Universidade do Minho
Mestrado Integrado em Engenharia Informática
Departamento de Informática

System Deployment & Benchmarking

Ghost

Grupo 2
Carlos Pinto Pedrosa - A77320
Eduardo Gil Ribeiro da Rocha - A77048
Manuel Gouveia Carneiro de Sousa - A78869
28 de Dezembro de 2018

Conteúdo

1 Introdução

Nos dias atuais, o mais simples serviço envolve uma série de componentes tecnológicos, equipas de manutenção, gestão, entre outros, tornando-se, por vezes, extremamente difícil gerir toda a infraestrutura que suporta o sistema.

No âmbito da Unidade Curricular de *System Deployment And Benchmarking*, foi-nos proposto, numa primeira fase, o estudo de um sistema e, nesta segunda fase, o seu *deployment*, monitorização e avaliação.

Assim, neste relatório, iremos explicar as decisões que foram tomadas ao longo do desenvolvimento desta fase, e detalhar de que forma o *deployment* do sistema foi feito.

2 Ghost

Ghost é uma plataforma *open source*, criada em 2013, com o intuito de construir e modernizar publicações *online*. A grande missão do *Ghost* é a de criar ferramentas abertas para todo o tipo de jornalistas e escritores, tendo assim um forte impacto no futuro dos *media*.

Assim, neste capítulo iremos apresentar a arquitetura e os componentes do sistema, bem como todo o processo de *deployment*, monitorização e *benchmark*.

2.1 Arquitetura & Componentes da Aplicação

Do ponto de vista arquitetural, o *Ghost* possui 3 grandes ideais, os quais sustentam todo o Sistema:

- ***RESTFul JSON API* no seu núcleo;**
- **Aplicação sobre um cliente (ou web) para o Administrador;**
- **Um *front-end* bem desenhado.**

Todas estes ideais funcionam de forma concreta e concisa, fornecendo ao utilizador uma panóplia de opções de customização.

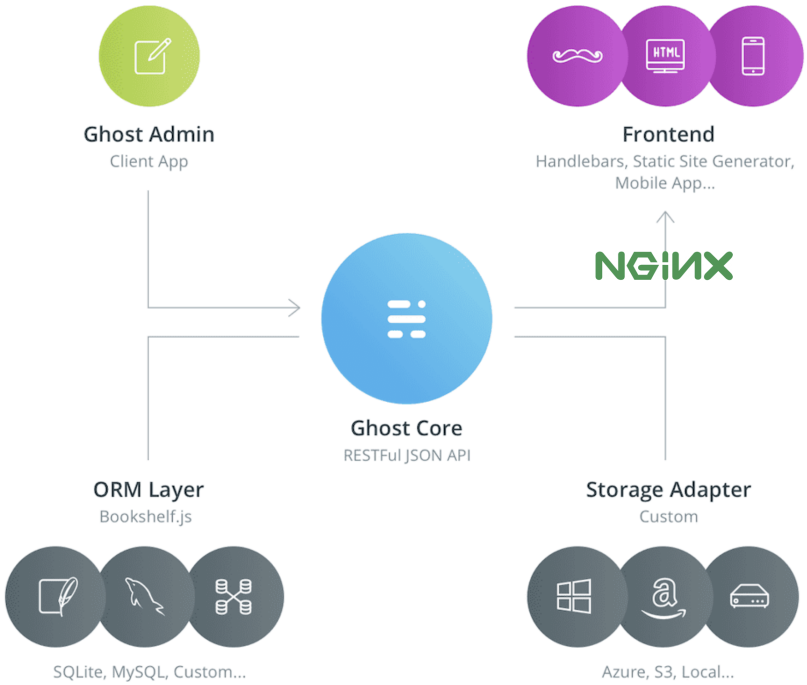

Figura 1: Arquitetura do Sistema Ghost

Assim, quando o administrador pretende, por exemplo, submeter uma nova publicação que já escreveu usando o front-end disponibilizado, esta é transformada em *JSON* e enviada para o servidor web *NGINX*, que é depois interpretado pelo *Ghost Core*. No seguimento e consoante as configurações do *Ghost*, esta é depois guardada na Base de Dados ou Serviço de Armazenamento Externo.

Falaremos agora dos principais componentes presentes no *Ghost*.

2.1.1 Base de Dados & Sistema de Ficheiros

Em termos de Bases de Dados, o *Ghost* vem equipado com a biblioteca *Bookshelf.js ORM* que permite a utilização de inúmeros motores de bases de dados. No entanto, é sugerido *MySQL* no ambiente de produção.

Adicionalmente, apesar do *Ghost* utilizar o sistema de ficheiros local por defeito, este também permite a utilização de serviços que possibilitam colocar o sistema de ficheiros externamente, como na *Cloud*.

2.1.2 Ghost Core

No núcleo desta aplicação, encontra-se uma *API RESTFul*, a qual é usada para interpretar os pedidos enviados pelos utilizadores da plataforma.

De facto, quando um consumidor pretende usufruir de algum serviço disponibilizado pelo *Ghost*, este envia um método *HTTP* (*POST, GET, etc*) acompanhado de um *JSON* que contém toda a informação necessária para satisfazer o pedido de um cliente. Estas informações são recebidas pelo servidor *web* que, por sua vez, os redireciona para o *Ghost Core* para processamento. Este, por sua vez, graças à *API RESTFul*, conseguirá desempacotar os dados úteis e, de acordo com eles, continuará a sua execução.

2.1.3 Web Server NGINX

Devido ao facto da plataforma funcionar em ambiente web, é necessário, pelo menos, um servidor capaz de lidar com os pedidos *HTTP*. Efetivamente, nesta plataforma em particular, é usado o servidor *NGINX*.

