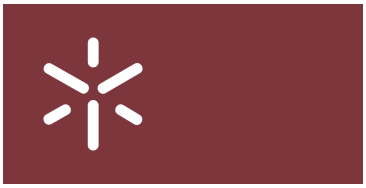

Universidade do Minho
Escola de Engenharia
Mestrado em Engenharia Informática

Unidade Curricular de Aplicação e Serviços de Computação em Nuvem

Ano Letivo de 2024/2025

GrupoTP-23

Pedro Miguel Costa Azevedo
PG57897

Francisco Manuel Afonso
PG57873

João Miguel Mendes Moura
A100615

Rúben Gonçalo Araújo da Silva
PG57900

Daniel Cracel Rodrigues
PG57871

Janeiro, 2025

Data da Receção	
Responsável	
Avaliação	
Observações	

GrupoTP-23

Janeiro, 2025

Índice

Lista de Figuras

Lista de Tabelas

1. Introdução

Este trabalho prático, desenvolvido no âmbito da unidade curricular de **Aplicações e Serviços de Computação em Nuvem**, teve como objetivo principal a instalação e configuração da aplicação **Moonshot**, utilizando os serviços disponibilizados pela **Google Cloud**. A execução do projeto foi realizada de forma totalmente **automatizada** com o auxílio da ferramenta **Ansible**, que garantiu a eficiência no provisionamento e deployment da aplicação.

Para o gerenciamento dos containers que compõem o sistema, foi empregada a tecnologia **Kubernetes**, disponibilizada através do Google Kubernetes Engine (**GKE**). Essa solução permitiu implementar estratégias de **replicação** e **escalabilidade automática**, essenciais para otimizar o desempenho e assegurar a robustez da aplicação.

Ao longo deste relatório, serão apresentadas as decisões técnicas tomadas durante o processo de instalação e configuração da aplicação Moonshot, abordando os mecanismos de escalabilidade horizontal. Serão também detalhadas as ferramentas utilizadas, como o **Apache JMeter**, para testes de desempenho, e os serviços de **monitorização** da Google Cloud Platform, que permitiram uma análise contínua do comportamento da aplicação sob diferentes cargas.

2. Instalação e Configuração Automática

O processo de configuração da aplicação Moonshot foi totalmente automatizado com o uso do **Ansible**, que gerencia eficientemente o ambiente **Kubernetes** através dos módulos k8s e k8s_info. As principais tarefas automatizadas incluem:

Provisionamento do cluster GKE: Configurado para incluir dois nós iniciais, utilizando as variáveis definidas no arquivo gcp.yml.

Configuração de componentes: Deploy da aplicação e serviço Moonshot, além da configuração do PostgreSQL como banco de dados, utilizando YAMLs de deploy.

Os comandos para executar e encerrar o deploy foram:

```
ansible-playbook -i inventory/gcp.yml moonshot-deploy.yml -e
'ansible_python_interprete r=/home/vagrant/.chekpoints/bin/pyhton3'
```

para iniciar a aplicação e

```
ansible-playbook -i inventory/gcp.yml moonshot-undeploy.yml -e
'ansible_python_interpreter=/home/vagrant/.chekpoints/bin/pyhton3'
```

para encerrar a aplicação.

Ao iniciar ou encerrar a aplicação, podemos utilizar flags específicas para controlar o comportamento da base de dados:

Inicialização da Aplicação:

`Flag: -e seed_database`

- true: Realiza o povoamento automático da base de dados
- false: Inicia a aplicação sem povoar a base de dados

Encerramento da Aplicação:

`Flag: -e delete_data`

- true: Remove o PersistentVolumeClaim (PVC) criado
- false: Mantém o PVC e os seus dados

Estas flags proporcionam **maior flexibilidade** no gerenciamento do ciclo de vida dos dados da aplicação, permitindo controlar tanto a **inicialização** quanto a **limpeza** do ambiente de forma automatizada.

