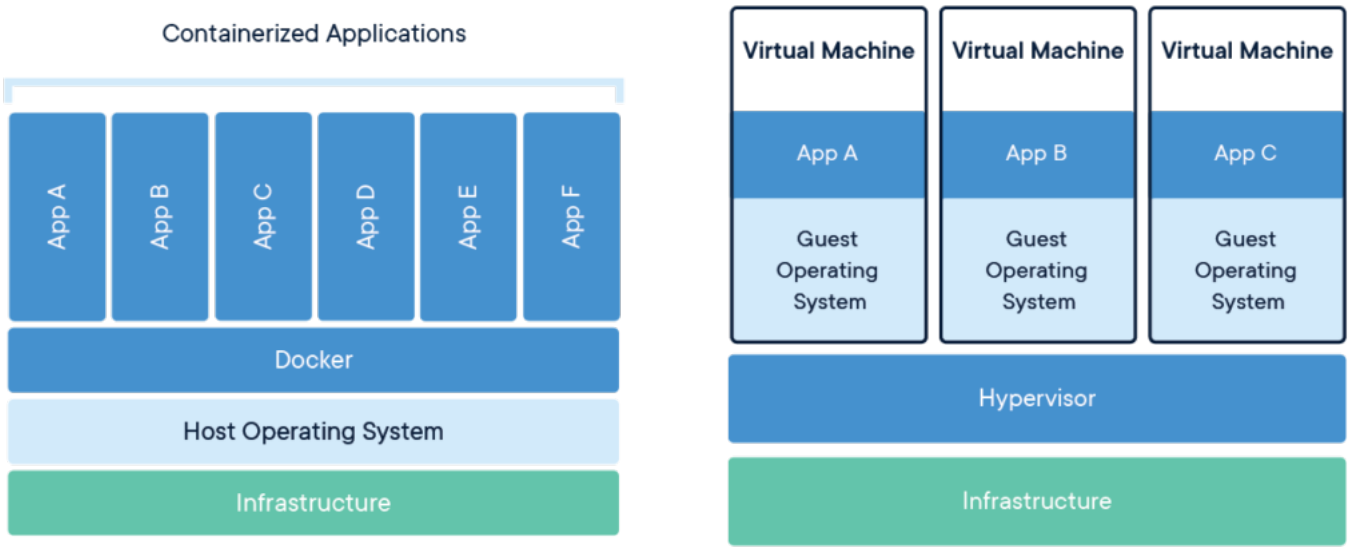

Figura 4.1: Docker vs VMs.

4.4 Kubernetes

Trata-se de uma plataforma de código aberto, que permite gerir aplicações em containeres e automatizar a sua implementação. **Kubernetes** introduz o conceito de pods que consiste em um ou mais containers. Com a utilização do **Kubernetes**, obtem-se um cluster. Um cluster é um conjunto de máquinas que correm aplicações em containers, que são geridas pelo **Kubernetes**. O cluster tem pelo menos um nodo (máquina) worker e um master, sendo que o worker corre os pods que são componentes da aplicação no nosso caso **Zulip**, e o master gere as máquinas worker e os pods do cluster, de notar que múltiplos nós master permite tolerância a falhas e alta disponibilidade. Neste caso decidiu-se utilizar um cluster com um master e quatro máquinas worker.

