Universidade do Minho
Escola de Engenharia

Laravel.io

Trabalho Prático
Aplicações e Serviços de Computação em Nuvem
Mestrado em Engenharia Informática

Grupo 39

Gabriela Santos Ferreira da Cunha - pg53829

João António Redondo Martins - pg53905

João Pedro Antunes Gonçalves - pg53932

Miguel de Sousa Braga - pg54095

Nuno Guilherme Cruz Varela - pg54117

dezembro, 2023

Conteúdo

1 Introdução **3**

2 Arquitetura da Aplicação **3**

3 Instalação e Configuração Automática **3**
3.1 *Deployment* da Base de Dados 4
3.2 *Deployment* da Aplicação 5
3.3 *Undeployment* 5

4 Exploração e Otimização da Aplicação **5**
4.1 Monitorização, Métricas e Visualizações 5
4.2 Avaliação e Testes Desenvolvidos 6
4.3 Otimizações Implementadas 8

5 Análise de Resultados **9**

6 Conclusão **12**

1 Introdução

No âmbito da UC de Aplicações e Serviços de Computação em Nuvem, foi-nos proposta a automatização da instalação, configuração, monitorização e avaliação da aplicação `Laravel.io`. Deste modo, este trabalho consiste em 2 tarefas principais:

- **Instalação e configuração automática da aplicação**: utilização da ferramenta *Ansible* para automatizar a instalação e configuração da aplicação `Laravel.io` no serviço *Google Kubernetes Engine (GKE)* da *Google Cloud*;
- **Exploração e otimização da aplicação**: compreensão e otimização do desempenho, escalabilidade e resiliência da aplicação.

2 Arquitetura da Aplicação

A aplicação `Laravel.io` é um portal para resolução de problemas e partilha de conhecimento associado à *framework* de desenvolvimento *web* `Laravel`. Tendo por base os passos de instalação presentes no GitHub, a aplicação divide-se em dois componentes principais:

- servidor aplicacional - desenvolvido em PHP, responsável por processar e responder aos pedidos, consultando a base de dados, se necessário;
- servidor de base de dados - responsável por armazenar a informação persistente da aplicação. Por defeito, é utilizado MySQL como motor de base de dados, embora seja possível utilizar outros.

Contudo, a nossa arquitetura final acabou por sofrer algumas mudanças relativamente à apresentada, uma vez que foram implementadas algumas otimizações, que serão aprofundadas posteriormente neste relatório. Uma das otimizações trata-se da replicação do servidor aplicacional em várias instâncias. Para isto, foi necessário assegurar a consistência do estado entre as diferentes réplicas, garantindo que as diversas réplicas comunicam com um servidor Redis.

3 Instalação e Configuração Automática

Com vista a automatizar os processos de instalação e configuração da aplicação, recorremos às seguintes ferramentas:

- *Docker* - como forma de empacotar a aplicação e base de dados numa imagem que possa ser reaproveitada;
- *Ansible* - como ferramenta de automatização de tarefas (*setup* da aplicação, testes automáticos, etc.);
- *Google Kubernetes Engine (GKE)* - como ambiente de computação em nuvem, onde a aplicação e a base de dados serão instaladas.

Para construir a imagem *docker*, foi necessária a criação do `Dockerfile`, cujo propósito é instalar todos os componentes necessários à aplicação, como dependências PHP, dependências do *node* e o próprio código fonte, importado do repositório. O `Dockerfile` é também responsável por expor a porta do *container* no qual executa a aplicação.

A ferramenta *Ansible* garantiu a instalação dos componentes da aplicação na *Google Cloud (GKE)*. Deste modo, apresentamos os *playbooks* disponíveis relativos à instalação e configuração da aplicação:

- `gke-cluster-create:` cria o *cluster* Kubernetes onde ocorrerá o *deployment* da aplicação;
- `gke-cluster-destroy:` destrói o *cluster* previamente criado;
- `laravelio-deploy:` instala a aplicação no *cluster*; é constituído por 3 *tasks* - *deployment* da base de dados, *deployment* da aplicação e *deployment* do Redis;
- `laravelio-undeploy:` desinstala a aplicação no *cluster*.