3. Arquitetura Inicial da Aplicação

A aplicação foi configurada inicialmente com:

- Uma instância de servidor aplicacional Moonshot.
- Uma instância da base de dados PostgreSQL.

Esses componentes foram hospedados no **cluster Kubernetes** gerido pelo **GKE**. A aplicação foi exposta através de um **LoadBalancer**, permitindo acesso externo.

3.1. Escalabilidade Inicial

A configuração inicial apresentou **limitações significativas** de desempenho:

- Com o **aumento** do número de clientes, o **tempo de resposta aumentou** drasticamente devido à limitação de recursos computacionais.
- Em caso de falhas nos componentes, toda a aplicação era comprometida.

Testes iniciais demonstraram que o sistema não suportava eficientemente diversos clientes simultâneos, a realizar pedidos a cada 3 segundos.

Num. users	50	500	1000
Resp. time mean (s)	0.811	37.764	66.349
Min. time (s)	0.749	1.104	1.105
Max. time (s)	2.879	351.434	476.646
Error %	0.000	3.095	7.488
Throughput (req/s)	13.030	8.707	12.408

Tabela 1: Desempenho da aplicação com diferentes users (sem escalabilidade)

Com **50 users** simultâneos, o sistema apresentou um desempenho relativamente estável, com tempo médio de resposta de 0.811 segundos e sem erros registrados.

A situação piorou significativamente com **500 users** simultâneos, onde o tempo médio de resposta aumentou para 37.764 segundos. Além disso, o sistema começou a apresentar falhas, com uma taxa de erro de 3.095%.

No cenário mais intenso, com **1000 users** simultâneos, o sistema demonstrou clara sobrecarga, com tempo médio de resposta de 66.349 segundos e uma taxa de erro elevada de 7.488%.

Uma estratégia eficaz para abordar estes problemas de escalabilidade é implementar a replicação dos componentes da aplicação através do **HPA**.

3.2. Escalabilidade Final

Com as melhorias implementadas, a aplicação demonstrou diferentes níveis de desempenho e resiliência:

HorizontalPodAutoscaler (HPA): Monitorizou métricas de CPU para ajustar dinamicamente o número de pods em resposta à carga, demonstrando eficácia especialmente em cargas moderadas.

Balanceamento de carga: O uso de um LoadBalancer procurou garantir a distribuição uniforme dos pedidos entre as réplicas, embora os tempos de resposta ainda tenham sido impactados em cargas elevadas.

Persistência de dados: O uso de PersistentVolumeClaims (PVC) assegurou a continuidade dos dados mesmo com a recriação de pods.

Os testes demonstraram a capacidade de escalabilidade do sistema com diferentes cargas de usuários:

Num. users	50	500	1000
Resp. time mean (s)	0.742	9.507	15.325
Min. time (s)	0.705	0.447	1.982
Max. time (s)	1.756	45.683	68.435
Error %	0.000	10.687	25.215
Throughput (req/s)	13.344	18.236	9.319

Tabela 2: Desempenho da aplicação com diferentes users (com escalabilidade)

Com as melhorias implementadas, a aplicação demonstrou diferentes níveis de eficiência dependendo da carga de usuários simultâneos:

Para **50 users**, o sistema apresentou um desempenho estável e eficiente e 0% de taxa de erro. O throughput de 13.344 requisições por segundo indica uma operação consistente e confiável nesta carga.

No cenário de **500 users**, observou-se um aumento significativo no tempo médio de resposta para 9.507 segundos, acompanhado de uma taxa de erro de 10.687%. Apesar dos desafios, o sistema conseguiu aumentar o seu throughput para 18.236 requisições por segundo, demonstrando a eficácia parcial das estratégias de escalabilidade.

Com **1000 users** simultâneos, o sistema apresentou sinais claros de sobrecarga, com tempo médio de resposta de 79.105 segundos e uma taxa de erro elevada de 25.215%. O throughput reduziu para 9.319 requisições por segundo, indicando que o sistema atingiu os seus limites mesmo com as estratégias de escalabilidade implementadas.

