Universidade do Minho
Escola de Engenharia

Análise de Desempenho de Armazenamento de Kubernetes

Laboratório em Engenharia Informática

Trabalho realizado por:

Daniel Regado, PG42577
Diogo Ferreira, PG42824
Filipe Freitas, PG42828
Vasco Ramos, PG42852

Trabalho orientado por:

Professor João Tiago Paulo

18 de Junho de 2021

Índice

Capítulo 1

Introdução

1.1 Contextualização

No panorama atual, a utilização de aplicações com *containers* é extremamente comum, para permitir o seu escalonamento, a sua resiliência, bem como permitir melhorar significativamente a sua performance, adaptando as necessidades de recursos das aplicações ao ambiente de utilização, balanceando a carga da aplicação entre vários servidores, e retirando aos desenvolvedores de *software* a responsabilidade da manutenção do *runtime* das aplicações. É neste sentido que surge o *Kubernetes*, um sistema de gestão de aplicações em *containers* que permite fazer o seu escalonamento, gestão, *deployment* e configuração automáticamente, providenciando também às aplicações alta resiliência e alta performance. Para suportar persistência de dados, *Kubernetes* suporta diversos tipos de *backends* de *storage*, tais como servidores *NFS*, vários tipos de *Cloud Providers* (Google, Amazon, Azure, etc), bem como outros tipos de serviços, mais adaptados às necessidades de cada entidade que pretende correr este tipo de aplicações.

Assim sendo, na sequência da UC de Laboratório em Engenharia Informática foi-nos proposta a realização de um trabalho relativo à avaliação da *performance* de várias aplicações a correr em *Kubernetes* com alguns tipos diferentes de *backends* de *storage*, de modo a poder retirar conclusões sobre as vantagens e desvantagens de cada um num ambiente real.

Todo o conteúdo associado ao presente trabalho, desde as configurações dos *deployments*, *storage*, resultados obtidos e gráficos gerados, é público e encontra-se disponibilizado no seguinte repositório do GitHub: `https://github.com/vascoalramos/kubernetes-storage-bench`.

1.2 Estrutura do Relatório

Este relatório encontra-se dividido em 6 partes, contendo também 3 apêndices:

A primeira parte é constituída por esta introdução;

Na segunda parte é feita uma introdução à plataforma *Kubernetes*;

Na terceira parte é feita uma análise das vantagens de desvantagens de diversas aplicações, sendo dada uma explicação de quais foram as aplicações escolhidas para efetuar a análise de desempenho e as razões para essa escolha;

Na quarta parte são descritos, em detalhe, os objetivos dos testes desenvolvidos, as ferramentas utilizadas para a análise de desempenho, bem como toda a configuração dos clusters *Kubernetes*, *Ceph*, das máquinas que correm esses clusters, e é também explicada a configuração das aplicações, o seu processo de instalação, e todos os restantes detalhes relevantes para a correta análise do desempenho das aplicações;

Na quinta parte são analisados os resultados dos testes efetuados às diversas aplicações, bem como são retiradas conclusões detalhadas dos resultados dos mesmos;

Na sexta e última parte são retiradas as conclusões finais relativas a todo o trabalho desenvolvido.

Nos apêndices estão presentes exemplos de configuração das aplicações que foram utilizadas para a análise de desempenho, bem como outros documentos pertinentes relativos à configuração dos componentes necessários â análise.

Capítulo 2

Introdução ao *Kubernetes*

Tal como referido antes, este trabalho tem o propósito de avaliar performance de *storage* em ambiente *Kubernetes*, dado este ser um sistema de orquestração de serviços bastante popular e consensual, o que o torna um dos mais utilizados atualmente. Deste modo, neste capítulo é feita uma breve e sucinta introdução a alguns conceitos chave de *Kubernetes*, que nos foi necessário entender para desenvolver o trabalho.

2.1 Funcionamento do *Kubernetes*

Kubernetes é uma ferramenta de orquestração de serviços que permite instalar e gerir aplicações em *containers* num conjunto de nós pertencentes ao *cluster* (de *Kubernetes*). Para além de instalações básicas, o *Kubernetes* dispõe de mecanismos para rapidamente atualizar, replicar e escalar aplicações, bem como dispõe de uma série de mecanismos diferentes para expor as mesmas para o exterior.

Enquanto que em *Docker* a unidade atómica da *stack* é o *container*, no *Kubernetes* a unidade atómica é o *pod*. Cada *pod* é uma coleção de *containers* e volumes que correm num ambiente isolado. Apesar de ser uma coleção de *containers*, o padrão mais usual, de acordo com a documentação, é ter um *container* por *pod*.

Intrínseco ao *Kubernetes*, existe um sistema de *labelling*, que pode ser aplicado a qualquer tipo de entidade (*pod*, *deployment*, *service*, etc), de forma a agrupar um conjunto de entidades do mesmo tipo. É baseado neste sistema que o *Kubernetes* agrupa *pods* para construir *deployments* e expor esses *pods* através *services*.

Por outras palavras, um *deployment* em *Kubernetes* consiste em agrupar um conjunto de *pods* criados através da mesma imagem numa entidade que é capaz de os gerir de forma transparente, o que permite ter várias instâncias da mesma aplicação representadas por uma associação lógica. Esta associação torna o processo de gestão (ou de escalar o número de instâncias da aplicação) mais simples e percetível.

Um *service*, em *Kubernetes*, não é mais que um mecanismo que permite expor um conjunto de *pods*, que podem ou não ser originários de *deployments*, num único ponto de acesso. Existem várias possibilidades de configurar estas entidades[1]. A especificação usada no decorrer do trabalho, por melhor se aplicar à infraestrutura disponível, foi a *Node Port*, que expõe a aplicação na mesma porta, em todos os nós do *cluster*.

Para as aplicações instaladas, estes foram os principais conceitos usados, sendo que existem outros conceitos que são derivações e/ou especificações destes.

2.2 *Backends* de *storage* suportados

Por definição, e segundo a documentação disponível em [1], o *Kubernetes* suporta uma série de *backends* de *storage*, entre eles o *AWS Elastic Block Storage*, *Azure Disk*, *Cinder*, *Ceph*, *iSCSI*, *NFS*, e *GCE Persistent Disk*. Alguns destes são implementações de *storage* dos principais *Cloud Providers* (*AWS*, *Azure* e *GCP*), enquanto que os restantes são alternativas comerciais ou *open-source*.

De referir que desta lista destacam-se opções como o *Ceph* e o *NFS*, ou as implementações dos *Cloud Providers* que permitem aceder à *storage*, tanto para escrita como para leitura, a partir de mais do que um *pod*, o que é bastante importante, visto que só assim é possível ter cenários de *deployment* com mais do que uma instância dos servidores aplicacionais das aplicações. Este fator influenciou a nossa decisão de *backends* de *storage* a testar.

2.3 Disponibilização de *storage* aos diversos *pods*

A disponibilização de *storage* aos diversos *pods* é realizada através de *Persistent Volumes (PVs)*. Um PV é, essencialmente, uma abstração no conceito de volume, que é gerido pelo *Kubernetes*, e montado nos diversos *pods* que assim o peçam (desde que tal seja possível). Esses pedidos são feitos sob a forma de *Persistent Volume Claims*. PVs podem ser configurados de dois modos: manualmente, pelo administrador do cluster *Kubernetes*, ou automaticamente, através da utilização de *Storage Classes*. Uma aplicação cria, depois, um *Persistent Volume Claim*, referenciando qual o volume que pretende ver montado. O *Kubernetes* tenta depois satisfazer esse pedido, dados os recursos que tem disponíveis.

Para criar PVs em *Kubernetes*, normalmente é utilizado o conceito de *Storage Class*. Uma *Storage Class* é um mecanismo em *Kubernetes* que se pré-configura, e que utiliza um *provisioner* para configurar, de um modo automático, novos *Persistent Volumes* e associá-los, automaticamente, às diversas *pods*, sem ser necessária intervenção humana (após a sua configuração inicial). Deste modo, a *storage* é abstraída da execução e *deployment* da aplicação, sendo que a sua manutenção fica à responsabilidade de uma equipa especializada para o efeito (normalmente denominada de *DevOps*).

[1] https://kubernetes.io/docs/concepts/services-networking/service

Por defeito, o *Kubernetes* incluí diversos tipos de *storage backends*, como servidores *NFS* (que utilizamos neste projeto), diversos *Cloud Storage Providers*, assim como o *Ceph* (que também utilizamos neste projeto), entre outros sistemas de *storage* persistente.

2.4 Criação de outros módulos de *storage*

A criação de outros módulos de *storage* em *Kubernetes* é possível, através de extensões, utilizando as APIs de *Container Storage Interface (CSI)* ou *FlexVolume*. Estes plugins podem ser criados independentemente do *source code* do *Kubernetes*, e é possível efetuar o seu *deployment* em qualquer cluster *Kubernetes* (que suporte as referidas APIs), para adicionar novas formas de *storage backends* a um cluster.

Até recentemente, a única maneira de expandir os tipos de storage fornecidos pelo *Kubernetes* era com *plugins in-tree*, ou seja, editando diretamente o código fonte do *Kubernetes*, o que tornava o seu desenvolvimento bastante mais difícil, e a sua manutenção e instalação também bastante mais difíceis.

Capítulo 3

Análise de Aplicações

3.1 Escolha das Aplicações

3.1.1 Definição de Critérios para as Aplicações

Antes de iniciarmos o processo de pesquisa de aplicações para utilizarmos no processo de *benchmark*, começamos por definir alguns critérios para estas. O primeiro e mais importante de todos é que fosse uma aplicação exigente em termos de disco e acessos a *storage* dado que pretendíamos exercer carga sobre esta.

Como segundo critério escolhemos simplicidade, não pretendíamos aplicações com demasiados componentes que tornassem esta difícil de instalar e fazer o benchmarking. Demos também preferência a aplicações que tivessem suporte para *Kubernetes* de modo a diminuir o tempo que iríamos ter de despender a efetuar as respetivas instalações, dado que esta tarefa não era o objetivo principal do trabalho.

Por fim, das aplicações que respeitavam estes dois critérios, de modo a simular diferentes tipos de uso, houve um esforço por escolher aplicações com diferentes requisitos em termos de *storage* e diferente propósito de uso, para podermos analisar se o tipo de uso influencia a performance do *backend* em causa.

3.1.2 Aplicações Escolhidas

3.1.2.1 *NextCloud*

É uma aplicação para armazenamento de ficheiros *open-source* similar à **Dropbox** e à **Google Drive**. Esta aplicação oferece suporte para *Kubernetes* sobre a forma de um *Helm Chart*, que nós modificámos para efetuar a instalação.

3.1.2.2 *Wiki.JS*

É uma versão *open-souce* da **Wikipédia** que permite aos utilizadores criar páginas, fazer comentários, implementada em Node.js. Esta plataforma também tinha suporte nativo para *Kubernetes*, que nós utilizamos como um ponto de partida para realizar a nossa instalação.

3.1.2.3 *PeerTube*

È uma aplicação *open-source* de partilha de conteúdo áudio e vídeo semelhante ao **YouTube**. A instalação desta aplicação também foi feita através de um *Helm Chart* disponibilizado, que nós alterámos conforme foi necessário.

3.2 Escolha dos *Backends* de *Storage*

Em relação aos *backends* de *storage* que iremos testar, escolhemos ***NFS*** e ***Ceph***. Existem muitas opções no que toca a *backends* de *storage* para *Kubernetes*, sendo que a maioria destas é paga. Assim, impusemos a limitação para escolher soluções *free* e *open-source*.

Escolhemos o *backend Ceph* pois queríamos testar um *backend* distribuído. Como irá ser explicado em mais detalhe à frente, na nossa instalação, este está distribuído por três máquinas, e efetua replicação de dados tornando-o tolerante a falhas. O *backend NFS* foi escolhido por já termos alguma familiaridade com este e por ser relativamente simples de instalar. Este corre em apenas uma máquina, ao contrário do *Ceph*.

Ponderámos, também, numa fase inicial testar a *storage* **Local** dos *pods*, mas visto que esta é efémera, não representa nenhum caso de uso real, e dado o objetivo do trabalho, optamos por não prosseguir com esta.

Capítulo 4

Metodologia de Testes

4.1 Objetivos dos testes

Com os testes que a seguir serão descritos temos como principal objetivo avaliar a performance dos diferentes módulos de *storage* abordados neste projeto. Pretendemos descobrir de entre os módulos escolhidos se existe algum que é superior aos restantes independentemente dos casos de uso, ou se diferentes módulos se adequam a casos de uso específicos. Queremos analisar também como as diferentes aplicações escolhidas com diferentes necessidades de armazenamento impactam a performance destes módulos e quais os mais adequados a cada uma destas.

4.2 Configurações dos vários componentes

4.2.1 Descrição geral

Para a execução dos testes, foram-nos disponibilizadas 8 máquinas com diferentes especificações. Três dessas máquinas estão a correr um *cluster Kubernetes* (em diante abreviado de *k8s*), configurado do modo que irá ser descrito mais à frente. Outras três máquinas têm configurado um *cluster Ceph* (ou, em alternativa, uma delas está a executar um servidor *NFS*), e as últimas duas máquinas, bastante menos poderosas, apenas irão ser utilizadas para efetivamente executar os *benchmarks* sobre as aplicações que estão a correr nas restantes máquinas. Estas máquinas estão na mesma rede do que as restantes para eliminar *bottlenecks* de rede.

Para fazer o *benchmark* das aplicações referidas vamos utilizar o *Locust*, uma ferramenta de *benchmarking* escrita em *Python* com testes configuráveis, escalonáveis, e de fácil utilização, que permite fazer a integração fácil com as aplicações, bem como permite distribuir o *workload* por várias máquinas, o que se revelou importante, pois os *workloads* escolhidos estavam a saturar os recursos de uma só máquina

de *benchmarking* muito rapidamente.

A figura 4.1 apresenta um diagrama que dá uma visão geral da configuração do sistema.

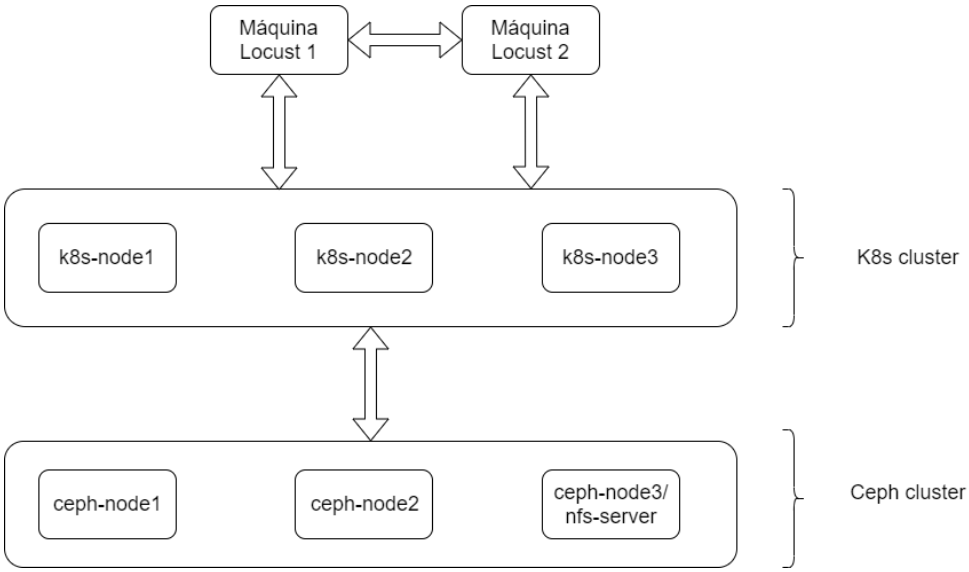

Figura 4.1: Visão geral da arquitetura do sistema

4.2.2 Configurações das máquinas

4.2.2.1 Configuração das máquinas de *benchmarking*

As máquinas de *benchmarking* tem as seguintes especificações:

- **CPU:** Intel(R) Core(TM) i3-3240 CPU @ 3.40GHz
- **RAM:** 8GB
- **Storage:** HDD 500GB SATA
- **OS:** Ubuntu 20.04.2 LTS

4.2.2.2 Configuração do cluster *Kubernetes*

As máquinas que alojam o cluster de *Kubernetes* têm as seguintes especificações:

- **CPU:** Intel(R) Core(TM) i3-4170 CPU @ 3.70GHz
- **RAM:** 8GB
- **Storage:** SSD SAMSUNG MZ7LN128 120GB
- **OS:** Ubuntu 20.04.2 LTS

O cluster *Kubernetes* foi configurado de uma forma simples com recurso ao Ansible, através dos Playbooks do Kubespray [2]. Para a disponibilização do servidor de *NFS*, foi utilizado o *Helm Chart* disponível em `https://kubernetes-sigs.github.io/nfs-subdir-external-provisioner/`. Separadamente, o servidor *NFS* foi configurado com o *Playbook Ansible* disponível na pasta *storage-backends/nfs-server* no repositório que acompanha o presente relatório. Relativamente ao *Ceph*, foi impossível arranjar um *provisioner* automático para configurar automaticamente PVs no *Kubernetes* e no *Ceph* que estivesse a funcionar com versões recentes do *Ceph* e do *Kubernetes*. Assim sendo, optamos por criar, manualmente, os PVs necessários para correr as aplicações, sendo que depois esses PVs foram associados manualmente a PVCs que foram também criados manualmente para cada aplicação. Essas aplicações depois utilizam os PVCs criados como forma de *storage*. Estes PVs utilizam, como *backend*, um servidor de *CephFS*, baseado, obviamente, no *Ceph*, pois este tipo de *backend* permite a montagem dos PVs como *ReadWriteMany*, o que é necessário para todas as aplicações que escolhemos. Relativamente à configuração do cluster *Ceph*, esta foi feita manualmente, com recurso à ferramenta *Cephadm*, cuja documentação está disponível em `https://docs.ceph.com/en/latest/cephadm/install/`. As configurações por defeito do cluster *Ceph* foram mantidas, tanto quanto possível, de modo a ter um ambiente replicável. Assim sendo, o cluster *Ceph* ficou configurado para fornecer 3 réplicas dos dados, que é o valor por defeito definido pelo *Ceph*.