Para permitir aos utilizadores proteger informações sensíveis, fez-se uso do *Ansible Vault*. O principal objetivo desta funcionalidade é garantir que dados confidenciais como senhas, chaves privadas e outras informações mais sensíveis não sejam expostos em *plain text*, em *playbooks* ou ficheiros com variáveis. Desta forma, para encriptar ou desencriptar a informação, é utilizada uma senha que deve ser fornecida durante o *runtime* do *playbook*. A *password* definida pela equipa foi `ascngrupo392023`. O *deployment* da aplicação deve ser feito através do seguinte comando:

```
ansible-playbook -i inventory/gcp.yml laravelio-deploy.yml
    --extra-vars "seed_database=true" --ask-vault-pass
```

Listing 1: Execução do *deployment* da aplicação.

3.1 *Deployment* da Base de Dados

A instalação da base de dados no ambiente virtualizado é realizada em 5 tarefas distintas.

1. Criação do *persistent volume*;
2. Aplicação do *persistent volume*;
3. Criação do *persistent volume claim* para o MySQL;
4. Criação do *deployment* para o MySQL;
5. Criação do serviço que expõe a aplicação para o exterior.

Por fim, foi adicionada uma tarefa extra para garantir que o *pod* do MySQL se encontra disponível quando se tenta instalar a aplicação e realizar o *seed* e as *migrations* da base de dados.

3.2 *Deployment* da Aplicação

A instalação da aplicação assenta na criação do *deployment* do serviço `laravelio`. Para além disso, integra também novas tarefas que dizem respeito à espera pelo *pod* e pela atribuição de um *ip* pelo *load balancer*. Por fim, foi alterado o `app_ip` e a porta no ficheiro `gcp.yml`, de forma a poderem ser corridos os testes de acesso à aplicação.

3.3 *Undeployment*

Para fazer o *undeployment* da aplicação, é apenas necessário que o estado de cada *role* seja alterado para *absent*, ao invés de *present*. Isso é feito tanto para o servidor aplicacional como para o Redis e a base de dados.

4 Exploração e Otimização da Aplicação

4.1 Monitorização, Métricas e Visualizações

De modo a entender quais os gargalos no desempenho da aplicação para a posterior otimização, recorremos à ferramenta de monitorização da *Google Cloud*, verificando o uso de recursos (CPU, RAM, I/O) pelos diferentes componentes da aplicação (servidor aplicacional e base de dados). A ferramenta também nos permitiu visualizar, através de gráficos temporais, a variação dessas métricas ao longo do tempo.

De maneira a fazer a monitorização e visualização das métricas, criamos três novos *playbooks*:

- `monitoring-deploy.yml:` cria as *dashboards* para visualização;
- `jmeter-benchmarking.yml:` automatiza a execução dos testes por parte do *JMeter*;
- `jmeter-install.yml:` faz a instalação correta do *JMeter* de forma automatizada.

Para a criação das *dashboards*, utilizamos um *template* do *Google Cloud*, sendo que alteramos alguns aspetos para atender às nossas necessidades.

4.2 Avaliação e Testes Desenvolvidos

De modo a testar o sistema face ao crescente número de pedidos, recorremos à ferramenta *JMeter* para simular o envio de pedidos HTTP de múltiplos clientes, a uma taxa predefinida. Esta ferramenta também nos permitiu saber o *delay* de cada pedido, assim como o código de resposta (saber se o pedido foi ou não resolvido).

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

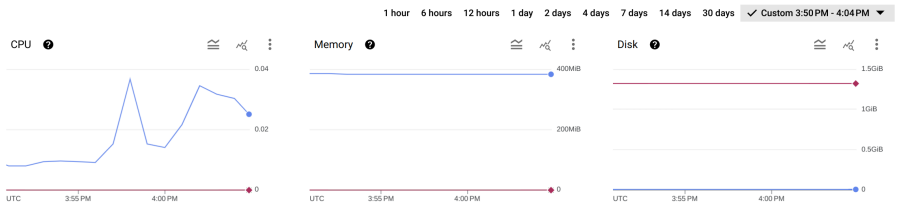

Figura 1: Métricas relativamente ao *deployment* do MySQL.