O uso do **HorizontalPodAutoscaler** (HPA) mostrou-se mais efetivo em cargas moderadas, mas os dados indicam a necessidade de otimizações adicionais para cargas mais intensas. A análise do throughput e das taxas de erro sugere que o ponto ótimo de operação do sistema está abaixo dos 500 usuários simultâneos, onde ainda mantém um equilíbrio aceitável entre desempenho e confiabilidade.

4. Perguntas Tarefa 2

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

Os principais componentes que se podem tornar gargalos de desempenho são:

Servidor aplicacional Moonshot: O consumo de CPU aumenta significativamente com o número de clientes, especialmente devido ao tempo de latência no escalonamento do HPA.

Base de dados PostgreSQL: Embora tenha apresentado estabilidade nos testes iniciais, uma carga muito alta poderia saturar as conexões ou causar lentidão devido a limitações de I/O.

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

Com 50 clientes: O sistema apresenta estabilidade tanto com HPA (0.742s) quanto sem (0.811s), com 0% de taxa de erro em ambos os casos.

Com 500 clientes: Sem HPA, o sistema degrada significativamente (37.764s de tempo médio), mas com HPA mantém um desempenho mais controlado (9.507s).

Com 1000 clientes: O sistema demonstra clara sobrecarga sem HPA (66.349s), mas com HPA consegue manter tempos de resposta mais gerenciáveis (15.325s), embora ainda com uma taxa de erro significativa.

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

Base de dados PostgreSQL: Não há réplicas configuradas para a base de dados. Em caso de falha, todo o sistema ficaria indisponível, pois a persistência de dados depende unicamente de uma instância.

Load Balancer: Caso o Load Balancer não distribua adequadamente as requisições, um único pod pode ser sobrecarregado, prejudicando o desempenho.

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

Replicação da base de dados: Implementar PostgreSQL com StatefulSets para criar réplicas e garantir maior disponibilidade e redundância dos dados.

Melhoramento do HPA: Reduzir o período de latência no escalonamento, ajustando os limites de CPU/memória para inicializar novos pods mais rapidamente. (Solução implementada pelo grupo).

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

Replicação da base de dados: Aumenta a resiliência da aplicação, reduzindo o risco de falhas e garantindo a continuidade do serviço em caso de problemas na instância principal da base de dados.

Melhorias no HPA: Reduzem o tempo de resposta durante períodos de alta carga, mitigando gargalos e garantindo uma melhor experiência do usuário.

5. Monitorização e Análise de Desempenho

5.1. Monitorização

Durante os testes realizados, foram analisados diversos aspectos do desempenho da aplicação Moonshot, incluindo o uso de CPU, memória e capacidade de escalabilidade. Os gráficos apresentados detalham como a aplicação respondeu em termos de utilização de recursos e comportamento do autoscaling, comparando cenários com e sem HPA para 500 clientes simultâneos.