Adicionalmente, este servidor efetua o redirecionamento dos pedidos para o núcleo do sistema, onde são interpretados e construída uma resposta.

Posto isto, esta resposta é devolvida ao *NGINX*, que por sua vez a encaminha para o Cliente.

2.2 Deployment do Sistema

No âmbito de automatizar o *deployment* do *Ghost* decidiu-se usar como ferramenta o *Ansible*. Este é um motor que permite automatizar o provisionamento, gestão de configurações, *deployment* de sistemas, orquestração de serviços internos, entre outras tarefas . Usando *YAML* como linguagem, este permite descrever diversas tarefas a serem automatizadas, através de *playbooks*, aproximando-se de uma linguagem natural.

Apesar daquilo que foi previamente modelado, a distribuição final do sistema foi um pouco alterada. Resolvemos então criar um esquema desta nova distribuição, com o intuito de ser comparada com a anterior. Posto isto, iremos discutir os passos dados até o *deployment* final, e explicar de que forma foram resolvidos os desafios que iam surgindo.

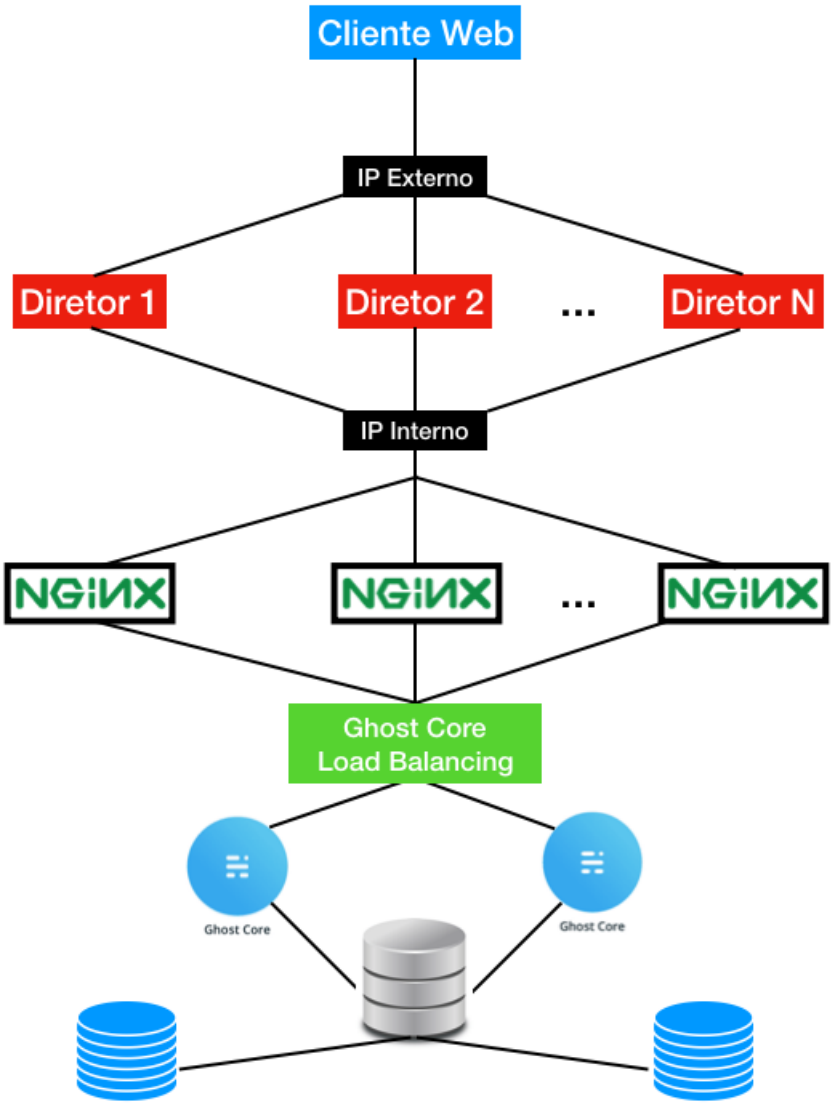

Figura 2: Arquitetura Modelada

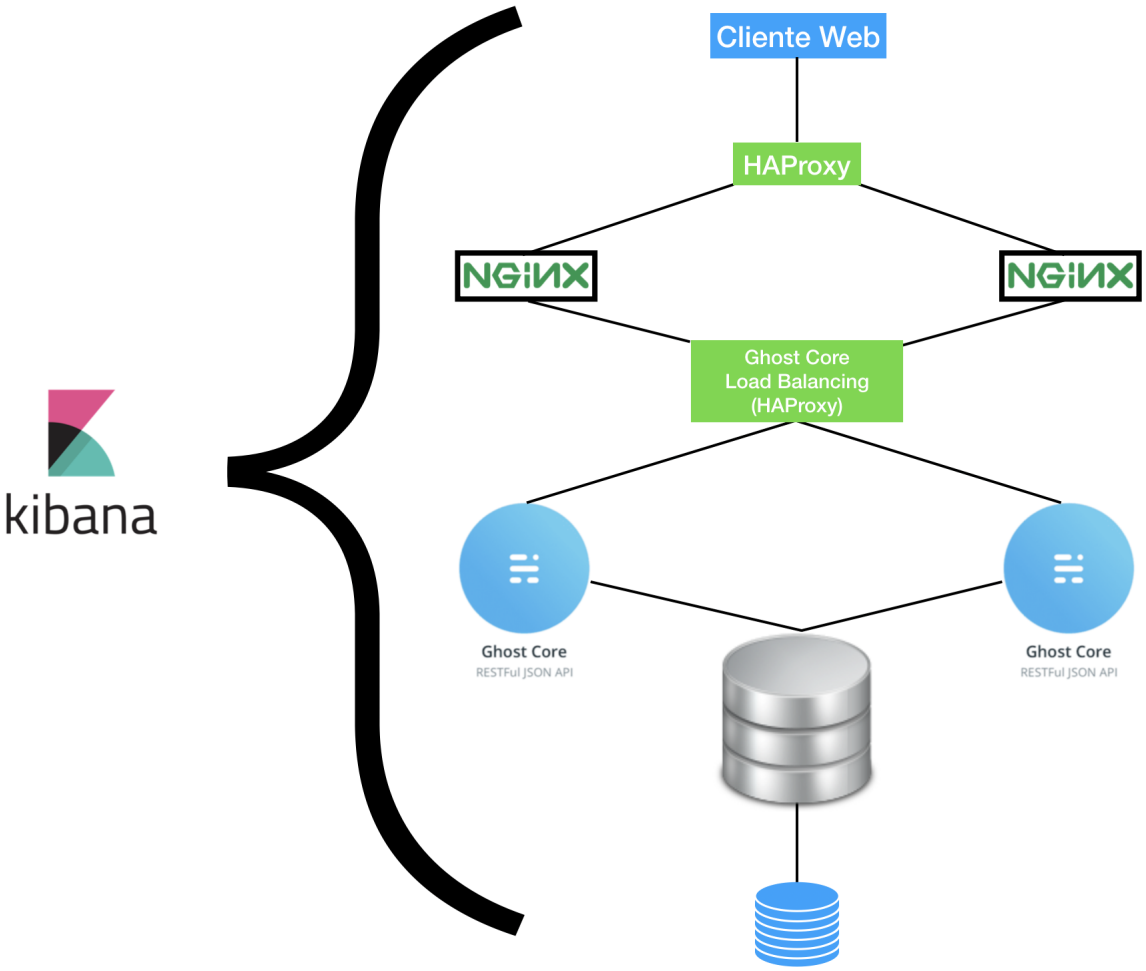

Figura 3: Arquitetura Implementada

Toda este processo foi automatizado através de diversas tarefas que foram detalhadas no *Playbook* desenvolvido. Neste, foram incluídas todas as instruções necessárias para a criação das instâncias de máquinas virtuais. Isto foi feito através de uma funcionalidade do *Ansible*, denominada de *Role*.

2.2.1 Provisionamento do Sistema

O primeiro passo foi o provisionamento do sistema, isto é, a criação das máquinas virtuais no *Google Cloud Platform*, onde mais tarde se iria instalar todos os componentes do sistema.

Inicialmente, com base na Figura 2, tínhamos modelado que iriam ser necessárias 11 máquinas virtuais (excluindo a monitorização) para albergar todo o sistema. No entanto, devido a diversos desafios que foram surgindo, apenas foram utilizadas 7 máquinas virtuais. De notar que foram escolhidas apenas máquinas virtuais com o intuito de promover uma elevada disponibilidade do sistema, uma vez que se uma das máquinas falhar, é apenas afetado um dos componentes. A título exemplificativo, segue-se o excerto pertencente ao *Playbook* que diz respeito à definição de cada uma das máquinas virtuais.

```
- hosts: localhost
  connection: local
  gather_facts: no
  roles:
    - role: createInstances
      vars:
        ...