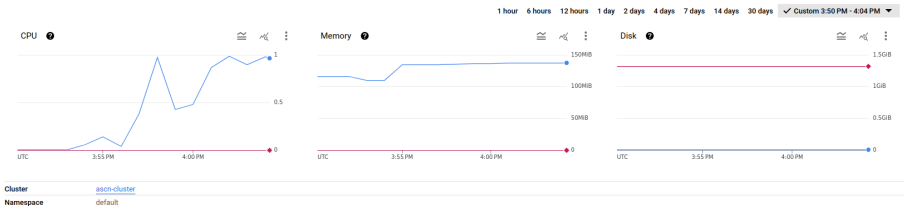

Figura 2: Métricas relativamente ao *deployment* da aplicação.

De acordo com os gráficos representados nas figuras 1 e 2, podemos verificar que o CPU e a memória são duas das métricas que mais impactam no desempenho da aplicação. Posto isto, um dos maiores gargalos de desempenho é o servidor aplicacional, visto que é um dos serviços onde a utilização de CPU é mais elevada.

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

O desempenho da aplicação, quando não tem qualquer tipo de estratégia de escalonamento implementada, varia conforme o número de clientes e cargas de trabalho. A diferença de desempenho na aplicação pode ser vista considerando as seguintes situações:

- **Carga de Trabalho Baixa** - Com poucos clientes, a aplicação funciona eficientemente, com bons tempos de resposta aos pedidos feitos e com uma boa utilização dos recursos, sem sobrecarga;
- **Carga de Trabalho Média** - Com o aumento do número de clientes, os tempos de resposta aumentam e os recursos ficam mais saturados, mas a aplicação continua a funcionar apesar de alguns pedidos, por vezes, não serem respondidos;
- **Carga de Trabalho Alta** - Com um grande número de clientes, a aplicação fica saturada e já não consegue responder à maior parte dos pedidos, visto que, existem erros, muitas vezes de *timeout*. No nosso caso, na maior parte das vezes que a aplicação estava sujeita a este tipo de cargas de trabalho o serviço ficava indisponível.

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

Os principais componentes da aplicação qure podem constituir um ponto único de falha são:

- **Servidor de Base de Dados (MySQL)** : Se houver uma falha no servidor de base de dados, esta falha pode tornar toda a aplicação inacessivel;
- **Servidor Web/PHP (Laravel)** : Tal como o servidor da BD, se houver uma falha no servidor aplicacional, esta pode levar a que a aplicação fique inacessível;

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

As otimizações de distribuição/replicação de carga que podem ser aplicadas à instalação base são:

- **HPA (*Horizontal Pod Autoscaler*)**: o HPA permite ajustar automaticamente a quantidade de réplicas de um *pod* num *deployment* ou *replica-set* com base na observação de métricas como o uso de CPU ou memória. Essencialmente, este ajuda a garantir que os *pods* sejam escalados horizontalmente, ou seja, aumentando ou diminuindo o número de réplicas para lidar com variações na carga de trabalho. As principais vantagens deste mecanismo são a monitorização de métricas, *auto-scaling* e flexibilidade e eficiência;
- ***Autoscaling* do *cluster***: refere-se ao ajuste automático do número de nodos virtuais num *cluster* com base na necessidade de carga de trabalho. Difere do *Horizontal Pod Autoscaler* (HPA), na medida em que o

segundo ajusta o número de réplicas de um *pod* dentro do *cluster*. As principais vantagens desta otimização são a adaptação dinâmica (permite que o sistema aumente ou diminue automaticamente o numero de nodos do *cluster*), eficiência de recursos e balanceamento de carga.

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

Tanto o HPA como o *autoscaling* do *cluster*, permitem que a aplicação escale horizontalmente, aumentando ou diminuíndo o número de *pods*, no caso do HPA, ou o número de *VM instances*, no caso do *autoscaling*, de acordo com os pedidos. Assim, estas 2 otimizações asseguram que os recursos são alocados de maneira eficiente. Durante picos de tráfego, a aplicação pode alocar mais recursos para manter um alto nível de desempenho e, durante períodos de baixa demanda, reduzir os recursos.

Em relação à resiliência, caso haja um aumento súbito de tráfego que ameaçe sobrecarregar a aplicação, são rapidamente adicionados mais nodos ou *pods* para lidar com a carga, o que ajuda a manter a aplicação estável e disponível. Em situações em que os *pods* falham ou se tornam não responsivos, o HPA pode iniciar automaticamente novos *pods* para substituir os que estão com problemas. Da mesma forma, em caso de falhas em um nodo, o sistema pode automaticamente provisionar novos nodos para substituir os que falharam, contribuindo para uma rápida recuperação de falhas.