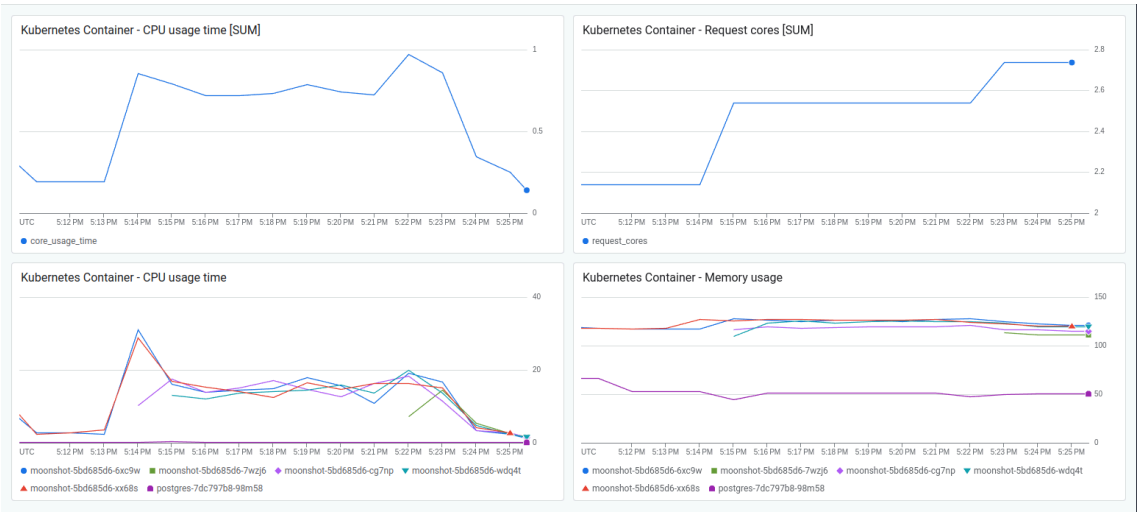

Figura 1: Monitorização do desempenho da aplicação com 500 usuários (com HPA)

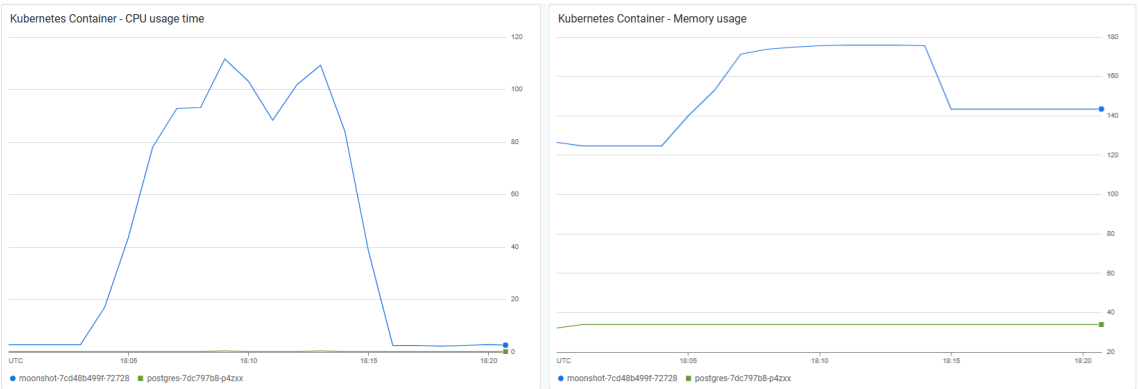

Figura 2: Monitorização do desempenho da aplicação com 500 usuários (sem HPA)

5.2. Comportamento da CPU

Os gráficos de utilização de CPU mostram comportamentos significativamente diferentes entre os cenários com e sem HPA:

Com HPA: O consumo de CPU foi distribuído entre múltiplos pods. O gráfico de "Request cores" mostra que o HPA ajustou dinamicamente os recursos, aumentando de 2.2 para 2.8 cores conforme a demanda, permitindo uma distribuição mais eficiente da carga.

Sem HPA: O consumo de CPU apresentou picos mais intensos, chegando a 110% de utilização, com uma curva mais acentuada de crescimento e queda. A ausência de múltiplos pods resultou em sobrecarga do único pod disponível, refletindo nos tempos de resposta significativamente maiores (média de 37.764s contra 9.507s com HPA).

5.3. Comportamento da Memória

Com HPA: A utilização de memória manteve-se estável em torno de 100-150 MiB por pod, com variações suaves entre os diferentes pods. O PostgreSQL apresentou um consumo constante de aproximadamente 50 MiB.

Sem HPA: O consumo de memória mostrou um padrão mais intenso, escalando de 120 MiB para 160 MiB durante o pico de carga, embora tenha se mantido dentro de limites aceitáveis. O PostgreSQL manteve um consumo estável em torno de 40 MiB.