        instances:
          - { index: 1, number: 1, tag: monitor }
          - { index: 2, number: 1, tag: wslb }
          - { index: 3, number: 1, tag: ws }
          - { index: 4, number: 2, tag: ws }
          - { index: 5, number: 1, tag: ghostlb }
          - { index: 6, number: 1, tag: ghostcore1 }
          - { index: 7, number: 2, tag: ghostcore2 }
          - { index: 8, number: 1, tag: masterdb }
          - { index: 9, number: 2, tag: slavedb }
```

Figura 4: Excerto Playbook

2.2.2 Configuração dos Componentes

Após a fase de provisionamento, são alterados os ficheiros de configuração dos diversos componentes do sistema presente na máquina *Host* para mais tarde copiar para as máquinas *Target*. Nestes, são alterados os *IP's* de cada componente. A título de exemplo, resolvemos incluir um excerto de código usando o comando fornecido pelo *Ansible*, denominado de *Replace*.

```
- name: Replace ElasticSearch Internal IP In MetricBeat Config File
replace:
    path: ".../metricbeat.yml"
    regexp: 'ELASTICSEARCHIP'
    replace: "{{instance.results[0].networkInterfaces[0].networkIP }}"
```

Figura 5: Replace Internal IP ElasticSearch

Posto isto, iremos passar a explicar cada um dos componentes envolventes do sistema.

2.2.2.1 Servidores Web

O **primeiro** componente a ser configurado foram os servidores web, bem como o balanceamento de carga relativamente a estes. De facto, como primeira abordagem, foram escolhidos os serviços *KeepAlive + HAProxy* para o balanceamento de carga. No entanto, como o *KeepAlive* utiliza internamente um IP (necessário para que todos os clientes se liguem a um único IP), que neste caso iria ser um IP público, não foi possível configurar corretamente este serviço. Como alternativa, era possível usar o *Google Cloud Load Balancing*, mas optámos por não usar visto que divergia do principal propósito desta tarefa. Decidimos então utilizar apenas um servidor de balanceamento de carga, que iria ser o ponto de entrada de todos os pedidos *HTTP*. Posto isto, baseado no algoritmo *Round-Robin*, é escolhido um servidor para responder ao pedido de um cliente. Na configuração inicial (*Playbook*), apenas estão contemplados dois *web servers*, no entanto, caso seja necessário escalar este componente do sistema, apenas é necessário incluir uma nova instância de "ws" na criação, bem como adicionar as respetivas entradas no ficheiro de configuração do *Load Balancer*.