A versão do cluster *Kubernetes* é a 1.20.4.

4.2.2.3 Configuração do cluster *Ceph*

As máquinas que alojam o cluster de *Kubernetes* têm as seguintes especificações:

- **CPU:** Intel(R) Core(TM) i3-7100 CPU @ 3.90GHz
- **RAM:** 16GB
- **Storage 1:** Samsung SSD 860 250GB SATA
- **Storage 2:** Samsung SSD 970 EVO Plus 250GB NVMe
- **OS:** Ubuntu 20.04.2 LTS

O cluster *Ceph*, durante a execução dos testes, está configurado para guardar os objetos nos discos NVMe mencionados, e também está configurado para executar nas três máquinas. Está igualmente configurado com um *size = 3*, no que é relativo à replicação e *sharding*, pois assim obtemos alguma resiliência sem comprometer demasiado a *performance* do *cluster*.

No entanto, uma destas máquinas está também configurada com o servidor de *NFS* (no disco NVMe). Apenas uma das configurações é executada a qualquer momento (i.e., quando estamos a fazer *benchmarking* do *Ceph*, apenas o *Ceph* está em execução; e quando estamos a fazer *benchmarking* do *NFS*, apenas o *NFS* está em execução).

Todas as instalações são replicáveis de acordo com os *playbooks* ou instruções providenciadas nos anexos deste relatório ou no repositório relativo a este projeto.

A versão do *cluster Ceph* é a 16.2.1. A versão do servidor *NFS* é a 1.3.4 (obtida a partir dos repositórios do Ubuntu).

4.2.2.4 Configuração do restante *Software*

O restante software em execução nas várias máquinas tem as seguintes versões e/ou configurações de instalação, todas obtidas a partir dos repositórios do Ubuntu:

- **Python**: 3.8.5
- **Locust**: 1.5.3
- **Collectl**: 4.3.1-1

4.2.3 Configuração das aplicações

Para a instalação das aplicações, o objetivo geral foi prepará-las de forma a que fossem escaláveis e suficientes para representar uma situação de stress para os mecanismos de *storage*, tentando garantir que os principais *bottlenecks* não estivessem nas aplicações em si. Contudo, não sendo as instalações o objetivo final do trabalho, em vez de preparar os *deployments* de raiz, decidiu-se utilizar o *package manager* construído para *Kubernetes*: *Helm*, cuja documentação se encontra disponível em [3]. Esta ferramenta funciona, de certa forma, como o *Docker Hub*, mas para *Kubernetes*. Permite criar *deployments* adaptativos e genéricos que podem ser reutilizados por qualquer pessoa, bastando apenas configurar o que for necessário. A sua estrutura é constituída essencialmente por *templates* YAML que são utilizados como um *package* completo, bastando apenas providenciar o ficheiro de configuração adequado. A estas estruturas dá-se o nome de *Helm Charts*. Para além de serem facilmente adaptáveis e extensíveis, os *Helm Charts* têm outra vantagem bastante importante para este trabalho: permitem construir novos *Charts*, utilizando *Charts* já existentes como componentes. No caso das nossas aplicações, esta possibilidade foi bastante importante, pois, permitiu-nos usar *Charts* pré-existentes para servidores de bases de dados como PostgreSQL e Redis (que já dispõe de mecanismos de *clustering* e replicação) e integrá-los nos nossos *deployments*, aumentando assim a robustez e agilidade das instalações.

Para além disso, é relevante dizer que as instalações foram pensadas de forma a definir explicitamente os recursos dos *Pods* dos servidores aplicacionais de cada aplicação. Foi também possível aplicar mecanismos de *auto-scaling*, o que permite à instalação lançar mais *Pods* conforme a carga aumenta e se torna insustentável para os *Pods* já existentes.

As próximas secções servem para especificar decisões de *deployment* específicas a cada uma das aplicações, dentro do contexto já descrito.

4.2.3.1 *NextCloud*

No caso da aplicação *NextCloud*, ao fazer a pesquisa sobre os seus mecanismos de *deployment*, o grupo encontrou um ***Helm Chart* oficial**, disponível em [4], para a instalação da mesma em *Kubernetes*. Por esta razão, decidiu-se utilizar o *Chart* oficial encontrado. Para poder operacionalizar a instalação, foi necessário customizar o ficheiro YAML de configuração de forma a que a mesma cumpra os requisitos do trabalho.

Para tal, fez-se as alterações necessárias de forma a que a instalação utilize uma base de dados relacional externa **MySQL**. Este servidor de base de dados está replicado numa tipologia *Master-Slave* com um servidor *Master* para escritas e leituras e dois servidores *Slave* para leituras e replicação de dados.

Algo que também foi tido em consideração foi ser possível tirar partido de um servidor uma base de dados **Redis** para tarefas pendentes e cache. Este servidor está replicado com uma tipologia Master-Slave, com configuração semelhante à do **MySQL**.

Tanto no caso do servidor MySQL como do Redis, a instância *Master* tem persistência, ao contrário dos dois *Slaves* que têm apenas *storage* interna, que é efémera.

O servidor aplicacional tem, também ele, persistência, para todos os ficheiros e pastas que são guardados na aplicação. Para além disso, foi estipulado que cada *Pod* tem de ter disponível pelo menos 350 unidades de CPU[1] do cluster.

Ainda relativamente ao servidor aplicacional, este foi configurado para escalar dinamicamente, através de um *Horizontal Pod Autoscaler* (HPA), de forma a que no máximo, existam 15 *Pods* destes. O *autoscaler* aumenta o número de *Pods* quando a utilização de CPU total dos *Pods* existentes no *deployment* ultrapassa os 55%. Por outro lado, quando o *autoscaler* identifica que a carga não justifica o número de *Pods* existentes, faz *downscale*.

Todos os *Pods* que têm persistência (ambas as bases de dados e os servidores aplicacionais) têm também a configuração dessa persistência, seja através de *Persistent Volume Claims* (PVCs) criados dinamicamente por *Provisioners*, ou através de PVCs criados manualmente. Esta é a parte da configuração que é necessário alterar quando se quer mudar o mecanismo de *storage* a utilizar (*NFS*, *Ceph*, etc).

A versão da imagem do container do Nextcloud utilizada é a versão 20.0.9 (apache).

A figura 4.2 mostra a arquitetura descrita para o *deployment* da aplicação *NextCloud*.

[1] Em *Kubernetes*, uma unidade de CPU equivale a um milésimo de CPU. No caso especificado, 350 unidades de CPU significa que cada *Pod* tem de ter disponível, pelo menos, 350 milésimos, isto é, 35% do CPU. Restante documentação disponível em [5].

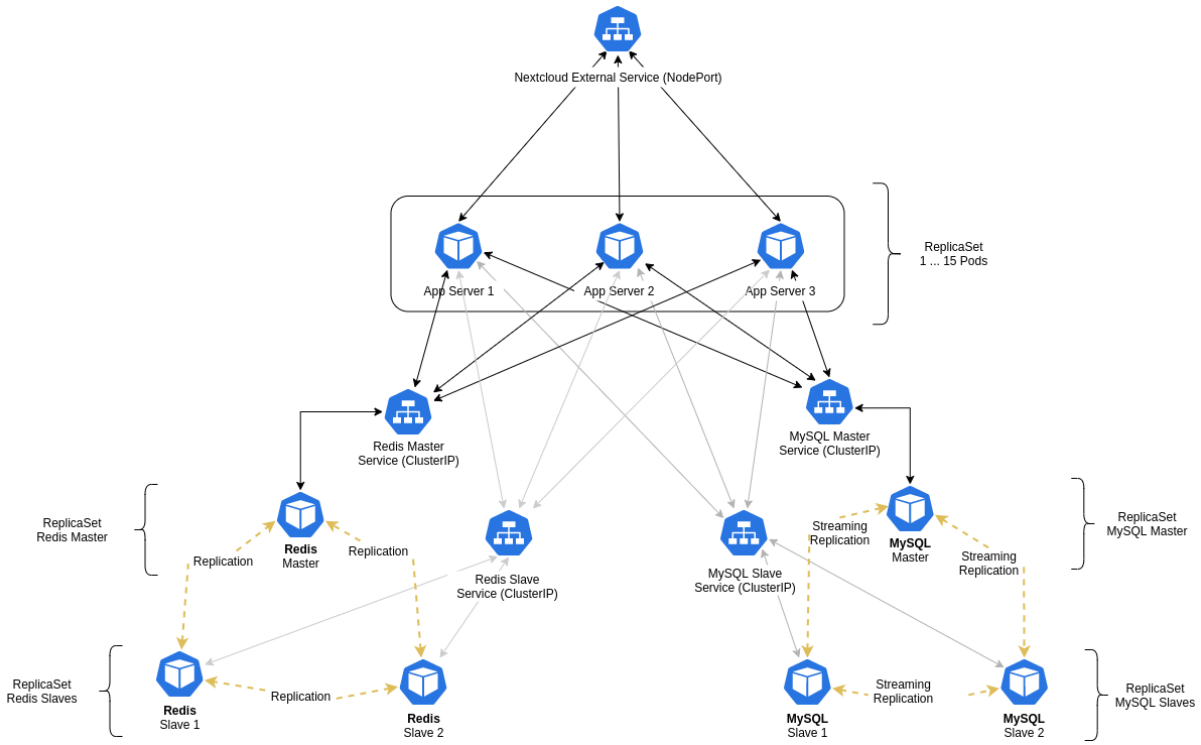

Figura 4.2: *Nexcloud* - Arquitetura de *Deployment*

O apêndice A mostra um exemplo do ficheiro de configuração YAML do *NextCloud*.

4.2.3.2 *Wiki.JS*

No caso da aplicação *Wiki.JS*, tal como no caso do *NextCloud*, existia um *Helm Chart* oficial [4] para a instalação em *Kubernetes*. No entanto, foram necessárias alterações e configurações adicionais de modo a que a instalação fosse ao encontro dos nossos objetivos. Por exemplo, foi adicionada uma configuração que especifica que o servidor aplicacional só deve começar o *deployment* depois da base de dados *PostgreSQL* ficar ativa, visto que tivemos problemas com tempos de espera e sincronização. Adicionalmente, foi também definida a configuração para a utilização do HPA, visto que esta não estava contemplada no *Chart* oficial.

Relativamente à base de dados *PostgreSQL*, está com replicação numa tipologia *Master-Slave*, onde o *Master* executa operações de leitura e escrita, e os dois *Slaves* tratam operações de leitura e replicação de dados. Trata-se de uma abordagem bastante semelhante à base de dados descrita para a aplicação *NextCloud*.

Em termos do servidor aplicacional, este apresenta também capacidades de escalar dinamicamente, através de um *Horizontal Pod Autoscaler*. Esta configuração de manutenção de *pods* relativos ao servidor aplicacional é bastante flexível, pelo que apresenta uma configuração idêntica à descrita anteriormente para o *NextCloud*. Ou seja, cada *pod* tem disponíveis 350 unidades de CPU, e sempre que a utilização do CPU ultrapassa os 55%, o número de *pods* ativos aumenta até um máximo de 15 *pods* simultâneos. Caso exista mais do que 1 *pod* ativo e a carga não justifique a existência de vários *pods*, estes serão também eliminados de forma a não reservar recursos não utilizados.

A versão da imagem do *container* utilizada pelo *Wiki.JS* é a 2.5.201 (*latest* durante a realização do trabalho).

A figura 4.3 mostra a arquitetura descrita para o *deployment* da aplicação *Wiki.JS*.

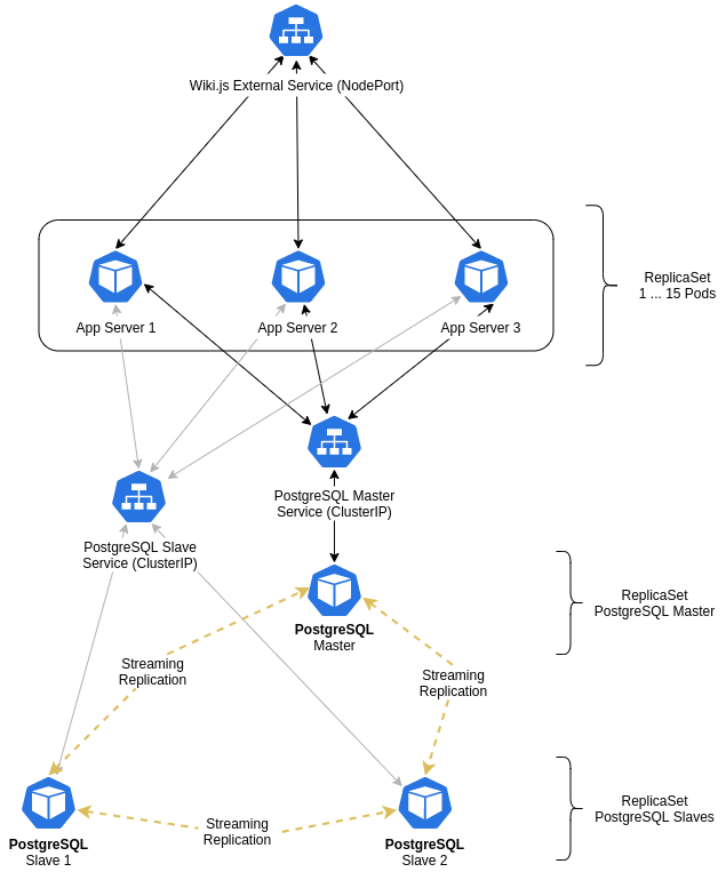

Figura 4.3: Wiki.js - Arquitetura de *Deployment*

O apêndice B mostra um exemplo do ficheiro de configuração YAML do *Wiki.JS*.

4.2.3.3 *PeerTube*

A instalação do *PeerTube*, tal como as duas aplicações referidas anteriormente, foi feita utilizando um *Helm Chart* pré-existente. Tal como nas aplicações anteriores, foi necessário alterar algumas das configurações por defeito, nomeadamente para a aplicação utilizar os *backends* de armazenamento que queremos testar. Para além disso, tal como no caso da aplicação *Wiki.JS*, foi necessário fazer algumas melhorias e correções no *Chart* de forma a cumprir os requisitos necessários.