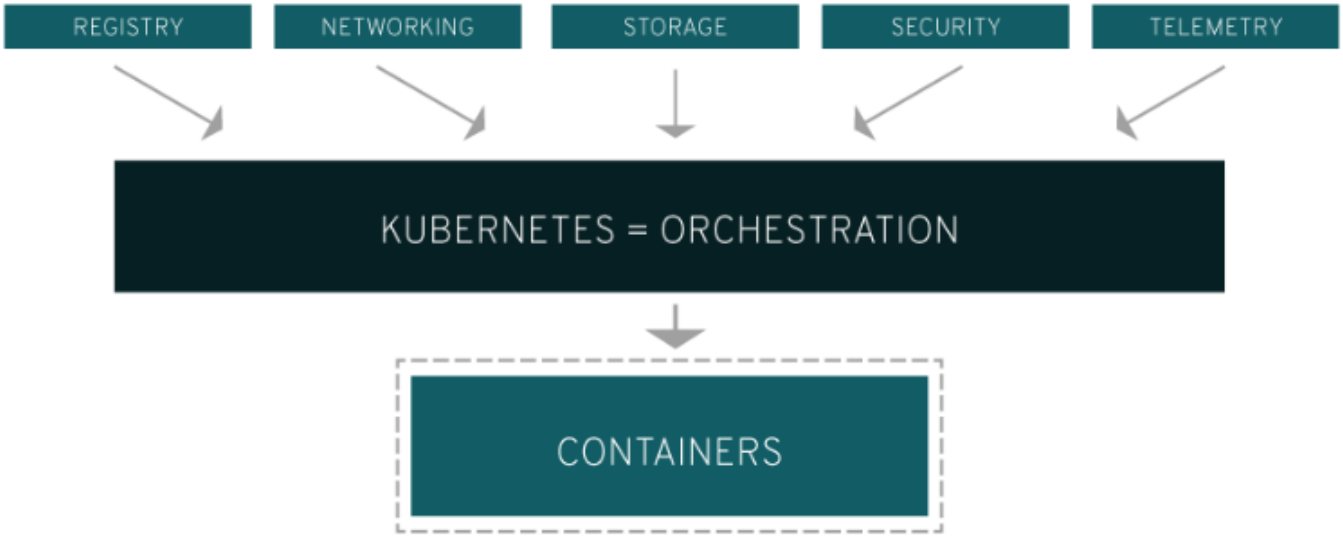

Figura 4.2: Arquitetura do **Kubernetes**.

4.5 Automatização do Deployment

A ferramenta **Kubernetes** facilita bastante a automatização do processo de **Deployment**, pois consegue-se definir as configurações dos componentes no ficheiro *zulip-rc.yml*, em relação ao número de replicas. Com o ficheiro de configuração preparado, apenas com o comando da figura 4.3 consegue-se fazer o **Deployment** para o cluster de **Kubernetes**, onde automáticamente, os componentes são distribuídos por containers, em diferentes nodos.

```
ricardo@ricardo-ubuntu:~/Downloads$ clear && kubectl apply -f zulip-rc.yml
```

Figura 4.3: Comando automático para **Deployment**.

```
ricardo@ricardo-ubuntu:~/Downloads$ kubectl get nodes
NAME                                        STATUS   ROLES    AGE    VERSION
gke-zulip-cluster-default-pool-02f7beb8-8q11   Ready    <none>   167m   v1.13.11-gke.14
gke-zulip-cluster-default-pool-02f7beb8-9700   Ready    <none>   119m   v1.13.11-gke.14
gke-zulip-cluster-default-pool-02f7beb8-rzcc   Ready    <none>   5d3h   v1.13.11-gke.14
gke-zulip-cluster-default-pool-02f7beb8-srks   Ready    <none>   167m   v1.13.11-gke.14
gke-zulip-cluster-default-pool-02f7beb8-sxv0   Ready    <none>   119m   v1.13.11-gke.14
gke-zulip-cluster-default-pool-02f7beb8-wxsb   Ready    <none>   167m   v1.13.11-gke.14
ricardo@ricardo-ubuntu:~/Downloads$ clear && kubectl delete node gke-zulip-cluster-default-pool-02f7beb8-8q11_
```

Figura 4.4: Comando de listar nodos, e apagar nodos.

```
ricardo@ricardo-ubuntu:~/Downloads$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
quickstart-es-default-0           1/1     Running   0          2m52s
quickstart-kb-c5c8d54d7-npkwn     1/1     Running   0          100m
zulip-1-4ggcx                     5/5     Running   0          16m
zulip-1-9fg5w                     5/5     Running   0          36m
zulip-1-bqjds                     5/5     Running   0          19m
zulip-1-htxkh                     5/5     Running   0          47m
zulip-1-lvrpr                     5/5     Running   0          29m
zulip-1-m29rg                     5/5     Running   0          53m
zulip-1-pnrq5                     5/5     Running   0          53m
zulip-1-tdmfc                     5/5     Running   0          98s
```

Figura 4.5: Listagem dos **pods**, replicas dos componentes.