2.2.2.2 Ghost Core

O **segundo** componente a ser configurado foram ambos os *Core's* do *Ghost* e o seu próprio balanceador. De novo, numa primeira abordagem, foi decidido usar o *KeepAlive + HAProxy* para este balanceamento. De facto, como referido anteriormente e devido a certos desafios que não foram ultrapassados, resolvemos usar apenas um servidor de balanceamento. Novamente, caso seja necessário escalar este componente do sistema, apenas se

criam as novas instâncias de "ghostcore" e posteriormente altera-se o ficheiro de configuração do balanceador do *Ghost*. É também importante realçar que o *Ghost* armazena dados no sistema de ficheiros local, como por exemplo, imagens que tenham sido carregadas para o blog. Neste projeto, usamos dois *core's*, bem como sistemas de ficheiros locais, e por isso podem existir falhas nos pedidos dos clientes. Uma solução para este problema é a utilização de um sistema de ficheiros partilhado ou remoto.

2.2.2.3 Base de Dados

Como **terceiro** e último componente, foi configurada a base de dados do sistema. De facto, este é dos componentes mais importantes, visto guardar toda a informação do sistema. Deste modo, é crítico que os dados sejam não só guardados na base de dados principal, mas também numa réplica (*Master/Slave Replication*), para desta forma, caso algum imprevisto ocorra, os dados não serem perdidos. Para além da replicação, é também possível incluir *Load Balancing* neste tipo de arquitetura (*Master/Master Replication*). Nesta fase, pretendíamos implementar uma arquitetura apenas com um dois elementos, *Master & Slave*, no entanto, não fomos bem sucedidos devido a um erro não ultrapassado do *MySQL*. Por último, em termos de escalabilidade automática, se a arquitetura adotada for do tipo *Master/Slave*, apenas é necessária adicionar uma nova instância de "slavedb" e adicionar o ficheiro de configuração respetivo. Por outro lado, se a arquitetura adotada for do tipo *Master/Master*, irá ser necessário alguma configuração extra.

2.3 Monitorização do Sistema

De modo a avaliar o sistema, recorremos a ferramentas capazes de apresentar diversos resultados acerca de vários componentes. Estes resultados, baseados em diferentes métricas, seriam posteriormente interpretados e avaliados pelo Administrador do Sistema, de maneira a que seja possível verificar a necessidade de alterar a arquitetura do sistema.

2.3.1 Ferramentas Utilizadas

Para este efeito, as ferramentas utilizadas para monitorização de todo o sistema foram as seguintes:

- **Metricbeat** - Recolhe métricas de vários sistemas. De CPU a memória e até diversos serviços (por exemplo, MySQL ou NGINX), é uma forma eficiente de enviar estatísticas de sistema.
- **ElasticSearch** - Motor de busca e de análise que recolhe e organiza todas as métricas enviadas pelo *Metricbeat*. Aqui, somos capazes de pesquisar e analisar todas as métricas previamente monitorizadas.
- **Kibana** - Permite-nos visualizar toda a informação compilada pelo *Elasticsearch* e navegar pela mesma. Através de *dashboards*, somos capazes de visualizar inúmeros gráficos que dizem respeito a diferentes métricas de cada serviço monitorizado.

Posto isto, resolvemos desenvolver um modelo que explica a arquitetura de monitorização usada neste projeto.