Em relação às bases de dados, esta aplicação tem uma estrutura muito semelhante à da aplicação *NextCloud*. Temos uma base de dados relacional, *PostgreSQL*, a ser replicada numa estrutura *Master-Slave*. Possuímos também uma base de dados *Redis* que está a ser replicada da mesma forma que a base de dados *PostgreSQL*.

Num momento inicial, o servidor aplicacional desta aplicação, tal como os anteriores, tinha sido configurado para escalar dinamicamente, de acordo com a utilização do CPU. No entanto, após alguns testes, observámos que a aplicação não lidava bem com a existência de várias instâncias do servidor

aplicacional, mostrando-se incapaz de propagar a todas as instâncias alterações de configurações da aplicação e mostrando também conteúdos diferentes conforme a réplica que respondia. Deste modo optamos por ter apenas uma instância do servidor aplicacional e atribuir-lhe mais recursos, especificamente, atribuímos-lhe 1000 unidades de CPU.

No seguimento destes testes preliminares, observamos também que a maioria do tempo gasto durante um *upload* era a efetuar o *transcoding* do vídeo, o que é uma operação bastante exigente a nível de CPU. Assim sendo, de modo a evitar que o CPU se tornasse o *bottleneck* desta aplicação, decidimos desativar o *transcoding* de vídeos. Isto reduziu bastante o tempo de *upload* de um vídeo mas restringe os formatos aceites para apenas *MP4*.

A versão utilizada da imagem do *container* do *PeerTube* é a 3.2.1-buster (*latest* na realização do trabalho).

A figura 4.4 mostra a arquitetura descrita para o *deployment* da aplicação *PeerTube*.

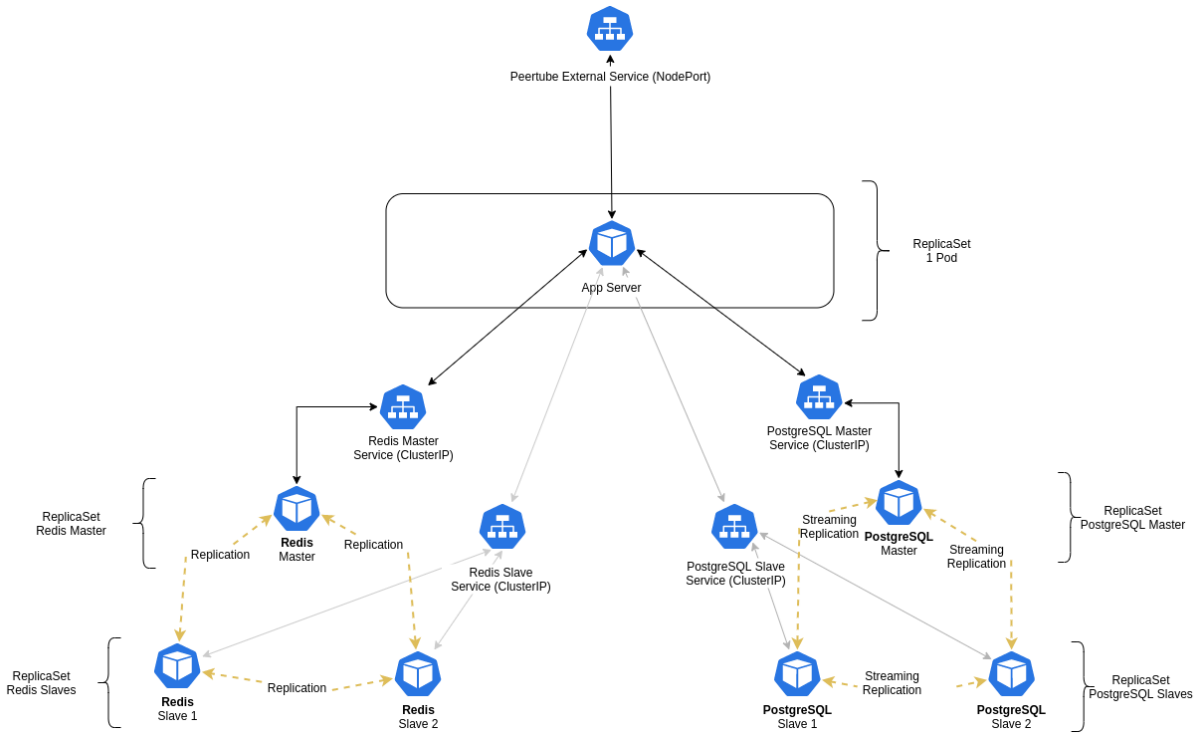

Figura 4.4: Peertube - Arquitetura de *Deployment*

O apêndice C mostra um exemplo do ficheiro de configuração YAML do *PeerTube*.

4.3 Especificações dos *benchmarks*

Como referido anteriormente, iremos utilizar a *framework* **Locust**, baseada em **Python**, para a execução dos *benchmarks*, principalmente devido à sua flexibilidade e extensibilidade para o desenvolvimento de testes. Durante a execução dos testes, serão recolhidas informações relativas à execução dos pedidos (*requests per second* - RPS, *failures per second* e *response time*) e relativas à monitorização da infraestrutura (tráfego de rede e utilização de disco, CPU e memória). Tanto para monitorização de operações de I/O sobre o disco, relativamente às maquinas que suportam o armazenamento, como monitorização de utilização de CPU, memória e tráfego de rede, nas máquinas do *cluster Kubernetes*, utilizaremos a ferramenta **collectl**.

Antes de entrar no detalhe de cada aplicação, é de referir que cada aplicação vai ser testada em três cenários diferentes: apenas escritas, apenas leituras e simulação do utilizador. Estes serão executados para um diferente número de utilizadores e para os diferentes mecanismos de *storage*, como é especificado na metodologia abaixo descrita.

Além disso, é relevante salientar que devido à execução dos *benchmarks* em ambiente distribuído, cada cenário de testes tem implementado uma configuração para *master* e outra para *worker*. De modo a tirar o máximo partido das máquinas disponíveis para executar tais testes (descritas em 4.2.2.1 como *dual core*), serão executados um total de 4 *workers* para cada *benchmark*, segundo a recomendação na documentação do *Locust*, de corresponder um *worker* por cada CPU *core* disponível. O *master* estará alocado no computador que iniciará os *benchmarks*, não nas máquinas de *benchmark*. Isto pode ser feito porque foi certificado que o *master* não interage diretamente no *benchmark*, apenas divide as tarefas pelos *workers* e agrega os dados provenientes destes.

Foi-nos também proposta a utilização do *CNSBench*, um *benchmark* desenhado para ser *cloud-native*, ou seja, avaliar a performance de *storage* diretamente em *clusters cloud*, como são os *clusters Kubernetes*. Após a análise do artigo que descreve este novo *benchmark*, e após realizar uma análise custo-benefício, decidimos não o utilizar devido à sua curva de aprendizagem e tempo disponível, bem como todo o processo de instalação. Tornou-se claro que, considerando todo o trabalho a desenvolver para o utilizar, a sua utilização não iria trazer grandes benefícios ao nosso trabalho, sendo a maior parte dos seus benefícios negligíveis em comparação com os *benchmarks* já desenvolvidos para as aplicações que vamos analisar.

4.3.1 *NextCloud*

Tal como referido acima, para cada aplicação implementou-se três cenários de teste:

- **Cenário de apenas leituras:** consiste na sequência de leituras (*downloads*) de ficheiros existentes no *NextCloud*. Este cenário contém um passo inicial para adquirir a lista de todos os ficheiros existentes no servidor de forma a poder fazer os pedidos de *download*. Neste cenário de teste, o tempo entre cada pedido, por utilizador, é de 5 segundos. No servidor existem 17 ficheiros diferentes com tamanhos entre os 5.3Kb e os 7.8Mb.

- **Cenário de apenas escritas:** consiste na sequência de escritas (*uploads*) de ficheiros e criação de pastas. Para o universo de *uploads*, foi utilizado um conjunto de 100 ficheiros (documentos, PDFs e imagens) cujos tamanhos variam entre os 1.9Kb e os 8.5Mb. Apesar de a amplitude entre tamanhos ser bastante grande, dado que os testes vão ser executados por um número relevante de utilizadores em simultâneo e durante um tempo considerável, esta amplitude acaba por não ter grande relevância ou impacto negativo. Também este cenário contém um passo inicial, neste caso para adquirir a lista de todos os ficheiros presentes no nosso conjunto, disponíveis para fazer *upload*, e carregá-los para memória. Neste cenário de teste, o tempo entre cada pedido, por utilizador, é de 10 segundos.

- **Cenário misto (simulação do utilizador):** consiste na mistura dos dois cenários anteriores, através de uma ponderação de recorrência: as ações mais recorrentes ocorrem pela seguinte ordem: *download* de ficheiro, *upload* de ficheiro, criação de pasta, sendo a mais recorrente o *download* e a menos recorrente a criação de uma pasta. Tal como nos outros dois cenários, existe um passo inicial para: adquirir a lista de ficheiros existentes no *NextCloud* para os *downloads* e adquirir a lista do conjunto de ficheiros disponíveis localmente para os *uploads*, bem como carregá-los para memória. Neste cenário de teste, o tempo entre cada pedido, por utilizador, varia entre 5 e 25 segundos.

Algo que é comum a todos os cenários de teste da aplicação *NextCloud* é o facto de a escolha do ficheiro a fazer *download* ou *upload* é sempre aleatória, dado que o elevado número de execuções acaba por garantir que todos os ficheiros vão acabar por ser escolhidos, seguindo uma distribuição uniforme.

4.3.2 *Wiki.JS*

Os cenários definidos para benchmark desta aplicação foram os seguintes:

- **Cenário de leituras**: Neste cenário, existe apenas a leitura de páginas já existentes na aplicação, quer estas tenham sido geradas automaticamente, pelo cenário de apenas escritas, ou manualmente. Para tal, antes de cada leitura, é percorrida a diretoria base, que contém as pastas que constituem as páginas e os seus ficheiros de conteúdo. Uma vez escolhida uma pasta (de forma aleatória), cada utilizador irá ler a página contida nessa pasta. Entre cada operação de leitura, existe um tempo de espera constante de 10 segundos. É de salientar que os pedidos têm em consideração o conteúdo da página, pelo que uma vez obtida a página HTML, esta é percorrida de forma a carregar também imagens, vídeos, etc.

- **Cenário de escritas**: Neste cenário são executadas, exclusivamente, ações para a geração de novas páginas com conteúdo. Tendo em conta uma página genérica para uma aplicação deste tipo, as páginas criadas irão conter os seguintes tipos de conteúdo: imagens, vídeos, GIFs, áudios e texto. Cada página gerada irá conter o mesmo número de itens para cada tipo de conteúdo, e

estes irão compreender ficheiros com tamanho entre 100Kb e 5Mb, presentes num conjunto de ficheiros totalizando cerca de 83Mb. Cada página é gerada com um tempo de espera constante de 20 segundos.

- **Cenário misto**: Consiste na execução de tanto ações de escrita de páginas como de leitura. Considerando a frequência destas ações por parte de um utilizador, foi atribuído um peso superior à leitura de páginas, em relação às escritas. Isto porque, num cenário real, é mais provável que um utilizador leia mais páginas do que as escreve. Portanto, irão ser executadas mais ações de leitura, de modo a obter uma simulação de atividade realista. Em contraste com as escritas no cenário descrito anteriormente, o número de elementos de cada tipo de conteúdo é variável, de modo a obter um pouco mais de aleatoriedade. Cada ação é seguida de um tempo de espera entre 10 e 25 segundos.

4.3.3 *PeerTube*

Os cenários definidos para benchmark desta aplicação foram os seguintes:

- **Cenário de leituras**: este cenário consiste em executar consecutivamente o *download* de um vídeo escolhido aleatoriamente de uma lista através de um *link* disponibilizado pela aplicação. Não encontrámos nenhuma maneira de automatizar o processo de recolha dos *links* de *download* dos vídeos da plataforma, e como tal, estes têm de ser recolhidos manualmente. Estamos a realizar *downloads* pois a plataforma em questão assim o permite e também por não encontrarmos nenhuma ferramenta que funcionasse com esta aplicação de modo a simular o *streaming* de vídeos.

- **Cenário de escritas**: neste cenário fazemos *uploads* consecutivos de vídeos para a plataforma. Para cada *upload*, é escolhido aleatoriamente um vídeo de uma pasta, sendo que todos estes têm de estar no formato MP4 pois, tal como já foi referido anteriormente, este é o único aceite pela plataforma após ser desativado o *transcoding* de vídeos.

- **Cenário misto**: neste caso, combinamos os dois cenários anteriores e fazemos *uploads* e *downloads* de vídeos, sendo que fazemos com mais frequência *downloads* do que *uploads*.

4.4 Metodologia de *Benchmarking*

Esta secção irá especificar qual a metodologia de execução e avaliação de testes utilizada de forma a facilitar a posterior análise e discussão de resultados.

De forma a obter termos de comparação e mais dados para identificar possíveis *bottlenecks*, cada um dos cenários de teste será executado com diferentes níveis de carga, em termos de utilizadores simultâneos. Para isso, foram definidos os seguintes níveis de carga, que serão aplicados para cada

cenário, para todas aplicações. O número de utilizadores foi definido de forma a encontrar um equilíbrio entre cenários reais de utilização e capacidades computacionais do *cluster*, não sendo portanto o objetivo executar testes exageradamente desproporcionais e irrealistas. Em termos de tempo de execução, foi definido um período de 15 minutos para cada teste por ser considerado tempo suficiente para o *cluster* se adaptar à carga e estabilizar. Por fim, o número de novos utilizadores por segundo foi definido de forma a que todos os utilizadores do teste em questão estivessem instanciados no primeiro minuto de execução, de forma a igualar o tempo em carga máxima definida para cada nível de carga.

A tabela 4.1 apresenta um resumo da especificação de execução dos testes acima descrita.

# Utilizadores	*Spawn Rate* (utilizadores/seg)	Tempo de execução (min)
25	0.5	15
50	1	15
75	1.5	15
100	2	15

Tabela 4.1: Resumo da especificação da execução dos testes

A estratégia escolhida para executar os testes de *benchmarking* foi executar os testes de cada aplicação de forma mutuamente exclusiva, isto é, de forma a que a aplicação em teste fosse a única a correr no *cluster*, para termos um termo de nivelamento dos resultados.

Inicialmente, pensou-se que uma segunda fase de testes com várias aplicações a serem testadas em simultâneo seria interessante, contudo, após alguns testes iniciais percebeu-se que tanto o *PeerTube* como o *Wiki.JS* não permitiam tal interação devido a problemas de replicação e baixa performance, no caso do *PeerTube*; e excessivo consumo de recursos, no caso da *Wiki.JS*.

É importante também referir que antes de cada teste foi executado uma pré-execução do sistema, tanto para população de dados, como para garantir que o sistema não estava em total modo de repouso, o que iria prejudicar os primeiros minutos do teste, visto que se iria ter apenas um *Pod* do servidor aplicacional da aplicação em causa. Para além disso, após cada teste executado apagou-se todo o conteúdo criado resultante da execução desse teste, para garantir uma uniformidade e igualdade no estado das aplicações em cada um dos testes.

Capítulo 5

Análise de Resultados

Neste capítulo apresenta-se os resultados obtidos nos vários testes executados, bem como as conclusões possíveis de retirar dos mesmos.

5.1 *NextCloud*

5.1.1 *NFS*

5.1.1.1 Cenário de leituras

Relativamente ao cenário de leituras da aplicação *NextCloud*, com a *storage NFS*, a tabela 5.1 apresenta um resumo dos resultados obtidos para os diferentes níveis de carga aplicados. Analisando a tabela é possível ver que para todos os níveis de carga, não ocorreu qualquer tipo de erro ou exceção, pelo que a aplicação conseguiu lidar com todos os pedidos submetidos. Este indicador, quando analisado com os baixos tempos de resposta obtidos, visto que os tempos medianos de resposta nunca ultrapassaram os 75 milissegundos, permite perceber que o *NextCloud*, pelo menos em processos de leitura, não sofre de *bottlenecks* significativos relativamente a recursos do sistema (memória, CPU, rede, etc). Para além disto, olhando para os tempos de reposta, é possível ver que entre os diferentes níveis de carga, este valores oscilam um pouco sem uma regra aparente, contudo, sendo esta oscilações na ordem dos 10 milissegundos, o grupo não considera que esta oscilação seja algo de grande relevância.