Capítulo 5

Monitorização

As ferramentas utilizadas para Monitorização da infraestrutura foi: **MetricBeat**, como colecionador e *parser* de dados, **ElasticSearch**, para analisar (*RESTful search and analytics*) e por fim o **Kibana**, para visualização dos dados.
Inicialmente, começou-se por criar uma máquina para monitorização, com os componentes **ElasticSearch** e **Kibana**, mas ao tentar-se instalar os **MetricBeats** nos nós do cluster, não se conseguiu. Isto deveu-se ao facto de se estar a usar o **Kubernetes**, e o mesmo estar preparado para instalar estes componentes de forma automática em containers.

Apesar da instalação do **Elasticsearch** e **Kibana** terem funcionado com **Kubernetes**, novamente a equipa teve problemas com o **MetricBeats**, pelo que devido à falta de tempo para conclusão do projeto, decidiu-se usar as métricas de monitorização fornecidas pelo **Google Cloud**.

Assim, deste modo, verifica-se que uma das necessidade de métricas, em que a monitorização foi útil, foi a deteção de falta de memória **RAM**, tal como se pode ver na figura 5.1 , quando se testou diferentes configurações dos componentes na fase de **benchmark**.

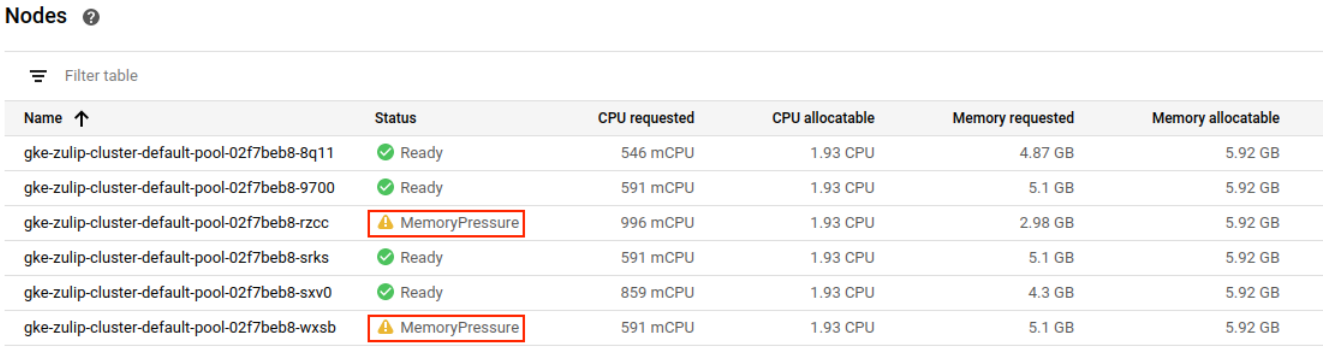

Figura 5.1: Monitorização do Google Cloud.

Capítulo 6

Benchmarking

Após a configuração da arquitetura e deployment do sistema foi efetuado o **benchmarking** simultâneamente à monitorização, desta forma é possível analisar com **dados concretos** se a arquitetura atual é **eficiente** para o uso **realista** da aplicação zulip por parte de vários clientes. Note-se que não é pretendido ter o sistema com os melhores recursos possíveis, isso seria dispendioso, pretende-se ter recursos suficientes para responder aos pedidos dos clientes atendendo ao binómio **custo vs rapidez**.

Foi efetuado 1 **HTTP(S) request**:

1. obtenção página inical: method **GET** https://zulip/login/

O request foi analisaefetuado com 9 configurações que variam entre o **número de clientes** e **número de pedidos de cada cliente** a cada momento. Desta forma os resultados obtidos em relação ao **número de pedidos por minuto** (throughput)que a aplicação zulip consegue atender e o respetivo tempo total encontram-se nas seguintes tabelas.

#Clientes + #Pedidos	Throughput	Tempo Total
250 C + 1 P	65 217	menos de 1 seg
250 C + 5 P	186 567	menos de 1 seg
250 C + 10 P	268 817	menos de 1 seg
500 C + 1 P	68 965	menos de 1 seg
500 C + 5 P	72 011	2 segs
500 C + 10 P	223 665	2 seg
1000 C + 1 P	37 243	1 seg
1000 C + 5 P	73 511	4 segs
1000 C + 10 P	180 018	5 segs

Tabela 6.1: Resultados para o HTTP(S) request 1 (página incial).