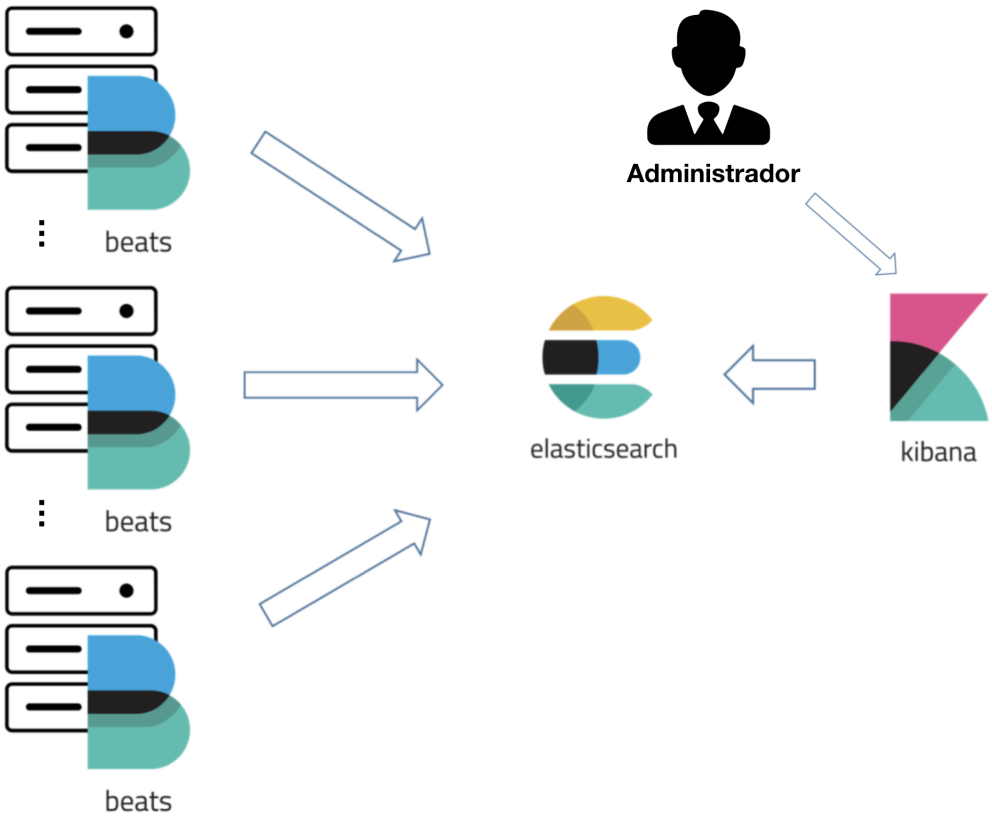

Figura 6: Arquitetura da Monitorização do Sistema

De facto, para instalar e configurar tanto o *Elasticsearch* como o *Kibana* decidimos usar o *Docker*. Esta ferramenta, baseado em *containers*, é bastante poderosa permitindo correr vários *containers* numa única máquina virtual. Como já existia uma imagem *Docker* para ambos os componentes acima mencionados, a sua configuração foi bastante trivial.

2.3.2 Métricas Escolhidas

Por forma a avaliar o sistema com o maior rigor possível, são definidas uma panóplia de métricas para diferentes componentes. Assim, dependendo do ambiente (produção/desenvolvimento), diferentes métricas são usadas.

2.3.2.1 Métricas de Desenvolvimento

Aquando do processo de automação, achamos mais adequado focarmo-nos em métricas de sistema, para assim analisar a carga a que as diferentes máquinas estão sujeitas. Essas métricas, disponibilizadas pelo *Metricbeat*, foram as seguintes:

- System Core
- System CPU
- System DiskIO
- System Filesystem
- System FSSTAT
- System Load
- System Memory
- System Network
- System Process
- System Socket

Através de todas estas métricas, conseguimos tirar diversas conclusões acerca da arquitetura atual do sistema. De facto, avaliando cada máquina internamente, somos capazes de observar se realmente é necessário uma alteração na arquitetura previamente definida. A título de exemplo, se se observar que, mesmo em casos de teste, os nossos *Ghost Core's* necessitam de bastante CPU, talvez seja melhor a adição de mais um componente desse tipo.

2.3.2.2 Métricas de Produção

Em termos de produção, é viável usarmos métricas adicionais, as quais avaliem componentes específicos usados na arquitetura do sistema, como por exemplo os servidores *web* e até mesmo as bases de dados. Neste caso, as métricas por nós adotadas seriam:

- System Module (Core, CPU, Memory, ...)
- HAProxy
- NGINX
- MySQL

Tendo em consideração tanto as métricas de sistema, como as métricas de componentes específicos, conseguimos avaliar concretamente e de forma concisa todo um sistema.

Como prova de conceito, resolvemos incluir o *dashboard* do *Kibana* que traduz todas as métricas monitorizadas previamente.

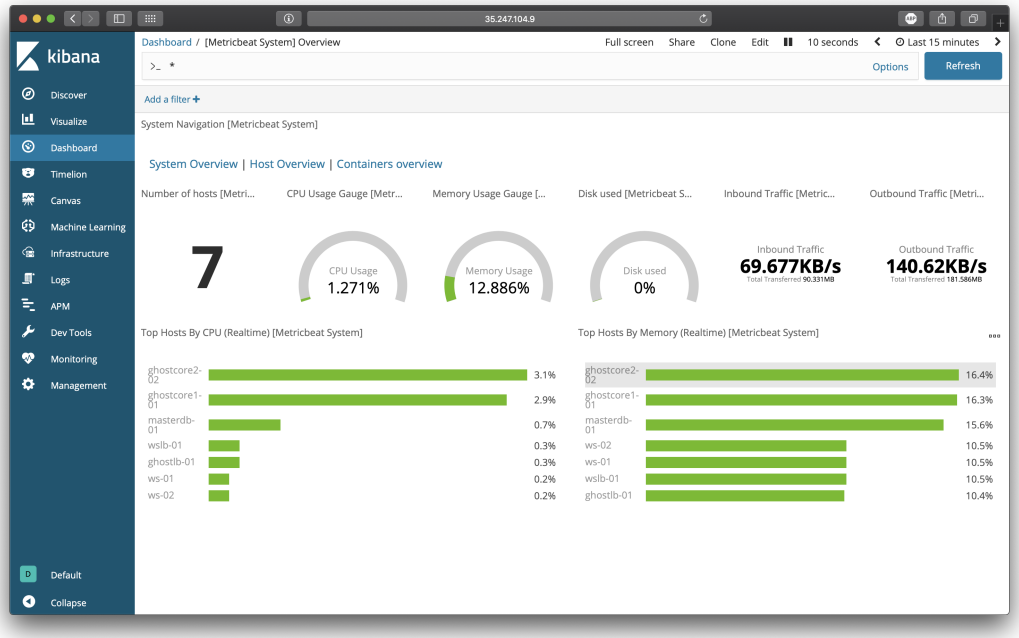

Figura 7: Dashboard do Kibana

2.4 Benchmarking do Sistema

2.4.1 Ferramentas Utilizadas

Com o intuito de avaliar o desempenho da arquitetura final, foram conduzidos vários tipos de testes aos diversos componentes que constituem o sistema. Para isso, recorremos ao auxílio de uma ferramenta denominada de *Apache JMeter*, o qual permite avaliar a *performance* de sistemas, simulando grandes cargas nos servidores e a determinados objetos. Através deste tipo de simulação, é possível avaliar o quão robusto é o nosso sistema quando exposto a diferentes tipos de situações. Como complemento, ainda recorremos à plataforma *RStudio* para analisar os resultados provenientes do *JMeter*.