Para a tabela apresentada seguidamente, os valores percentuais apresentados representam a comparação de um dado valor com o valor base correspondente, apresentado para a carga mínima testada, 25 utilizadores.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	2.43	73	0
50 +100%	9.58 +294.24%	66 -9.59%	0
75 +200%	14.29 +488.07%	74 +1.37%	0
100 +300%	19.10 +686.01%	65 -10.96%	0

Tabela 5.1: *NextCloud NFS*- Resumo dos resultados do cenário de leituras

Analisando de forma mais detalhada os resultados obtidos, e começando pelo nível de carga mais baixo, as figuras 5.1 e 5.2 mostram os gráficos relativos à evolução dos tempos medianos de resposta e *throughput* da aplicação durante o decorrer do teste, respetivamente. Na figura 5.1, relativa aos tempos de resposta, é possível ver que, exceto 2 picos aos 6 e 11 minutos do teste, os tempos de resposta mantêm-se bastante próximos do valor mediano obtido, o que mostra o serviço bastante constante. Este facto é também corroborado pelo gráfico da figura 5.2, que apresenta um valor de pedidos por segundo também bastante constante ao longo de todo o teste.

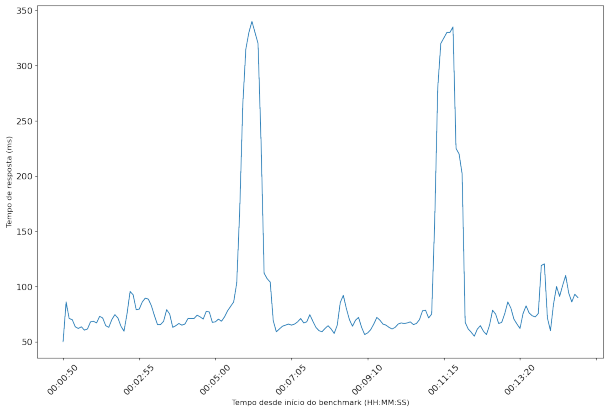

Figura 5.1: Tempos medianos de Resposta, 25 utilizadores

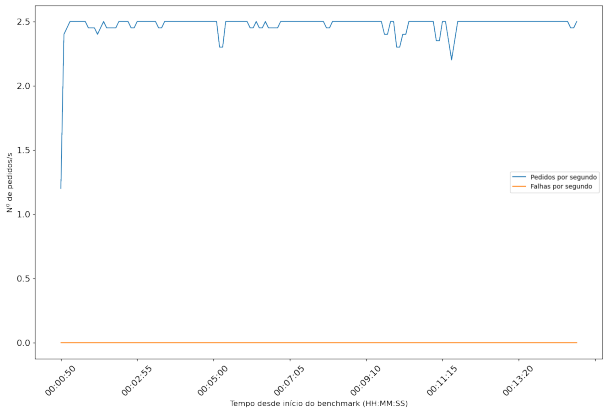

Figura 5.2: *Throughput* (pedidos/s), 25 utilizadores

Para além disto, olhando para as figuras 5.3 e 5.4 torna-se claro que no nível mais baixo de carga para o *NextCloud*, este não apresenta qualquer dificuldade em lidar com essa carga visto que a utilização mediana do CPU nunca ultrapassa os 15% e a utilização mediana de memória nunca ultrapassa os 55%.

É importante também notar que, e relacionando as figuras 5.1 e 5.3, é possível ver que os picos dos tempos de resposta existentes coincidem diretamente com os picos de utilização de CPU. Isto deve-se a um *cronjob* que corre de 5 em 5 minutos para executar tarefas de *background* inerentes ao próprio *NextCloud*. Este padrão aparece recorrentemente ao longo dos vários testes executados ao *NextCloud*, mas não tem impacto direto nos resultados e conclusões finais.

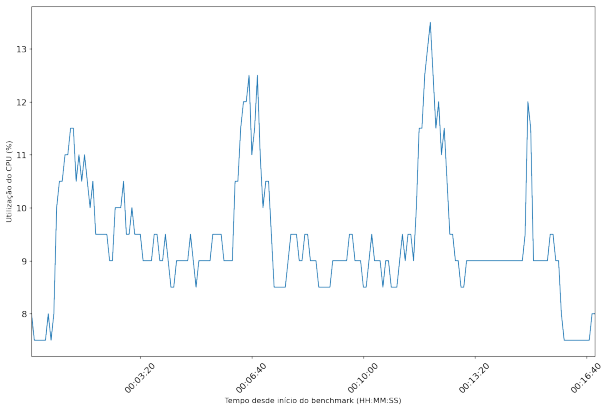

Figura 5.3: Utilização mediana do CPU (%), 25 utilizadores

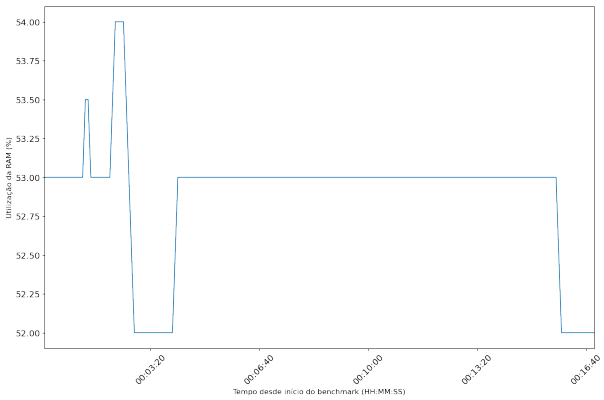

Figura 5.4: Utilização mediana de memória (%), 25 utilizadores

Tendo em conta que a evolução da performance e consumo de recursos, por nível de carga, apresentou-se ser uniformemente crescente, por uma questão de simplicidade, mostra-se os resultados detalhados do último nível de carga (100 utilizadores), visto que os níveis intermédios de carga não se traduzem em qualquer acrescento de informação relevante.

Deste modo, as figuras 5.5 e 5.6, permitem ver um comportamento muito semelhante ao das figuras 5.3 e 5.4, com a especial diferença dos valores serem taxativamente melhores com os 100 utilizadores, o que comprova que o *NextCloud* consegue lidar sem dificuldades com esta carga de utilizadores.

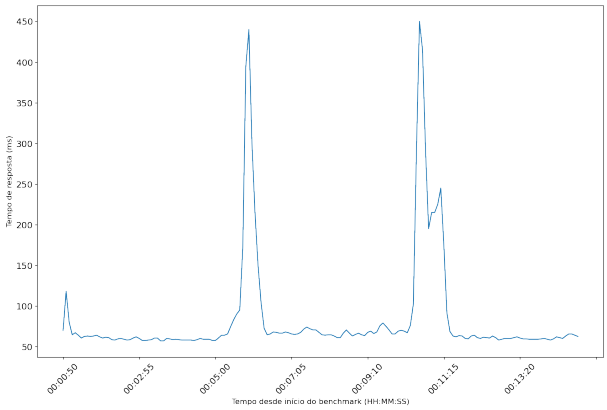

Figura 5.5: Tempos medianos de Resposta, 100 utilizadores

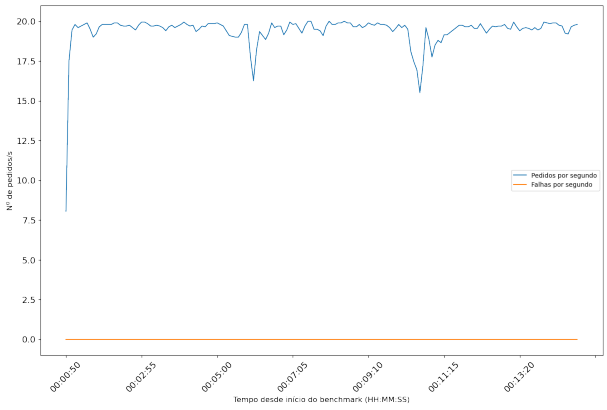

Figura 5.6: *Throughput* (pedidos/s), 100 utilizadores

Para além disto, olhando para os resultados de monitorização, presentes nas figuras 5.7 e 5.8, durante a execução deste cenário com os 100 utilizadores é possível ver que, também, neste nível de carga o *NextCloud* não aparenta ter *bottlenecks* de grande impacto (desconsiderando para já a *storage* que será analisado posteriormente, por comparação ao *Ceph*), visto que a utilização mediana do CPU nunca ultrapassa os 40% e a utilização mediana de memória nunca ultrapassa os 65%, que ainda não são valores notoriamente críticos.

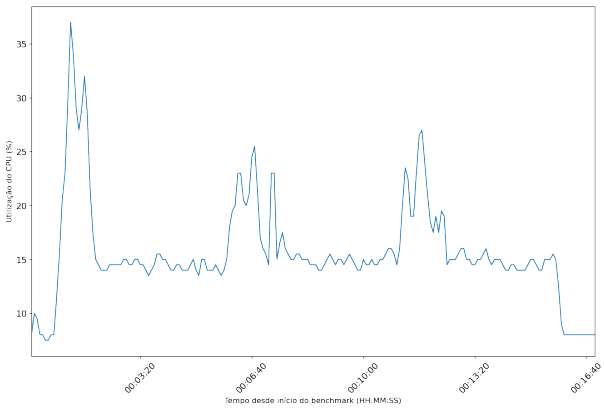

Figura 5.7: Utilização mediana do CPU (%), 100 utilizadores

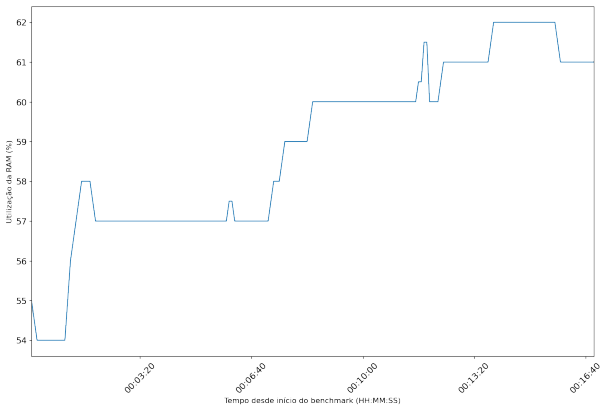

Figura 5.8: Utilização mediana de memória (%), 100 utilizadores

Por fim, analisando o gráfico presente na figura 5.9, é possível que ver que, ao contrário das escritas, não foi possível recolher métricas de leitura no disco, algo que foi transversal a todos os testes, para todas as aplicações. Esta particularidade foi apenas descoberta numa fase avançada dos testes e não foi possível resolver por parte do grupo, pelo que não foi possível perceber qual a razão visto que se experimentou com diferentes ferramentas de monitorização e o resultado foi semelhante com qualquer uma delas.

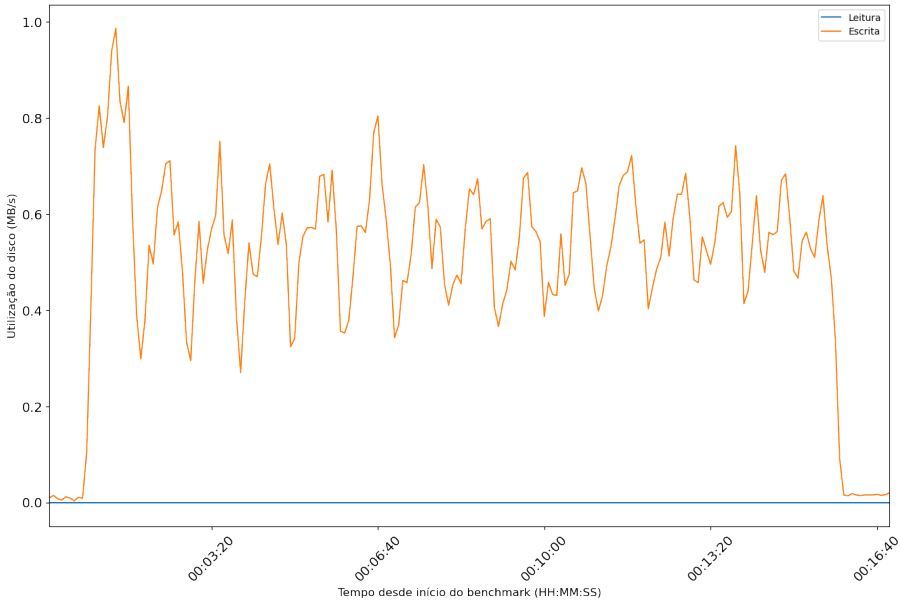

Figura 5.9: Utilização (débito) do cliente de *storage NFS*, para 100 utilizadores

5.1.1.2 Cenário de escritas

Relativamente ao cenário de escritas da aplicação *NextCloud*, com a *storage NFS*, a tabela 5.2 apresenta um resumo dos resultados obtidos para os diferentes níveis de carga aplicados. Analisando a tabela é possível ver que para todos os níveis de carga, apesar de não ser nula, ao contrário do cenário de leitura, a percentagem de erro das escritas é também esmagadormente baixa, estando sempre abaixo dos 0.05%,

demonstrando que também neste cenário a aplicação consegue responder com sucesso à grande maioria dos pedidos.

Comparativamente ao cenário de leituras é possível ver que houve uma diminuição dos indicadores de performance (diminuição do *throughput* e aumentos dos tempos de resposta), o que é espectável visto que, regra geral, as operações de escrita (*upload*) são mais custosas que as de leitura *(download*). Para além disto, é percetível uma diminuição na taxa de evolução de performance dos 75 para os 100 utilizadores, visto que o *throughput* não aumenta tanto comparativamente às restantes evoluções entre níveis de carga e existe também um aumento significativamente desproporcional do tempo mediano de resposta. Este diminuição na taxa de evolução de performance, mostra que o *NextCloud* no cenário de escritas começa a atingir um *bottlenecks* entre os 75 e os 100 utilizadores. Este fenómeno pode ser melhor observado no gráfico da figura 5.10, onde claramente se vê que entre os 75 e os 100 utilizadores é onde o *NextCloud* começa a atingir o seu "joelho" de performance.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	2.38	260	0.047
50 +100%	4.76 +100.00%	260 +0%	0.046
75 +200%	7.02 +194.96%	320 +23.08%	0
100 +300%	8.94 +275.63%	470 +80.77%	0.012

Tabela 5.2: *NextCloud NFS*- Resumo dos resultados do cenário de escritas

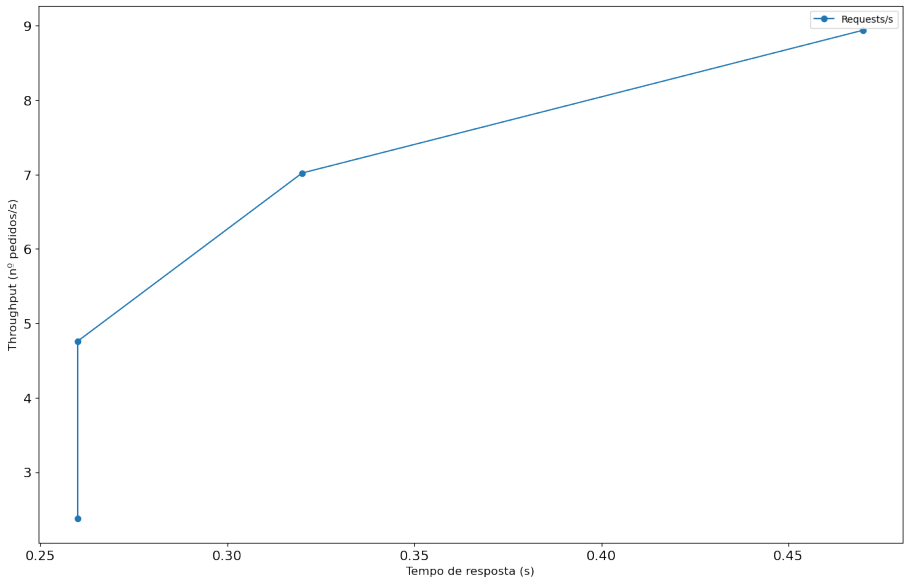

Figura 5.10: Relação entre *throughput* e tempo mediando de resposta, entre os vários níveis de carga

Tendo em conta que, para este cenário, os dois níveis de carga mais interessantes são os últimos dois, entre os quais existe uma quebra na taxa de evolução da performance, vamos analisar em maior detalhes esse dois últimos níveis.