4.3 Otimizações Implementadas

Neste sentido, iremos aprofundar a implementação do HPA. A outra otimização referida anteriormente, o *autoscaling* do *cluster*, não foi implementada, uma vez que não foi aconselhado alterar o ficheiro de criação do *cluster*.

Assim como foi abordado na secção 2, foi necessária a criação de um servidor Redis para a implementação do HPA. No contexto da nossa aplicação, o Redis é utilizado para fazer a gestão dos dados de sessão das várias réplicas do serviço, ou seja, garante a coerência entre as sessões para que, caso a aplicação guarde algum estado, por exemplo dados de *login* ou *cache*, este seja coerente entre as diversas réplicas. Para isto, foi criado um novo *role*, por forma a guardar as *tasks* relativas à utilização deste sistema de armazenamento. Estas *tasks*, por sua vez, são invocadas no *deployment* do `Laravel.io`.

Com vista a realizar o *auto-scaling* dos *pods*, foi acrescentada uma nova *task* ao *deployment* da aplicação. Para a implementação do *auto-scaling* do *cluster* foram atualizadas as *tasks* relativas à criação do *cluster*. Assim sendo, o número mínimo de *pods* foi alterado para 2 e o número máximo para 6.

5 Análise de Resultados

De forma a avaliar o comportamento da aplicação, elaboramos uma série de gráficos com base nos resultados obtidos pelo *JMeter*. Primeiramente elaboramos gráficos que revelam o tempo de resposta em função do tempo para cada teste. Depois, apresentamos um gráfico de escalabilidade, com os tempos mínimo, máximo, médio e mediano de cada teste. A mediana revelou ser uma métrica mais útil que a média uma vez que não é tão influenciada pelos *outliers*. Inicialmente, avaliamos o comportamento base do sistema, com base nos tempos de resposta dos pedidos `GET /login` efetuados.

Tal como podemos ver nas imagens abaixo, a aplicação consegue suportar os pedidos de 1 e 10 clientes a cada 300ms. No entanto, a aplicação não escala para 100 clientes, tal como esperado, verificando-se *timeouts* (resultados com latência 0) para pedidos a partir dos 20s.

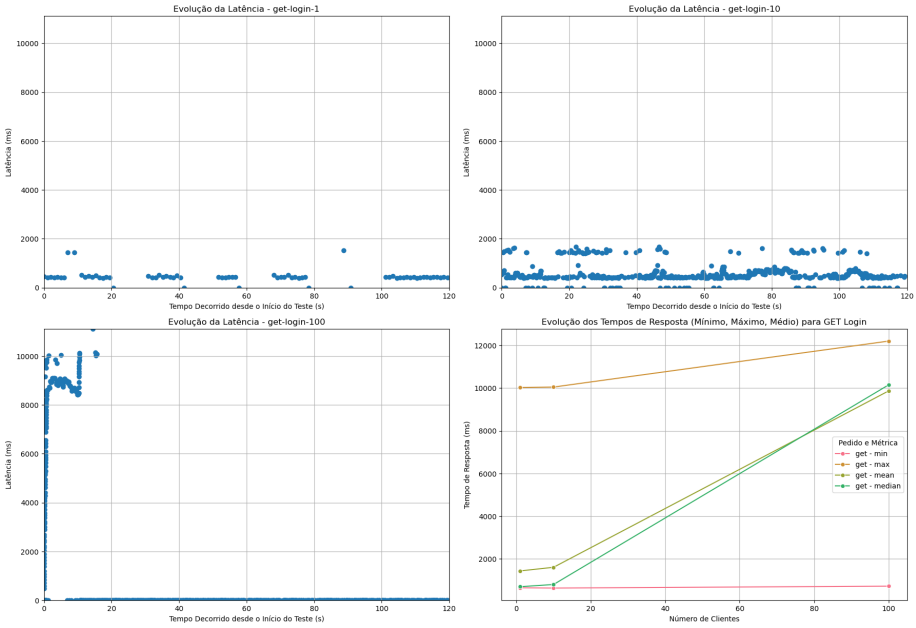

Figura 3: Resultados da versão básica.