Visto que a aplicação a ser testada é baseada no conceito de *blog*, fazia todo o sentido fazer testes focados em pedidos *GET* ao servidor. Nos vários casos que vamos apresentar, consideramos múltiplos utilizadores a fazer *GET* a uma página *web* em simultâneo, bem como configurações de *hardware* diferentes.

2.4.2 Testes Efetuados

Uma vez que se está a usar máquinas virtuais na *Google Cloud Platform*, uns dos maiores entraves ao desempenho do sistema são as suas limitações de hardware. Por essa mesma razão, resolvemos efetuar dois tipos de testes. Numa primeira fase, foi executado o *JMeter* para máquinas virtuais apenas com 1 *Core*. Numa segunda fase, optamos por colocar 2 *Cores* em cada máquina virtual.

- **1 *Core*** - *GET* à Página Principal (1000 Clientes)

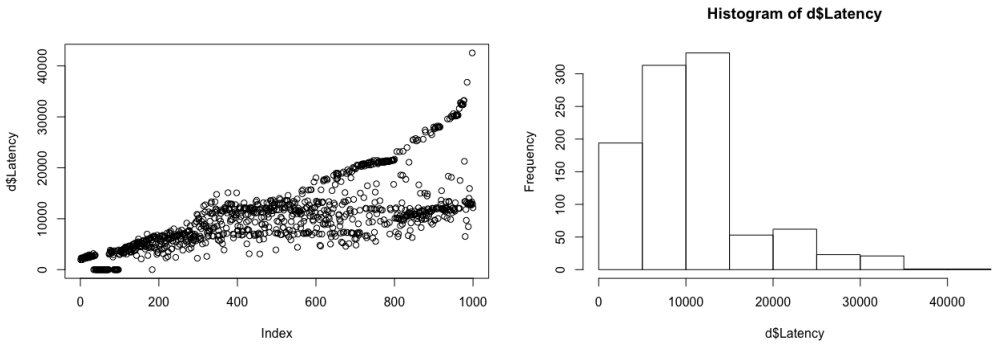

Figura 8: Latência sobre 1 Core e 1000 Clientes

- **1 *Core*** - *GET* à Página Principal (2000 Clientes)

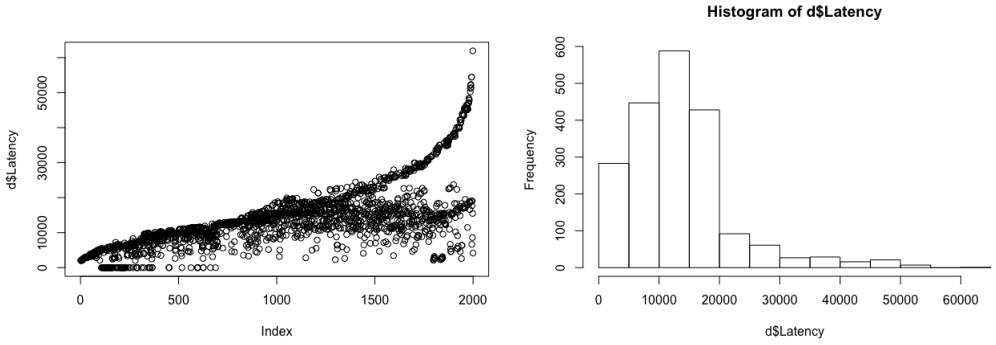

Figura 9: Latência sobre 1 Core e 2000 Clientes

- **1 *Core*** - *GET* à Página Principal (500 Clientes) e *GET* a dois artigos diferentes (250 clientes para cada)

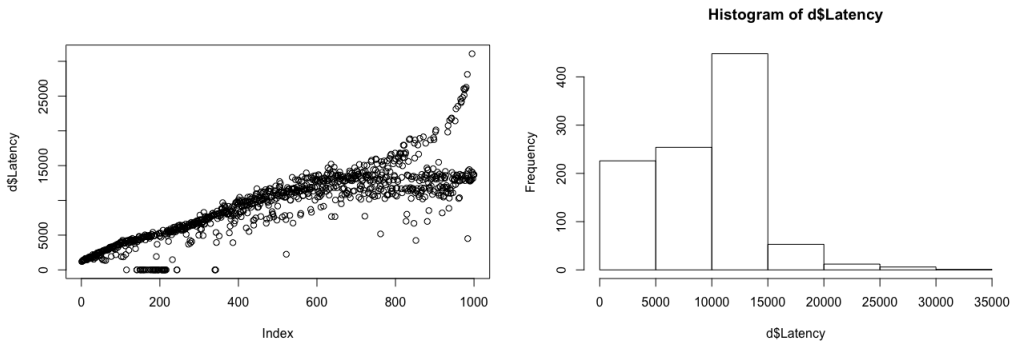

Figura 10: Latência sobre 1 Core e Clientes repartidos

- **2 *Cores*** - *GET* à Página Principal (1000 Clientes)

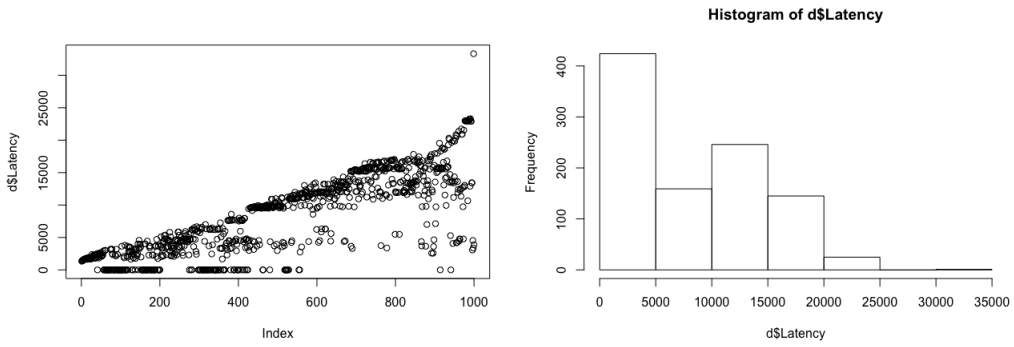

Figura 11: Latência sobre 2 Cores e 1000 Clientes

- **2 *Cores*** - *GET* à Página Principal (2000 Clientes)