Analisando primeiro o nível de carga com 75 utilizadores, os gráficos das figuras 5.11 e 5.12 mostram a evolução dos tempos de resposta e *throughput* no decorrer do teste, respetivamente. Ambos os gráficos não demonstram anomalias de relevo, exceto os picos já justificados anteriormente, indicando que com 75 utilizadores o sistema mantém-se relativamente estável, tendo em conta a carga submetida.

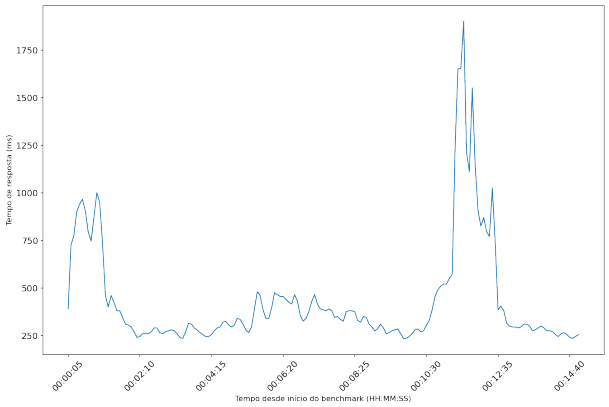

Figura 5.11: Tempos medianos de Resposta, 75 utilizadores

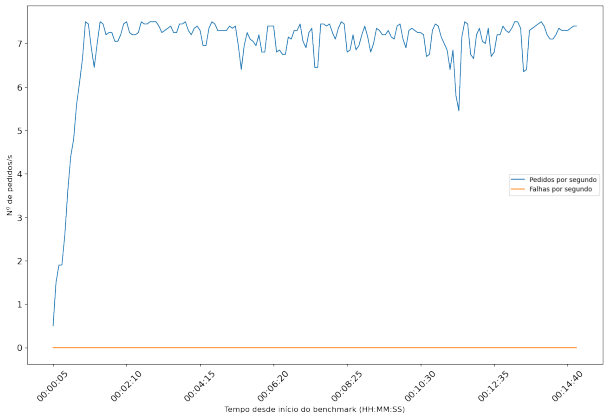

Figura 5.12: *Throughput* (pedidos/s), 75 utilizadores

Por outro lado, ao olharmos para o gráfico da figura 5.13, que representa a utilização mediana de memória, em percentagem, durante a execução do teste, é possível começar a perceber a possível razão para a quebra de performance no nível de carga seguinte. Olhando para o gráfico é possível perceber que no decorrer deste teste, a utilização de memória do *cluster* já ficou em valores relativamente elevados, o que poderá ser um constrangimento ainda maior com 100 utilizadores, afetando diretamente a performance da aplicação.

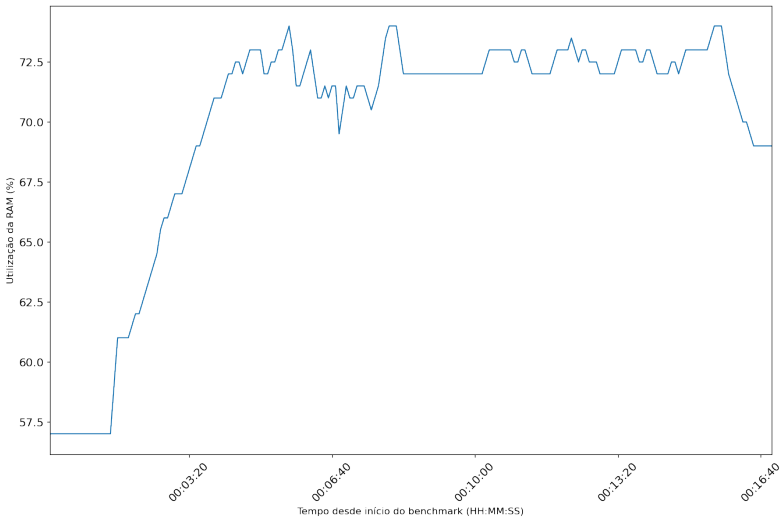

Figura 5.13: Utilização mediana de memória (%), para 75 utilizadores

Mudando agora o foco para o nível de carga com 100 utilizadores, e olhando para os gráficos das figuras 5.14 e 5.15 não se identifica uma degradação direta de performance no decorrer do teste, exceto os picos já discutidos anteriormente, contudo é possível perceber, principalmente no gráfico do *throughput*

que a capacidade de resposta já se demonstra um pouco mais irregular que nos resultados anteriores, dado que existe uma maior amplitude de oscilação do valor de *throughput*.

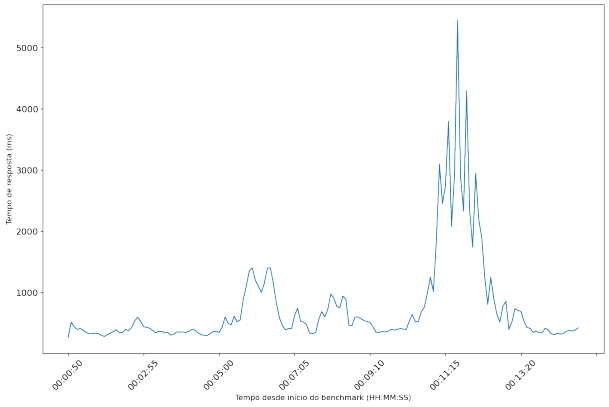

Figura 5.14: Tempos medianos de Resposta, 100 utilizadores

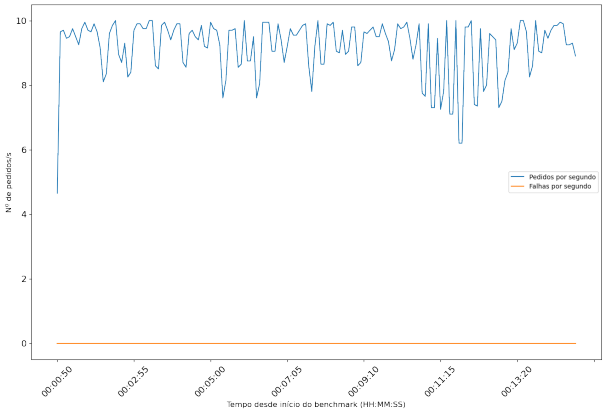

Figura 5.15: *Throughput* (pedidos/s), 100 utilizadores

Quanto à possível causa da observada quebra de performance mencionada acima, o gráfico da figura 5.16 acaba por confirmar a hipótese anterior que aponta a elevada utilização de memória como a razão mais plausível para a quebra de performance do *NextCloud* no cenário de escritas visto que, apesar de nos minutos finais baixar para valores na ordem dos 70%, durante grande parte do teste a utilização de memória do *cluster* encontra-se entre os 72% e os 80%.

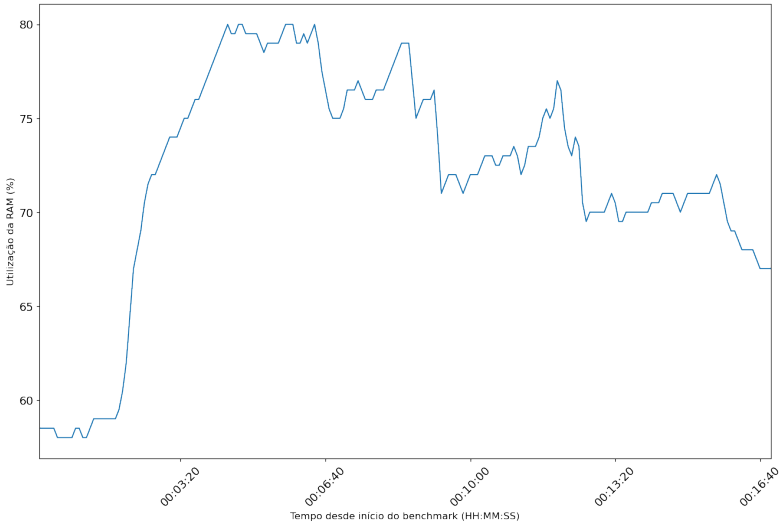

Figura 5.16: Utilização mediana de memória (%), para 100 utilizadores

5.1.1.3 Cenário misto

Por fim, vamos olhar para o último cenário de avaliação do *NextCloud* com o *NFS*, em que se simula a utilização espectável da aplicação, por parte dos utilizadores.

Analisando a tabela 5.3, a primeira característica que é possível destacar é que, no geral os valores tanto do *throughput* como do tempo mediano do resposta encontram-se em escalas intermédias comparando com os valores das tabelas 5.1 e 5.2, referentes aos dois cenários de teste anteriores. Este comportamento já era esperado visto que este cenário é, por si só, uma utilização equilibrada de escritas e leituras, consoante o que seria a utilização normal da aplicação.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	2.41	180	0
50 +100%	4.85 +101.24%	265 +47.22%	0.023
75 +200%	7.21 +199.17%	170 -5.56%	0
100 +300%	9.65 +300.41%	160 -11.11%	0

Tabela 5.3: *NextCloud NFS*- Resumo dos resultados do cenário de simulação do utilizador

Olhando para a percentagem de erros é possível ver que o único nível de carga que teve erros foi com 50 utilizadores e ainda assim a percentagem é tão baixa que é irrelevante.

Quanto à performance da aplicação neste cenário não é percetível qualquer quebra, o que acaba por ser espectável visto que, como já tinha sido referido no capítulo da metodologia, neste cenário de simulação as operações de leitura (*download*) são bastante mais recorrentes que as de escrita (*upload*), tal como acontece na utilização normal de uma *drive*.

Relativamente à performance do sistema, obtida pela monitorização, todas as métricas apresentaram resultados dentro do normal e esperado, pelo que não existe qualquer indicador de *bottlenecks* relativo a este cenário em específico, o que permite concluir que o *NextCloud*, com o *NFS*, é capaz de servir de forma aceitável as necessidades dos utilizadores.

5.1.2 *Ceph*

5.1.2.1 Cenário de leituras

Analisando agora os resultados do cenário de leituras da aplicação *NextCloud*, com a *storage Ceph*, e comparando estes resultados com os resultados já obtidos com o *NFS*, a tabela 5.4, apresenta um resumo dos resultados para os diferentes níveis de carga aplicados.

Tal como no cenário com o *NFS*, também com o *Ceph*, no cenário de leituras, a percentagem de erros continua a 0% em todos os níveis de carga, o que seria de esperar visto que o *Ceph*, mantendo todas as outras condições, no limite iria favorecer apenas as operações de leitura da aplicação.

Relativamente a performance, também aqui se manteve a linha de evolução uniformemente crescente, como no *NFS*, contudo o *NextCloud* conseguiu uma maior performance visto que para os mesmos níveis de carga conseguiu um maior *throughput* com os tempos medianos de resposta a rondar sensivelmente os mesmos que no *NFS*, o que por si só já representa um ganho considerável.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	4.81	72	0
50 +100%	11.94 +148.23%	70 -2.78%	0
75 +200%	17.90 +272.14%	69 -4.17%	0
100 +300%	23.85 +395.84%	70 -2.78%	0

Tabela 5.4: *NextCloud Ceph* - Resumo dos resultados do cenário de leituras

Após analisar os dados de monitorização relativo a este cenário e compará-los com os respetivos dados do *NextCloud* com *NFS*, foi possível perceber que as condições não mudaram a nível de utilização de memória e CPU. Os valores de utilização de disco também não foram reveladores, visto que, tal como já foi dito, não foi possível captar utilização de disco relativamente a leituras. Deste modo, para melhor perceber as diferenças deste cenário entre o *NFS* e *Ceph*, resta-nos apenas analisar a comparar a evolução do *throughput* e tempo mediano de resposta da aplicação, em ambos os mecanismos de *storage*. Para isso iremos analisar esta comparação nos dois níveis de carga mais representativos: 25 e 100 utilizadores, ou seja, primeiro e último níveis de carga.

Analisando primeiro o nível de carga dos 25 utilizadores, e olhando para o gráfico das figura 5.18, é possível ver com maior claridade o ganho de performance, a nível de *throughput*, que no caso do *Ceph* passou para sensivelmente o dobro dos valores obtidos com o *NFS*.

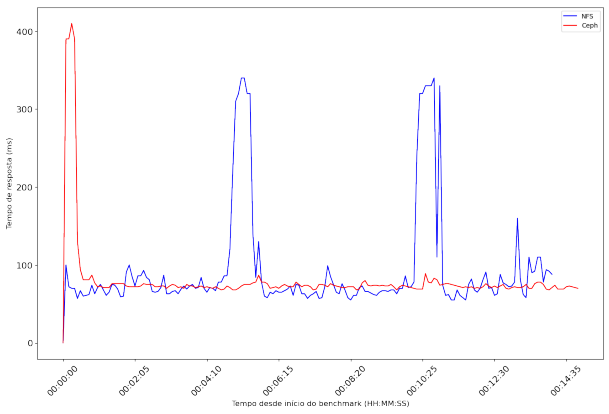

Figura 5.17: Comparação dos Tempos medianos de Resposta, *NFS* e *Ceph*, 25 utilizadores

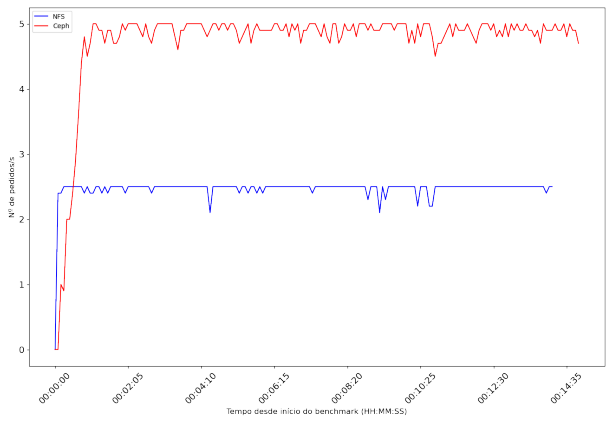

Figura 5.18: Comparação do *Throughput* (pedidos/s), *NFS* e *Ceph*, 25 utilizadores

Olhando para o gráfico da figura 5.17, é possível ver que após um pico inicial associado ao início do teste, os tempos medianos de resposta do *Ceph* são mais constantes e não sofrem de tantos picos como no caso do *NFS*. Contudo, visto que, se se ignorar os picos de ambos, os valores mais recorrentes tanto para o *NFS* como para o *Ceph* encontram-se na mesma gama, o que deixa a questão de como os valores de *throughput* poderam ter uma diferença tão grande, já que se manteve os mesmos níveis de carga entre os dois tipos de *storage*. Para melhor esclarecer esta dúvida, o grupo decidiu analisar não só os tempos medianos de resposta, que em algumas situações podem não ser totalmente representativos, mas

também os tempos **médios** de resposta, para os cenário de leitura. Esses valores podem ser observados na tabela 5.5, que confirmam o suposição que mais sentido faz de que os tempos de resposta efetivamente desceram no *Ceph*, comparativamente ao *NFS*.

	Tempo médio de resposta (ms)	
# Utilizadores	NFS	*Ceph*
25	119	90 -24.37%
50	96	88 -8.33%
75	123	85 -30.89%
100	112	88 -21.43%

Tabela 5.5: *NextCloud - Ceph* vs *NFS*- Tempos médios de Resposta

Do mesmo modo, analisando o nível de carga com 100 utilizadores, e tendo como referência os gráficos das figuras 5.19 e 5.20, as tendências observadas para os 25 utilizadores mantêm-se, apesar de não serem tão notórias, principalmente no caso do *throughput*.