Em contraste, a versão com *autoscaling* apresenta um comportamento bem mais interessante. Não são verificados *timeouts* para o cenário de 10 clientes e, apesar de na versão com 100 clientes ainda aparecerem *timeouts*, o comportamento é bastante melhor do que o inicial. A existência de *timeouts* indica a necessidade de aumentar o número máximo de réplicas do *deployment laravelio* (atualmente igual a 6).

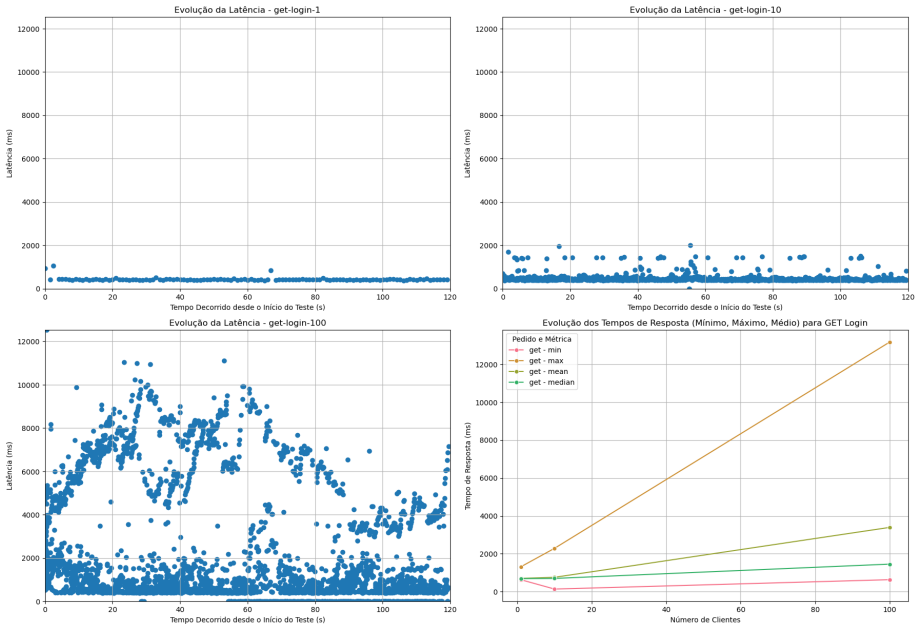

Figura 4: Resultados da versão otimizada.

Em relação à utilização do CPU de cada uma das réplicas, podemos verificar que a task de *autoscaling* está a fazer o pretendido. Até às 2:57, sensivelmente, apenas 2 réplicas do *deployment laravelio* se encontravam em execução. Após essa hora, novas réplicas desse *deployment* foram adicionadas, devido ao crescimento do número de pedidos e o tempo de CPU foi dividido pelas mesmas.

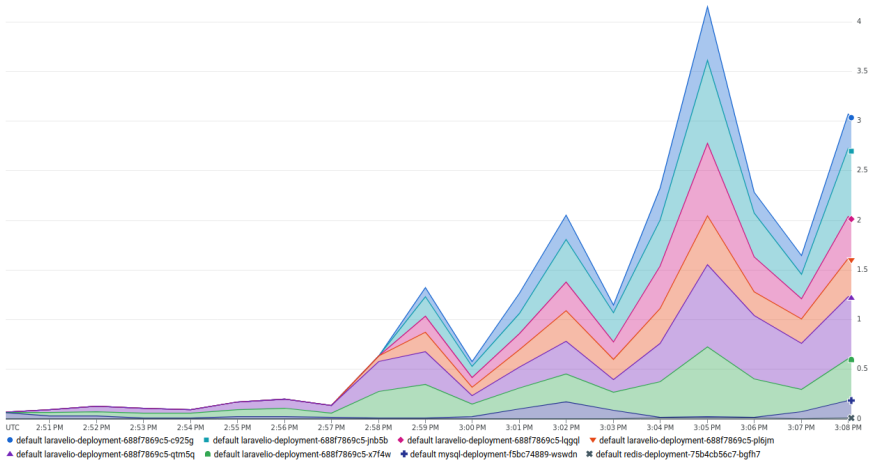

Figura 5: Utilização de CPU.

Por fim, uma análise do número de pacotes enviados pela aplicação por segundo permite-nos chegar à conclusão que, durante o teste, foi atingida a capacidade máxima da nossa aplicação, tendo em conta o número de nodos disponíveis, a capacidade das máquinas e o número máximo de réplicas. Isto fica visível pelo facto da execução do teste de carga com 100 clientes apresentar um pico inferior ao do teste com 10 clientes, o que pode sugerir que, no teste com 100 clientes, a aplicação esteja em sobrecarga.