5.4. Escalabilidade e Balanceamento de Carga

Com a ativação do HorizontalPodAutoscaler (HPA), o sistema demonstrou capacidade significativamente melhor de gerenciar a carga de 500 usuários:

O **tempo médio** de resposta **reduziu** de 37.764s (sem HPA) para 9.507s (com HPA)

A **distribuição da carga** entre múltiplos pods permitiu um **processamento mais eficiente** das requisições.

5.5. Ferramentas utilizadas

Para a realização dos testes de desempenho, selecionámos o **Apache JMeter**, lecionada nas aulas práticas. Esta ferramenta possibilita a **simulação de diferentes níveis de carga** e procede à **recolha detalhada de estatísticas** sobre o **desempenho** da aplicação face às várias cargas simuladas.

No que diz respeito à monitorização das métricas específicas das máquinas alojadas na Google Cloud, nomeadamente **CPU e memória**, optámos pelo **Google Cloud Monitoring**. Esta ferramenta revelou-se fundamental ao disponibilizar informações pormenorizadas sobre o estado dos diversos componentes da nossa aplicação no ambiente **Google Cloud**, oferecendo ainda a possibilidade de criar Dashboards personalizados com as métricas pretendidas.

5.6. Escalamento automático

A análise dos gráficos demonstra claramente o impacto do HPA no gerenciamento de recursos. No cenário **com HPA**, observamos:

- **Múltiplos pods** (moonshot) compartilhando a carga de trabalho
- **Distribuição mais uniforme** do consumo de CPU entre os pods
- **Escalonamento gradual** dos recursos conforme a demanda
- Melhor estabilidade no consumo de **memória**

Em contraste, o cenário **sem HPA** mostra:

- **Um único pod** absorvendo a carga toda
- **Picos** mais **intensos** de CPU
- **Maior instabilidade** no processamento das requisições

Analisando o gráfico de monitorização com HPA ativo, notamos que quando a utilização de CPU aumenta, o HPA detecta esse aumento de carga e inicia automaticamente a criação de pods adicionais. Este processo é claramente visível no gráfico através das múltiplas linhas que surgem ao longo do tempo, representando os novos pods (`moonshot-5bd685d6-6xc9w`, `moonshot-5bd685d6-7wzj6`, etc.).

À medida que cada novo pod se torna operacional, observa-se uma redução gradual na utilização de CPU do pod original, pois a carga de trabalho passa a ser distribuída entre todas as instâncias disponíveis. Este comportamento de balanceamento é evidenciado pela convergência das linhas de utilização de CPU dos diferentes pods para níveis mais moderados e estáveis.

Esta dinâmica demonstra a **eficácia** do HPA em realizar o escalamento horizontal automatizado, respondendo **proativamente** ao aumento da demanda e garantindo uma **distribuição equilibrada** da carga entre todos os pods ativos no sistema.

6. Conclusão

Com base na análise detalhada apresentada no relatório, podemos concluir que a implementação do **projeto Moonshot** na **Google Cloud Platform** demonstrou resultados significativos em termos de escalabilidade e desempenho. A utilização do **HorizontalPodAutoscaler** (HPA) provou ser uma estratégia eficaz para gerenciar cargas variáveis, como evidenciado pela significativa redução nos tempos médios de resposta.

Entretanto, os testes também revelaram algumas limitações e áreas para melhorias futuras:

- O **aumento na taxa de erro** sob cargas elevadas (chegando a 25.215% com 1000 usuários) sugere a necessidade de otimizações adicionais no sistema de escalabilidade.
- A **dependência de uma única instância** PostgreSQL representa um potencial ponto único de falha que poderia ser endereçado com implementação de replicação.

A **automatização** completa do processo de deployment usando **Ansible**, combinada com as capacidades de **monitoramento** do **GCD**, proporcionou uma **solução robusta** e **gerenciável**. As métricas recolhidas e a análise de desempenho fornecem uma base sólida para futuras otimizações e melhorias do sistema. Em suma, o projeto alcançou o seu objetivo principal de criar uma aplicação **escalável** e **resiliente** na **nuvem**.