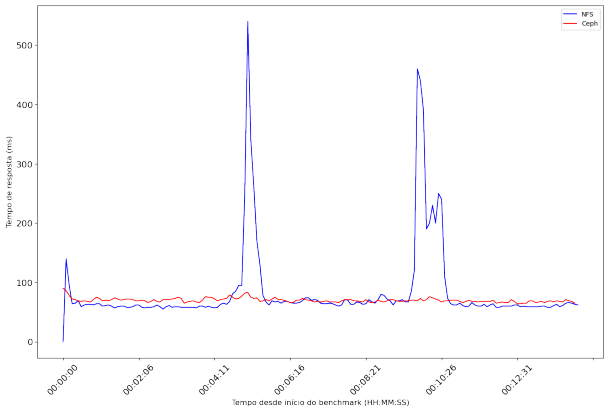

Figura 5.19: Comparação dos Tempos medianos de Resposta, *NFS* e *Ceph*, 100 utilizadores

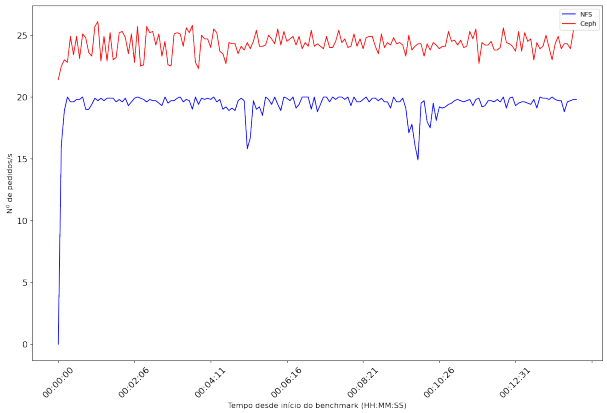

Figura 5.20: Comparação do *Throughput* (pedidos/s), *NFS* e *Ceph*, 100 utilizadores

Esta análise permite concluir que, indubitavelmente, o *NextCloud*, em operações de leitura (*download*), beneficia da utilização do *Ceph* como mecanismo de *storage*, o que faz sentido visto que os dados estão replicados, no caso dos testes, em 3 nós diferentes, o que resulta no aumento do *throughput* de leituras, tal como se constatou.

5.1.2.2 Cenário de escritas

Passando para os resultados obtidos no cenário de escritas da aplicação *NextCloud* com *Ceph*, e analisando o resumo presente na tabela 5.6, é possível de forma resumida perceber que as percentagens de erro voltam a ser baixas o suficiente para não representarem qualquer evidência de constrangimento no normal funcionamento da aplicação. Para além disso, os valores de *throughput* e tempo de resposta

voltam a seguir uma tendência semelhante à referida no caso do *NFS*, em que se nota uma quebra na evolução da performance entre os 75 e 100 utilizadores, como é possível observar no gráfico da figura 5.21.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	2.94	270	0.038
50 +100%	5.84 +98.64%	300 +11.11%	0
75 +200%	8.50 +189.12%	350 +29.63%	0.04
100 +300%	10.55 +258.84%	790 +192.59%	0

Tabela 5.6: *NextCloud Ceph* - Resumo dos resultados do cenário de escritas

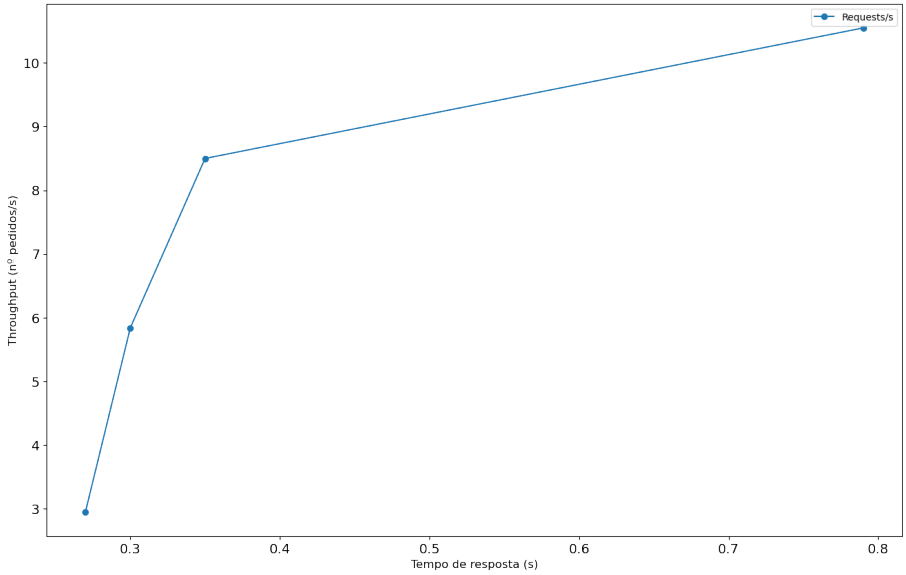

Figura 5.21: Relação entre *throughput* e tempo mediando de resposta, entre os vários níveis de carga

Comparando os resumos dos cenários de escrita nas tabelas 5.2 e 5.6, para o *NFS* e *Ceph*, respetivamente, é possível perceber que também aqui o *Ceph* apresentou um maior *throughput*, apesar de apresentar tempos de resposta considerável maiores. Quanto a este aumento dos tempos de resposta, percebeu-se que sucedia do facto do modo como a replicação no *Ceph* funciona, que faz com que quanto mais o teste avança, mais a *storage* fica sobrecarrega com processos de replicação que ainda estão a decorrer, o que acaba por aumentar os tempos de resposta dos pedidos que ocorrem mais no fim do teste, resultando no aumento significativo constatado. Este comportamento pode verificar-se no gráfico da figura 5.22, que compara os tempos de resposta entre o *NFS* e *Ceph*, para o cenário de escritas com 100 utilizadores.

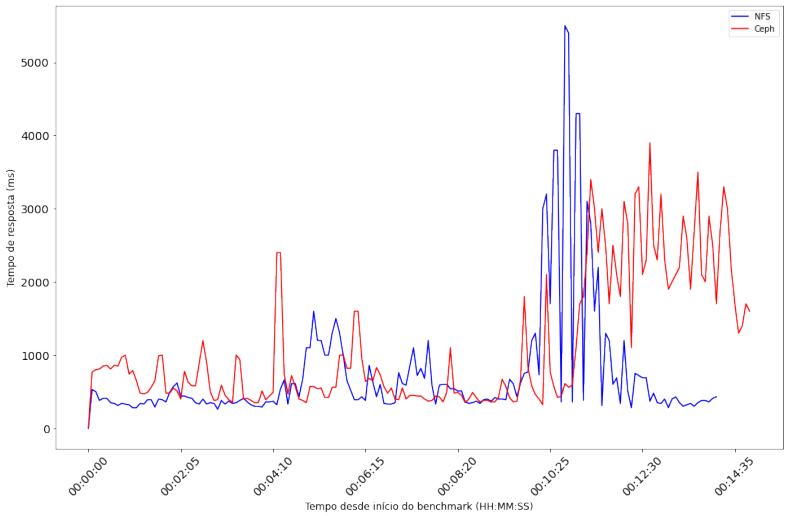

Figura 5.22: Comparação dos Tempos medianos de Resposta, *NFS* e *Ceph*, 100 utilizadores

Por fim, e analisando o primeiro e últimos níveis de carga, é útil comparar o débito de escritas dos clientes de *storage* (*NFS* e *Ceph*). Os gráficos das figuras 5.23 e 5.24 mostram essa comparação para os 25 e 100 utilizadores, respetivamente.

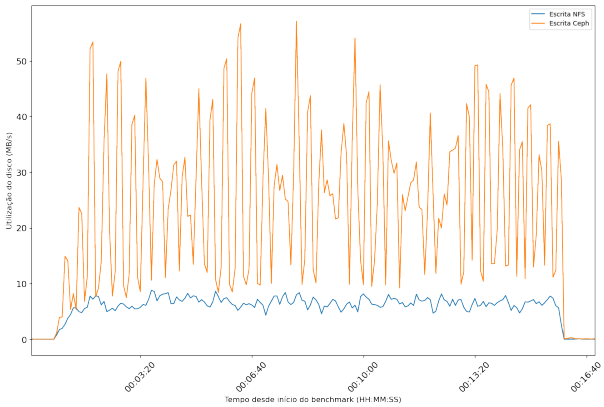

Figura 5.23: Comparação do débito de escritas dos clientes de *storage*, *NFS* e *Ceph*, 25 utilizadores

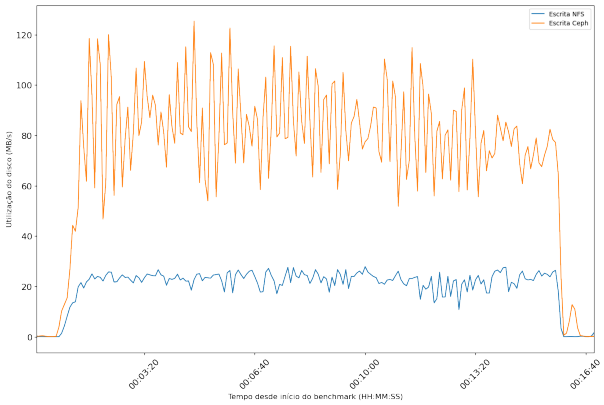

Figura 5.24: Comparação do débito de escritas dos clientes de *storage*, *NFS* e *Ceph*, 100 utilizadores

O gráfico da figura 5.23 permite perceber que para um baixo nível de carga o *Ceph* não é muito mais útil que o *NFS*, visto o débito do cliente *Ceph* (que representa a capacidade total do *cluster Ceph*, que dispõe de 3 nós) ultrapassa o débito do *NFS* de forma basta irregular e não muito significativa.

O mesmo já não é verdade no gráfico da figura 5.24, em que se vê claramente que o *cluster Ceph* é capaz de atingir débitos de escrita largamente superiores aos débitos de escrita do servidor *NFS*, para uma carga de 100 utilizadores.

Da análise feita para este cenário é possível concluir que, apesar dos aumentos dos tempos de resposta no cenário de escrita, o *NextCloud* beneficia da utilização do *Ceph* em situações de escrita intensiva e de elevada carga de utilizadores, visto que consegue atingir um maior *throughput* e também uma maior débito ao nível de *storage*.

5.1.2.3 Cenário misto

Para terminar o *benchmark* da aplicação *NextCloud* vamo-nos focar nos resultados do cenário de simulação dos utilizadores com o *Ceph*. À semelhança das análises anteriores, a tabela 5.7 apresenta um resumo dos resultados obtidos para este cenário. Pelos dados obtidos é possível que ver que, tal como neste mesmo cenário com *NFS*, obteve-se um estado intermédio entre os cenários de leituras e de escritas, o que era de esperar.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	3.20	96	0
50 +100%	6.34 +98.13%	110 +14.58%	0.017
75 +200%	9.42 +194.38%	120 +25%	0.012
100 +300%	12.47 +289.69%	120 +25%	0.018

Tabela 5.7: *NextCloud Ceph* - Resumo dos resultados do cenário de simulação do utilizador

Focando-nos numa comparação entre os resultados deste cenário para o *NFS* e *Ceph*, tanto ao nível do *throughput* (gráficos das figuras 5.25 e 5.26), como dos tempos de resposta (tabela 5.8) é possível perceber que o *NextCloud*, para uma utilização aproximada ao que será a de um utilizador real, apresenta uma maior performance utilizando o *Ceph*, ao invés do *NFS*. Isto é possível de ver visto que tanto para os 25 utilizadores como para 100 (e também para 50 e 75, apesar de os gráficos não serem apresentados, visto que a evolução é uniformemente crescente entre cada nível de carga) o *NextCloud* com *Ceph* apresenta um *throughput* superior do que com *NFS*. Para além disso, e relativamente ao tempos de resposta, a tabela mostra claramente que o *NextCloud* apresentou tempos medianos de resposta consideravelmente inferiores com *Ceph*, comparando com os respetivos tempos com *NFS*.

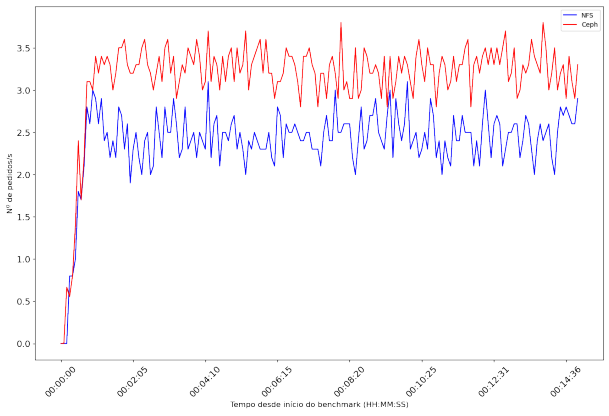

Figura 5.25: Comparação do *throughput* (pedidos/s), *NFS* e *Ceph*, 25 utilizadores

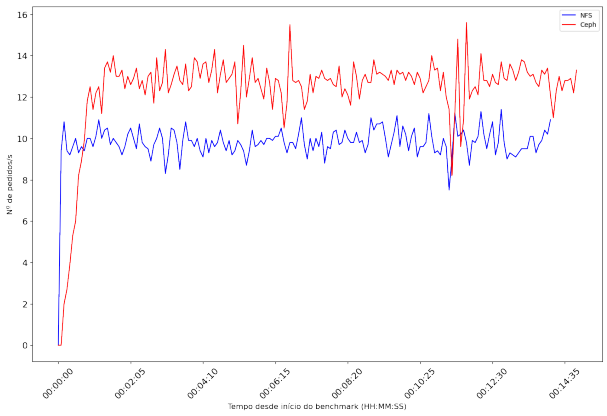

Figura 5.26: Comparação do *throughput* (pedidos/s), *NFS* e *Ceph*, 100 utilizadores

	Tempo mediano de resposta (ms)	
# Utilizadores	NFS	*Ceph*
25	180	96 -46.67%
50	265	110 -58.49%
75	170	120 -29.41%
100	160	120 -25.00%

Tabela 5.8: *NextCloud - Ceph* vs *NFS*- Tempos medianos de Resposta

Por fim, e em jeito de conclusão, foi possível concluir que o *Ceph* é a melhor opção de *storage* (de entre as testadas) para a aplicação *NextCloud*, que apresenta ganhos consideráveis de performance com esta, tanto em cenários de *benchmark* mais "puro" como em cenários mais próximos de uma utilização real do sistema, sendo que os cenários em que isso foi mais evidente foi nos de leitura e simulação do utilizador. No cenário de escritas, apesar do *throughput* ser maior com *Ceph* verifica-se, também, tempos de resposta superior, o que não invalida que o *Ceph* apresente uma melhor performance visto que também é preciso ter em conta que sendo o *Ceph* um mecanismo de *storage* distribuída com replicação aumenta a resiliência do sistema contra falhas ao nível da *storage*. Quanto ao cenário de leituras, foi possível perceber que com o *Ceph*, o *NextCloud* foi capaz de responder à carga submetida, e apresentar um aumento de *throughput*, em percentagem, superior ao aumento, também em percentagem, da carga de utilizadores, tal como se pode rever na tabela 5.4. Finalmente, no cenário de simulação de utilizadores, e para rematar, mais uma vez o *NextCloud* com o *Ceph* apresentou resultados francamente melhores, tanto a nível de *throughput* como de tempos de resposta.