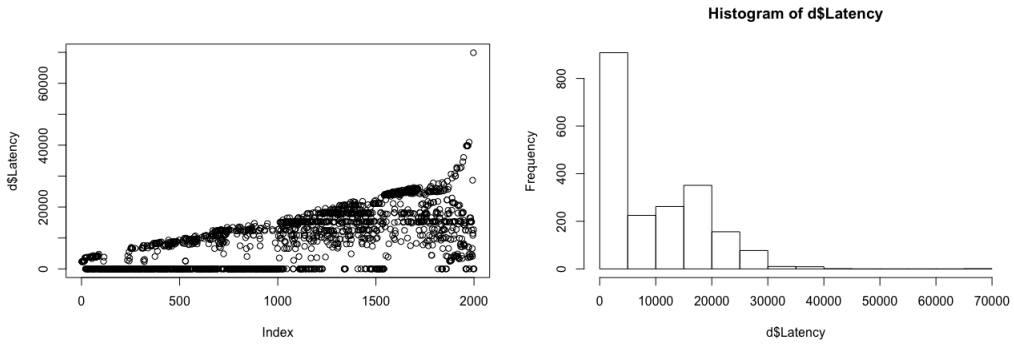

Figura 12: Latência sobre 2 Cores e 2000 Clientes

- **2 *Cores*** - *GET* à Página Principal (500 Clientes) e *GET* a dois artigos diferentes (250 clientes para cada)

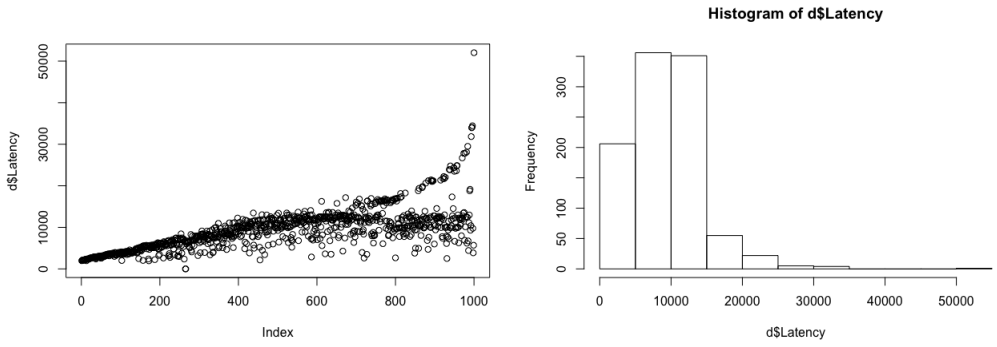

Figura 13: Latência sobre 2 Cores e Clientes repartidos

Depois de analisarmos os resultados obtidos, verificamos que existia uma latência elevada, provavelmente devido à localização das máquinas virtuais (Estados Unidos da América). Posto isto, resolvemos mudar a localização das mesmas para a **Europa**, e dessa forma correr alguns testes novamente.

- **2 *Cores*** - *GET* à Página Principal localizada na Europa (1000 Clientes)

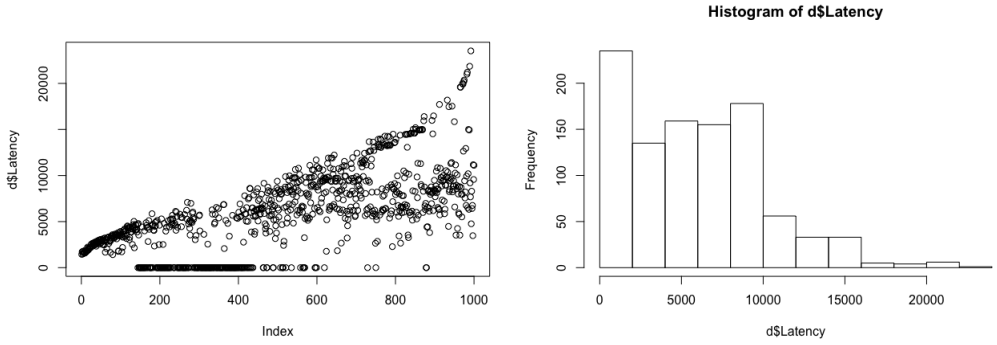

Figura 14: Latência sobre 2 Cores e 1000 Clientes (Europa)

Na realidade, combinando ferramentas de monitorização com ferramentas de avaliação, podemos ter uma visão mais completa do desempenho efetivo do sistema, como, por exemplo, gargalos que este possa possuir.