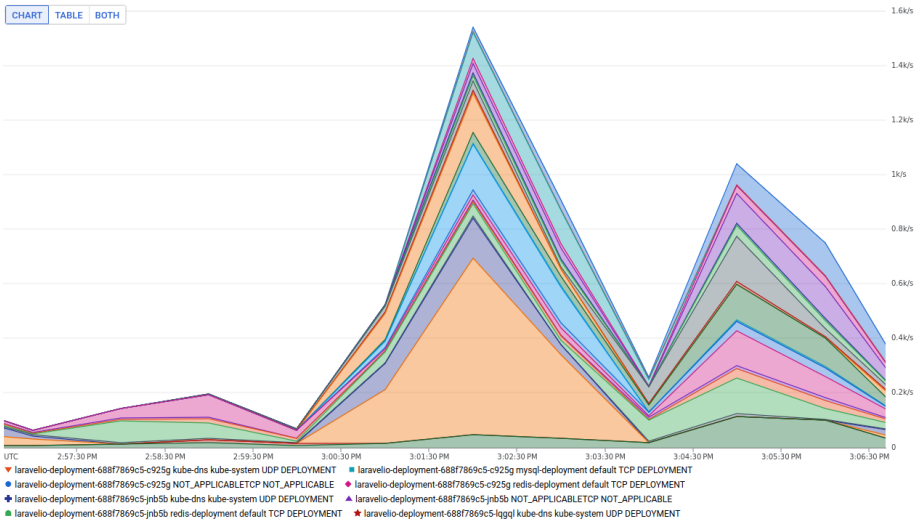

Figura 6: Pacotes enviados por segundo.

6 Conclusão

Neste trabalho, foi automatizada a instalação e configuração da aplicação `Laravel.io` no serviço *Google Kubernetes Engine (GKE)* da *Google Cloud* e foram exploradas otimizações com vista a aumentar o desempenho e resiliência da mesma.

Refletindo sobre toda a fase de desenvolvimento e momentos de avaliação, não encontramos dificuldades durante o primeiro *checkpoint*, que consistia na criação da imagem *docker*. Já em relação ao *checkpoint* seguinte, o desempenho da equipa não foi o pretendido, onde acabamos por não cumprir com os objetivos que tínhamos estabelecido. Contudo, a equipa detetou e corrigiu rapidamente os erros, culminando no trabalho entregue nesta presente fase, do qual estamos bastante satisfeitos. Para além de termos implementado com sucesso as funcionalidades essenciais, implementamos alguns extras como o *mailtrap*, utilizado para reencaminhar *emails* da aplicação para o utilizador, e o registo através do GitHub.

Em relação ao trabalho, poderiam ter sido realizados outros tipos de testes mais realistas, com base em distribuições não uniformes de pedidos. Contudo, por motivos de tempo, não foi possível testar e avaliar o sistema perante esses novos cenários. Para concluir, realçamos algumas otimizações que ainda poderiam vir a ser feitas. Uma delas é a otimização do desempenho e disponibilidade da base de dados. Numa primeira abordagem, os dados poderiam ter sido particionados entre instâncias diferentes. Esta estratégia contribui para a aceleração das operações de leitura e escrita, uma vez que cada partição tem um volume de dados menor para gerir e pode ser otimizada e escalada independentemente. Outra alternativa seria manter uma cópia primária da base de dados para operações de escrita, enquanto que outras várias réplicas seriam utilizadas para operações de leitura. Tendo em conta que as operações de leitura são as mais comuns e seriam distribuídas por várias réplicas, a carga na base de dados primária seria imensamente reduzida e as respostas seriam mais rápidas. Para além disso, em caso de falha da instância primária, uma das réplicas poderia ser promovida a primária, garantindo a continuidade do serviço. Por último, e com um grau de dificuldade de implementação e posterior de gestão muito mais alto, existem estruturas mais complexas com várias bases de dados primárias, como sistemas *multi-master*, que permitem operações de escrita em múltiplos locais. Esta solução seria particularmente útil em sistemas com várias operações de escrita, onde é essencial que a latência desta operação seja minimizada.