5.2 *Wiki.JS*

5.2.1 *NFS*

5.2.1.1 Cenário de leituras

Relativamente ao cenário de leituras da aplicação *Wiki.JS*, com a *storage NFS*, observando a tabela 5.9, verificamos que em todos os níveis de carga existem erros associados, embora estes representem uma percentagem bastante pequena. Por outro lado, o tempo mediano de resposta é bastante satisfatório, entre 10 e 18 ms. É de salientar que em comparação com a aplicação *NextCloud*, a aplicação *Wiki.JS* apresenta um *throughput* de pedidos por segundo muito superior. Isto deve-se ao facto de a *Wiki.JS* ser utilizada para apresentar páginas *web*, e que para estes testes são caraterizadas por uma variedade de conteúdo, como imagens, vídeos, áudios, etc.. Portanto, ao carregar uma página da *Wiki.JS*, este pedido é acompanhado pelos pedidos do conteúdo, como aconteceria num cenário real.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	53.11	10	0.021
50 +100%	94.76 +78.42%	12 +20%	0.024
75 +200%	139.20 +162.10%	13 +30%	0.021
100 +300%	170.49 +221.01%	18 +80%	0.026

Tabela 5.9: *Wiki.JS NFS*- Resumo dos resultados do cenário de leituras

Observando os gráficos relativos a tempos medianos de resposta (figura 5.27) e pedidos por segundo (figura 5.28), para 25 utilizadores, verificamos que não existem grandes oscilações nestes valores. Visto que o mesmo acontece para os restantes cenários, não esses gráficos não serão expostos aqui, por questões de simplicidade.

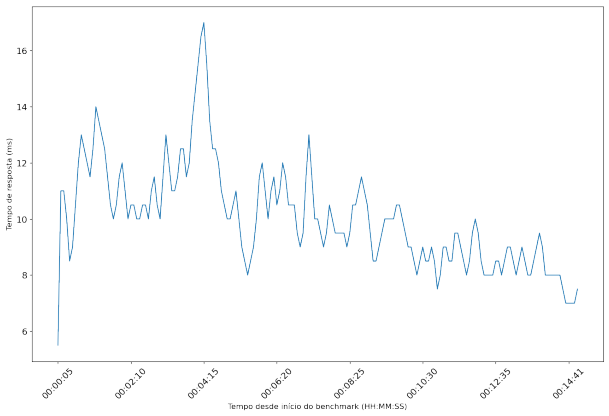

Figura 5.27: Tempos medianos de resposta, 25 utilizadores

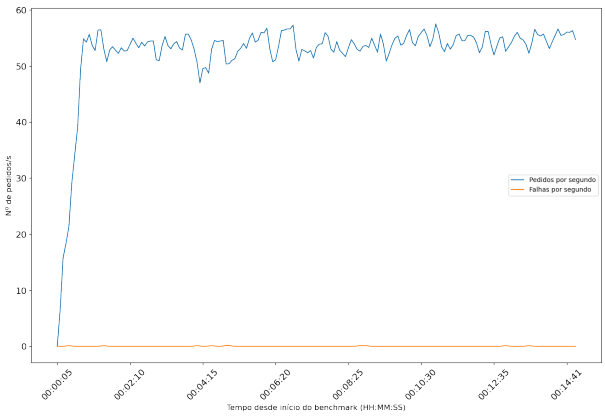

Figura 5.28: *Throughput* (pedidos/s), 25 utilizadores

De modo a perceber melhor quais serão as possíveis limitações no sistema, iremos agora observar as métricas de utilização de CPU e memória no *cluster*, no caso de 25 utilizadores.

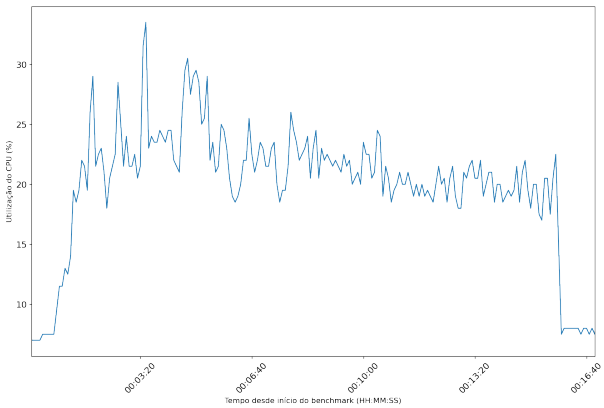

Figura 5.29: Utilização mediana do CPU (%), 25 utilizadores

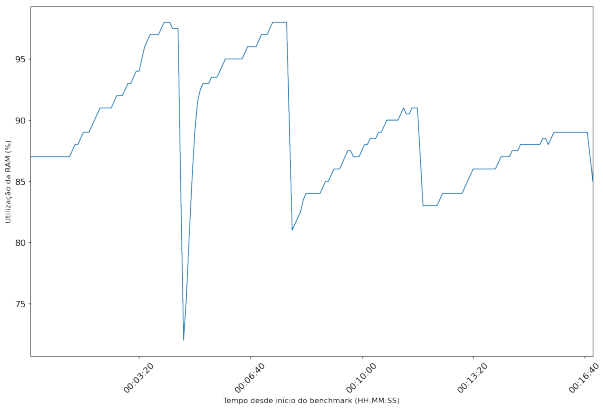

Figura 5.30: Utilização mediana de memória (%), 25 utilizadores

A utilização de CPU, evidenciada na figura 5.29, não ultrapassa os 35%, pelo que não será um *bottleneck* neste caso. No entanto, a utilização de memória, evidenciada na figura 5.30, é problemática. Verifica-se que esta atinge utilização praticamente máxima várias vezes, seguida de quebras na sua utilização. Verificamos portanto que a memória disponível é uma limitação, para a carga de 25 utilizadores a realizar operações de leitura constantes. Como esperado, esta situação piora com o aumento de utilizadores. Verifiquemos as métricas expostas anteriormente, mas para a carga de 50 utilizadores.

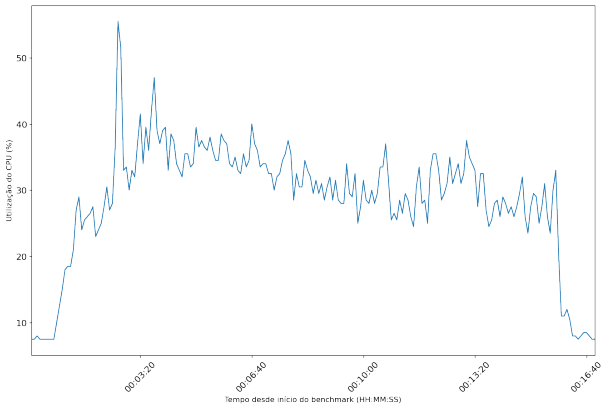

Figura 5.31: Utilização mediana do CPU (%), 50 utilizadores

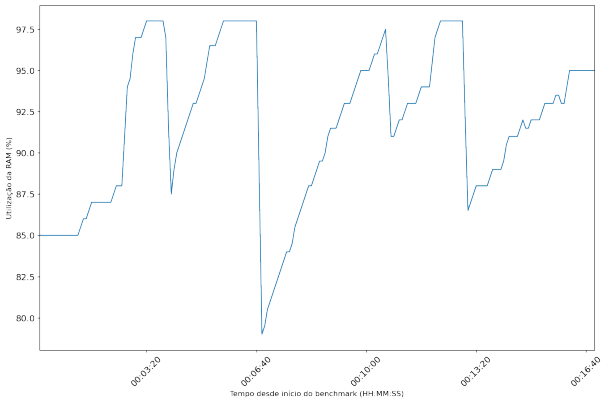

Figura 5.32: Utilização mediana de memória (%), 50 utilizadores

Em termos de utilização de CPU (figura 5.31), estes valores aumentam ligeiramente, mas ainda dentro de valores bastante aceitáveis. Por outro lado, a utilização de memória (figura 5.32) apresentou mais quebras, causadas por atingir os limites de utilização. O mesmo se verifica para os restantes níveis de carga, com 75 e 100 utilizadores, apenas com mais quebras na utilização de memória e ligeiro aumento uniforme na utilização de CPU, atingindo um máximo de 65% no pior caso.

Tendo em conta esta análise, concluímos que para este cenário composto exclusivamente por operações de leituras, o *cluster* foi capaz de lidar com todos os níveis de carga testados, aumentando uniformemente a utilização de CPU e tempo de resposta com o aumento de carga. No entanto, a memória disponível foi uma limitação observada desde o menor nível de carga, traduzindo-se em erros ocasionais nos pedidos.

5.2.1.2 Cenário de escritas

Mais uma vez, temos disponível na tabela 5.10 um resumo dos testes de escritas efetuados para os diferentes níveis de carga, representados pelo número de utilizadores. Observando as diferenças percentuais entre aumento de utilizadores e aumento de *throughput*, verificamos que ao contrário do cenário anterior (referido na subsecção 5.2.1.1), existe uma maior degradação entre a relação destes valores, visto que o aumento de *throughput* não acompanha o aumento de utilizadores, indicando a existência de um *bottleneck* de forma mais percetível. Esta observação é corroborada pela relação desproporcional entre aumento de utilizadores e tempo mediano de resposta, tendo em conta que um aumento de 100% do número de utilizadores (25 para 50), resulta num aumento de 253.66% do tempo mediano de

resposta (82ms para 290ms). Mais uma vez, todos os valores percentuais apresentados nas tabelas são comparações com o nível de carga base de 25 utilizadores.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	13.05	82	0
50 +100%	17.52 +34.25%	290 +253.66%	0
75 +200%	19.95 +52.87%	270 +229.27%	7.552
100 +300%	21.64 +65.82%	560 +582.93%	3.830

Tabela 5.10: *Wiki.JS NFS*- Resumo dos resultados do cenário de escritas

É esperado um tempo mediano de resposta superior neste cenário, visto que as operações de escrita apresentam uma primeira ação que é a criação de uma pasta no sistema de ficheiros virtual na *Wiki.JS*, sendo esta a operação mais demorada dos testes realizados.

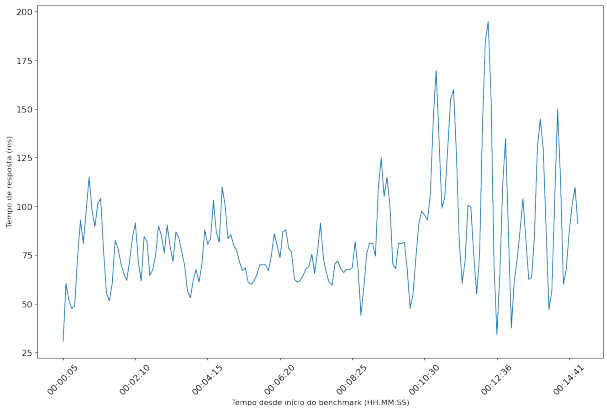

Figura 5.33: Tempos medianos de resposta, 25 utilizadores

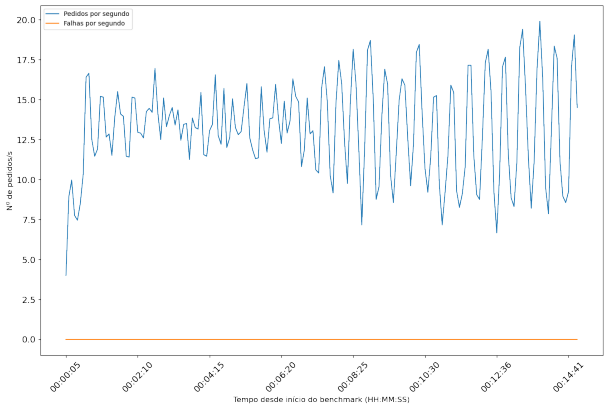

Figura 5.34: *Throughput* (pedidos/s), 25 utilizadores

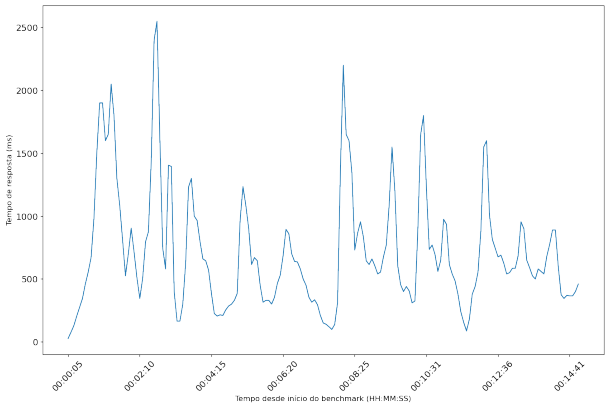

Figura 5.35: Tempos medianos de resposta, 100 utilizadores

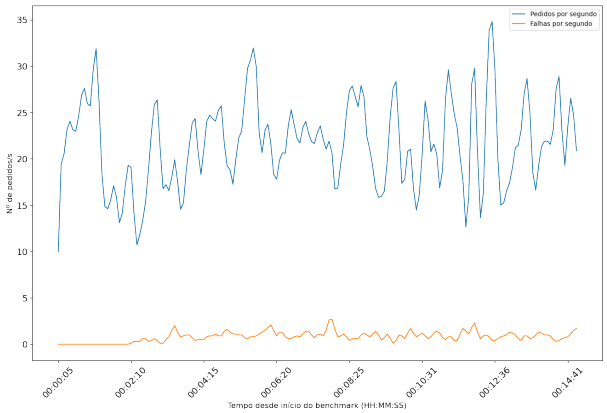

Figura 5.36: *Throughput* (pedidos/s), 100 utilizadores

Observando os gráficos de tempos medianos de resposta (figura 5.33) e *throughput* (figura 5.34), para 25 utilizadores, verifica-se alguma variância destes valores ao longo do teste. No entanto, este

comportamento verifica-se ainda mais para a carga de 100 utilizadores, onde já existe uma percentagem de erros substancial (3.83%), assim como tempos de resposta medianos bastante mais elevados (582.93% superiores em relação à carga de 25 utilizadores).

De forma a perceber qual as causas do aumento do tempo de resposta mediano e a taxa de erro elevada, iremos agora analisar as métricas relativas a CPU e memória para os vários níveis de carga.

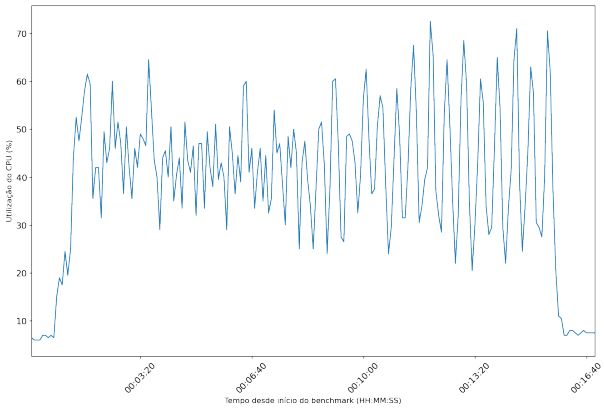

Figura 5.37: Utilização mediana do CPU (%), 25 utilizadores

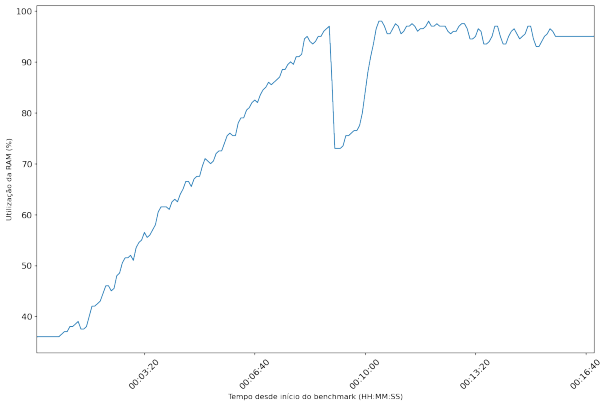

Figura 5.38: Utilização mediana de memória (%), 25 utilizadores

Para o nível de carga de 25 utilizadores, a utilização de CPU (figura 5.37) apresenta picos na ordem dos 70%, consideravelmente elevados tendo em conta o número de utilizadores. Além disso, a utilização de memória (figura 5.38) revela-se como fator limitante, atingindo utilização perto de 100% a meio do teste, sofrendo depois um quebra e estabilizando no máximo novamente. Estes dados apontam para fortes limitações nos restantes testes, hipótese que é corroborada pelos valores apresentados na tabela 5.10.