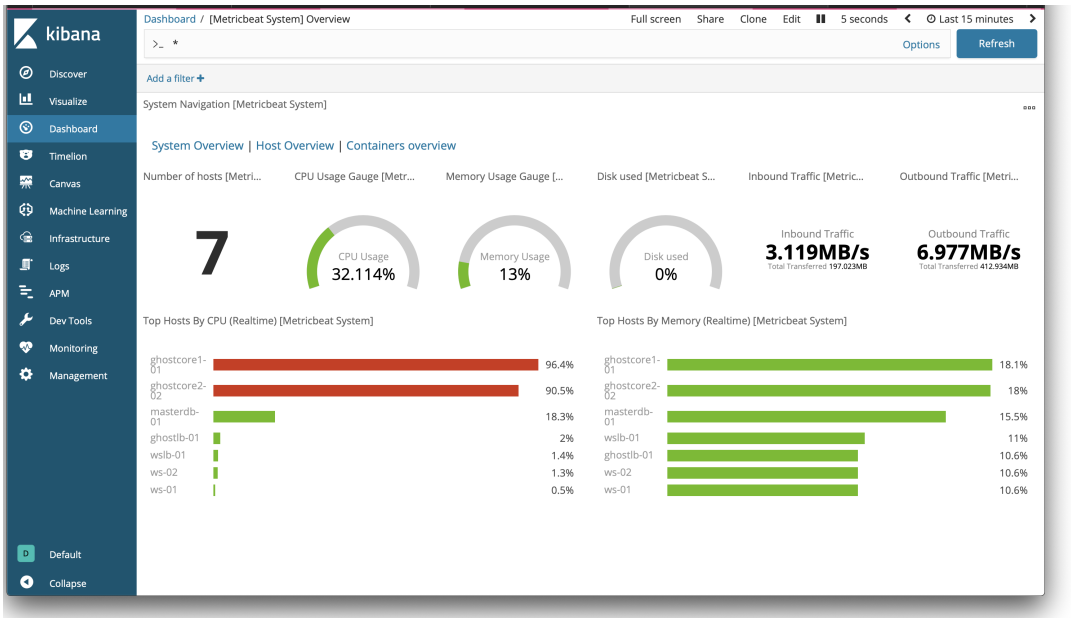

Figura 15: Métricas do Kibana durante o teste com JMeter

2.4.3 Análise de Resultados

Como se pode observar pelos gráficos do subcapítulo anterior, verifica-se que quando se altera o número de clientes de 1000 para 2000, se obtém um aumento significativo de latência. De facto, como verificado pela análise dos resultados obtidos através do *R*, verificamos que com 1000 clientes a aceder à página principal do *Ghost* se obtém uma latência média de 10358 ms, enquanto que para 2000 clientes a latência média aumenta para 13243 ms.

```
> summary(d$Latency)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0    6002    9886   10358   12371   42550
```

Figura 16: Síntese de resultados para 1 Core e 1000 Clientes

```
> summary(d$Latency)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0    8112   12356   13243   16448   61913
```

Figura 17: Síntese de resultados para 1 Core e 2000 Clientes

Quando usados 2 *Cores* nas máquinas virtuais, observa-se que a latência diminui significativamente, visto existir uma maior capacidade de processamento. Na realidade, como mostrado anteriormente, aumentando o número de clientes, existirá também um aumento de latência. No entanto, o aumento não será tão significativo quando comparado com apenas 1 *Core*.

```
> summary(d$Latency)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0    2690    8282    8148   13036   33322
```

Figura 18: Síntese de resultados para 2 Cores e 1000 Clientes

```
> summary(d$Latency)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0       0    8098    8841   15258   69883
```

Figura 19: Síntese de resultados para 2 Cores e 2000 Clientes

Ao repartir os pedidos ao servidor por diferentes páginas, averiguamos que a latência desce substancialmente. Sendo assim, apresenta-se de seguida ambas as sínteses para 1 e 2 Cores.

```
> summary(d$Latency)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0    5550   10238    9301   12838   31093
```

Figura 20: Síntese de resultados para 1 Core e Clientes repartidos

```
> summary(d$Latency)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0    5650    9260    9260   12001   52020
```

Figura 21: Síntese de resultados para 2 Cores e Clientes repartidos

Com o intuito de corroborar a teoria enunciada no subcapítulo anterior, resolvemos também incluir uma síntese dos resultados obtidos com as máquinas virtuais localizadas na Europa. Como esperado, a latência, em média, demonstra-se drasticamente (30%) menor em comparação com os testes anteriores, onde a localização era Estados Unidos da América.

```
> summary(d$Latency)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      0    2358    5722    5766    8619   23526
```

Figura 22: Síntese de resultados para 2 Cores e 1000 Clientes

3 Conclusão

A título de conclusão, é importante referir a importância que a automação de aplicações traz ao *deployment* de sistemas. De facto, através de ferramentas como o *Ansible*, foi possível criar um *script* capaz de, em questão de minutos, conseguir configurar todo um sistema. Como indicado no *website* da plataforma: "Ansible loves the repetitive work your people hate". Isto é um ponto muito importante no que trata de automação.

Em relação ao trabalho que foi realizado, podemos afirmar que este foi uma mais-valia no que toca à aprendizagem de uma nova tecnologia. Efetivamente, com o *Ansible*, fomos capazes de realizar o provisionamento e configuração individual ou em simultâneo de diversas máquinas/componentes. Para complementar a administração do sistema, através do modelo adotado pelo *Ansible*, é também possível escalar e alterar as configurações de um sistema com relativamente pouco esforço. No entanto, não fomos capazes de alcançar todos os objetivos que tínhamos definido, como utilizar mais do que um servidor de balanceamento e assim evitar pontos únicos de falha ou configurar a replicação de dados (*Master/Slave*)

Referências

[1] https://docs.ghost.org/concepts/architecture/

[2] https://docs.ghost.org/concepts/core/

[3] https://docs.ghost.org/concepts/admin/

[4] https://docs.ghost.org/concepts/config/

[5] http://www.haproxy.org

[6] http://www.keepalived.org

[7] https://www.elastic.co/products/elasticsearch

[8] https://www.elastic.co/products/kibana

[9] https://www.elastic.co/products/beats

[10] https://jmeter.apache.org