A topologia final utilizada para estes teste consiste em **6 nodos** e **7 réplicas**, note-se que cada réplica utiliza 5 serviços/ containers, logo na realidade a arquitetura consiste em **6 nodos** e **35 containers**.

O benchmarking **foi essencial para otimizar a configuração dos recursos** necessária para o funcionamento fluído da aplicação.

Inicialmente a configuração era de **3 nodos** e **1 réplica** apenas, quando foi analisado HTTP(S) request 1 verificou-se o seguinte resultado.

#Clientes + #Pedidos	Throughput	Tempo total
1000 C + 1 P	457	120 segs

Tabela 6.2: Resultados para o HTTP(S) request 1 para **3 nodos** e **1 réplica**.

Este resultado é péssimo para uma aplicação que à partida é utilizada por muitos clientes simultâneamente, uma vez que uma espera de 2 minutos é impensável, logo foi alterada a topologia, decidiu-se analisar o mesmo HTTP(S) request para várias configurações.

#Clientes + #Pedidos	Throughput	Tempo total
1000 C + 1 P	37 428	1 seg

Tabela 6.3: Resultados para o HTTP(S) request 1 para **6 nodos** e **4 réplicas**.

#Clientes + #Pedidos	Throughput	Tempo total
1000 C + 1 P	33 783	1 seg

Tabela 6.4: Resultados para o HTTP(S) request 1 para **6 nodos** e **7 réplicas**.

#Clientes + #Pedidos	Throughput	Tempo total
1000 C + 1 P	71 942	menos de 1 seg

Tabela 6.5: Resultados para o HTTP(S) request 1 para **6 nodos** e **8 réplicas**.

#Clientes + #Pedidos	Throughput	Tempo total
1000 C + 1 P	39 040	1 seg

Tabela 6.6: Resultados para o HTTP(S) request 1 para **6 nodos** e **10 réplicas**.

#Clientes + #Pedidos	Throughput	Tempo total
1000 C + 1 P	574	112 segs

Tabela 6.7: Resultados para o HTTP(S) request 1 para **8 nodos** e **8 réplicas**.

Conclui-se que a melhor configuração seria **6 nodos** e **8 réplicas**, no entanto após algum tempo com a configuração ativa notou-se que por certos períodos de tempo, existia pressão na memória RAM, tornando a aplicação mais lenta e por vezes o servidor parava, desta forma testou-se a topologia com **6 nodos** e **7 réplicas** e após bons resultados ficou a topologia final.

Capítulo 7

Conclusão

Em suma, conclui-se que alguns dos objetivos inicialmente propostos foram atingidos, tais como, análise da arquitetura, análise dos pontos críticos, instalação/deployment da aplicação e benchmark.

No entanto, em relação à monitorização, não se conseguiu instalar as aplicações necessárias para a mesma, mais especificamente, o **MetricBeat** em cada nodo.
Como alternativa, e devido à falta de tempo, a equipa baseou-se na ferramenta de monitorização da **Google Cloud**, mais propriamente **GKE - Google Kubernetes Engine**, que disponibiliza algumas métricas, apesar de que em menos quantidade, foram bastante úteis, tal como já foi dito, no momento de teste de diferentes configuraçõe dos componentes da aplicação, para deteção de falta de **RAM**.

A utilização de kubernetes, facilitou o processo de automatização do **Deployment** da aplicação, bem como das alterações da configuração dos componentes, tornando possíveis instalações para produção mais facilitadas.
A análise de benchmark da aplicação apesar de se conseguir boas conclusões dos resultados, poderia ter sido feita com mais funcionalidades, mas devido à falta de tempo, não foi possível.
Após os resultados obtidos nas várias configurações dos componentes, dos benchmarks, verificou-se que a melhor configuração, a nível performance, consiste seis nodos e sete replicas de cada componente.
Por fim, e não menos importante a análise dos pontos críticos da aplicação passou por uma análise da arquitetura da mesma e identifcar possíveis problemas. Os principais problemas encontrados foram a não tolerância a faltas de determinados componentes, que compromentem todo o sistema.