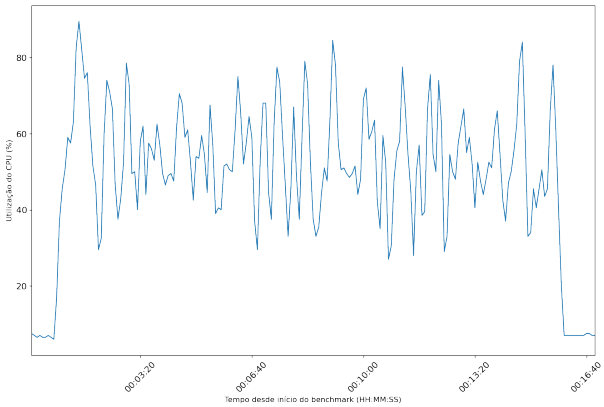

Figura 5.39: Utilização mediana do CPU (%), 50 utilizadores

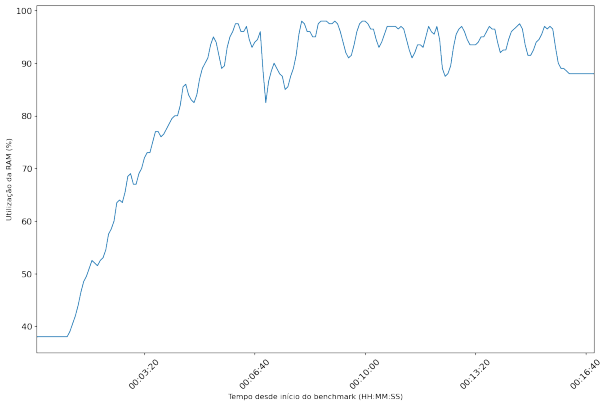

Figura 5.40: Utilização mediana de memória (%), 50 utilizadores

Para o nível de carga de 50 utilizadores, verifica-se que a utilização de CPU (figura 5.39) apresenta já picos já superiores a 80%, pelo que se começa a atingir a exaustão deste recurso. Em relação à utilização de memória (figura 5.40), esta apresenta mais uma vez valores de utilização no limite do disponível, representando portanto um *bottleneck*. Este estado de utilização de recursos traduz-se num aumento

ligeiro de *throughput* (34.24%), mas num aumento considerável do tempo de resposta mediano (253.66%), quando comparado com a carga anterior de 25 utilizadores.

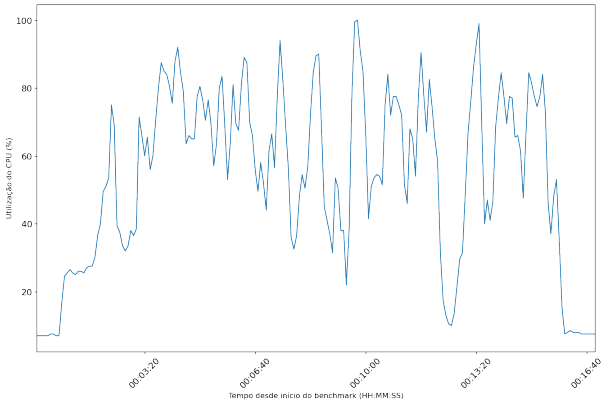

Figura 5.41: Utilização mediana do CPU (%), 75 utilizadores

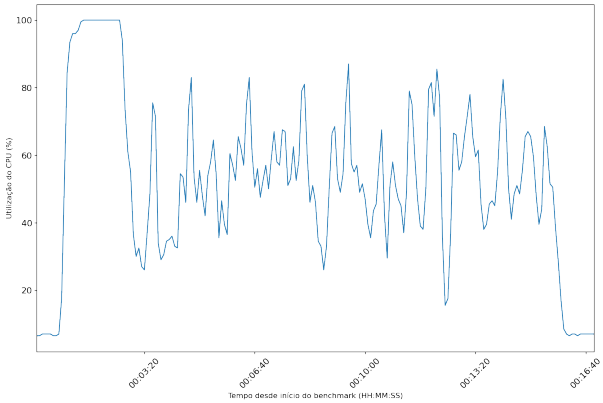

Figura 5.42: Utilização mediana do CPU (%), 100 utilizadores

Como ambos os testes anteriores já apresentavam limitações em termos de utilização de memória, esses gráficos não serão apresentados para os testes restantes (75 e 100 utilizadores), visto que a utilização de memória será similar, ou seja, constantemente nos níveis máximos.

No entanto, os gráficos de utilização de CPU para 75 e 100 utilizadores (figuras 5.41 e 5.42), respetivamente) explicam o motivo para estes testes já apresentarem erros, e no caso de 100 utilizadores, um tempo de resposta mediano ainda mais inflacionado em relação ao nível de carga base (aumento de 582.93%). Observando esses gráficos, verificamos utilizações de CPU a atingir 100 % em vários picos (em 5.41) e até durante um período invulgarmente prolongado (em 5.42).

Neste cenário de escritas, apenas a carga de 25 utilizadores apresentou resultados satisfatórios, já com indícios de falta de memória. Para os restantes níveis de carga, as limitações em termos de memória e CPU resultaram em tempo de resposta medianos superiores, taxa de erro notória ou ambos.

5.2.1.3 Cenário misto

Relativamente ao cenário misto, este é composto por uma combinação de operações de escrita e operações de leitura, sendo a última bastante mais recorrente, simulando a utilização real por parte de um utilizador.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	28.44	13	0
50 +100%	52.15 +83.37%	21 +61.54%	0.070
75 +200%	70.82 +149.02%	25 +92.31%	0.049
100 +300%	70.27 +147.08%	42 +223.08%	0.039

Tabela 5.11: *Wiki.JS NFS*- Resumo dos resultados para simulação de utilizador

Na tabela 5.11 está o resumo deste cenário para os vários níveis de carga, e esta apresenta semelhanças com as tabelas dos cenários anteriores (tabelas 5.9 e 5.10), como seria espectável. Apresenta tempos medianos de resposta baixos e uma pequena taxa de erro associada, tal como no caso do cenário de leituras. No entanto, apresenta a degredação de *performance* da carga de 75 para 100 utilizadores, como acontece no cenário das escritas. Ou seja, duplica o tempo mediano de resposta e mantém o *throughput* praticamente igual.

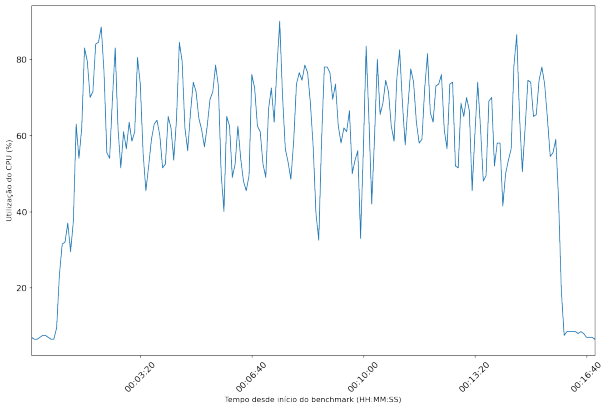

Figura 5.43: Utilização mediana do CPU (%), 100 utilizadores

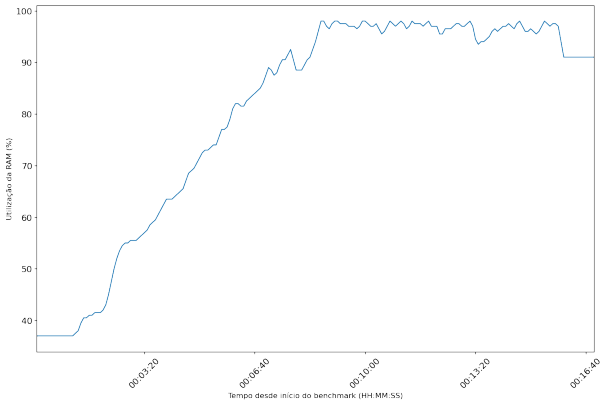

Figura 5.44: Utilização mediana de memória (%), 50 utilizadores

Em termos de utilização de recursos, os níveis de utilização de CPU não atingem o máximo, embora com picos perto dos 85% de utilização, no pior caso (figura 5.43). No entanto, os níveis de utilização de memória atingem o máximo a partir da carga com 50 utilizadores (figura 5.44), sendo apenas a carga de 25 utilizadores a única que não satura este recurso, estabilizando a utilização deste perto de 80% no final do teste.

Visto que este cenário é maioritariamente composto por leituras, é de esperar semelhanças na utilização de recursos entre estes cenários, como por exemplo, nas limitações de memória, o que realmente se verificou. No entanto, as limitações de memória associadas a uma utilização de CPU mais elevada, a rondar os 80%, resultou numa quebra de *performance* entre os níveis de carga de 75 e 100 utilizadores. Nesta transição, o *throughput* manteu-se praticamente constante, assim como a taxa de erro, mas o tempo mediano de resposta duplicou, indicando que já se trata de carga superior à desejada para as condições do *cluster*.

5.2.2 *Ceph*

5.2.2.1 Cenário de leituras

Analisando agora os resultados com a *storage Ceph*, podemos verificar o resumo das várias cargas para este cenário de leituras na tabela 5.12. Tendo em conta o objetivo destes *benchmarks*, iremos comparar este resultados com os resultados da tabela 5.9, correspondente aos resultados para a *storage NFS*.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	42.75	16	0.002
50 +100%	75.67 +77.01%	17 +6.25%	0.004
75 +200%	105.42 +146.60%	18 +12.5%	0.010
100 +300%	138.82 +224.73%	17 +6.25%	0.006

Tabela 5.12: *Wiki.JS Ceph* - Resumo dos resultados do cenário de leituras

Em jeito de exemplificação, comparemos seguidamente os gráficos de tempos medianos de resposta e *throughput* para 25 utilizadores, para ambas as *storages*.

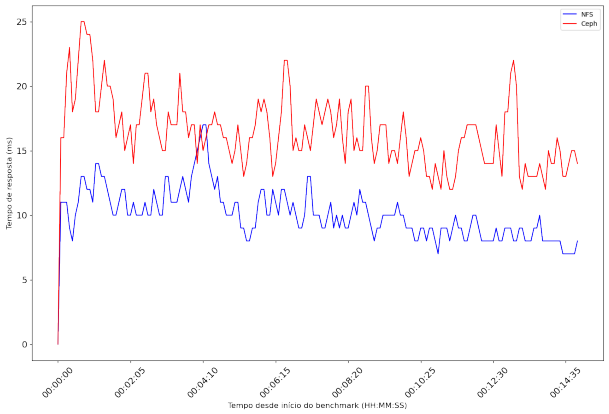

Figura 5.45: Comparação dos tempos medianos de resposta, *NFS* e *Ceph*, 25 utilizadores

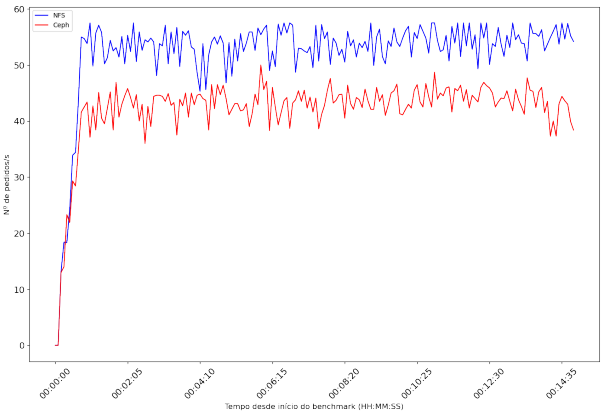

Figura 5.46: Comparação do *Throughput* (pedidos/s), *NFS* e *Ceph*, 25 utilizadores

Comparando os resultados do *Ceph* com *NFS*, nas figuras 5.45 e 5.46, verificamos piores resultados com a *storage Ceph*, aplicando-se esta conclusão também para os restantes níveis de carga, sendo esta diferença de *performance* mais acentuada com o aumento do número de utilizadores. Ao contrário do que acontece na aplicação *NextCloud*, a *Wiki.JS* armazena os ficheiros numa base de dados *PostgreSQL*, e não num sistema de ficheiros tradicional. No entanto, segundo a documentação oficial do *Ceph*, seria de esperar melhores resultados, visto que por parte da base de dados, existem 3 instâncias com capacidade de leitura (um *master* mais dois *slaves*), e o *Ceph* deveria apresentar capacidades de leitura superiores ao *NFS*, ou pelo menos similares. Isto porque a existência de várias instâncias para leitura deveriam beneficiar de *storage* distribuída. Assim sendo, apesar não ter sido possível encontrar uma justificação clara para o ocorrido, é possível concluir que os resultados sem replicação seriam ainda pior.

Em termos de utilização de recursos, a utilização de CPU foi similar neste cenário de leituras para ambas as opções de *storage*, atingindo um valor de utilização máximo de 65%. No entanto, a utilização de memória revelou-se ainda mais crítica com o *Ceph*, com valores de utilização acima de 94% em praticamente todos os níveis de carga.

Para este cenário, embora não seja possível verificar as diferenças em termos velocidades de leitura do disco, por razões mencionada anteriormente, conseguimos concluir que, para as condições testadas,

a *Wiki.JS* obtém melhores resultados com a utilização do *NFS* como *backend* de *storage*.

5.2.2.2 Cenário de escritas

Nesta secção, iremos analisar os resultados do *benchmark* recorrendo ao cenário de escritas, para o *storage Ceph*, e posteriormente fazer comparações com os resultados obtidos com o *storage NFS* (tabela 5.10). Podemos fazer comparações recorrendo à tabela 5.13, que apresenta um resumo dos resultados.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	11.80	83	0.009
50 +100%	16.26 +37.80%	360 +333.73%	0
75 +200%	19.84 +68.14%	520 +526.51%	2.610
100 +300%	17.57 +48.90%	980 +1080.72%	2.441

Tabela 5.13: *Wiki.JS Ceph* - Resumo dos resultados do cenário de escritas

Comparativamente com os resultados com *NFS*, os valores de *throughput* e tempo mediano de resposta foram semelhantes, sendo que os resultados obtidos pelo *Ceph* ligeiramente piores. No entanto, para o caso de 75 utilizadores, embora o *throughput* fosse praticamente igual, o tempo mediano de resposta foi quase duas vezes superior para o *Ceph* (comparando com o valor de 270ms obtido pelo *NFS*). Por fim, o caso de 100 utilizadores foi onde existiram diferenças consideráveis nos resultados, sendo que os valores obtidos de *throughput* foram 18.81% inferiores (comparativamente com 21.64 pedidos por segundo para *NFS*) e o tempo mediano de resposta foram 75% superior (comparativamente com 560ms para *NFS*).

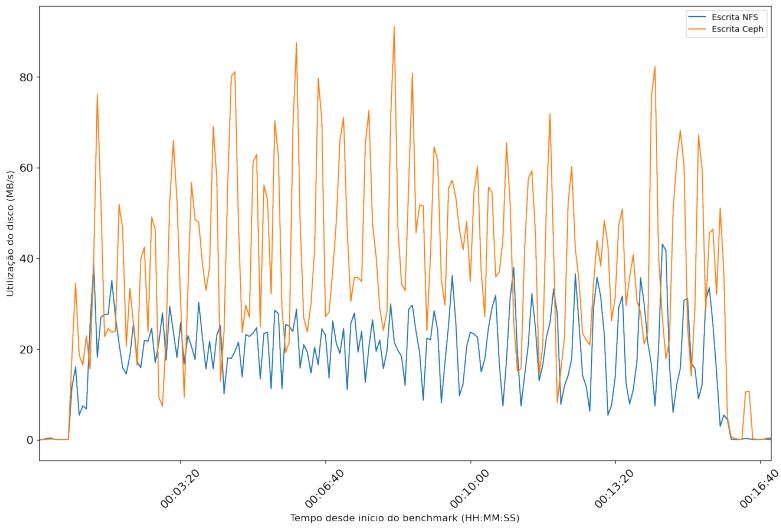

Figura 5.47: Comparação do débito de escritas do clientes de storage, *NFS* e *Ceph*, 25 utilizadores

Contudo, analisando os valores de débito de escritas para ambos os clientes de *storage*, verificamos na figura 5.47 que o *Ceph*, mesmo apresentando piores resultados globais no *benchmark*, apresenta

um débito maior em termos de escritas no disco. No entanto, estes valores de débito de escritas obtidos para o *Ceph* ficam aquém das expetativas, visto que este tem três instâncias implementadas para a distribuição do armazenamento, mas isso não se reflete nos valores médios de débito de escritas, que não são proporcionalmente superiores.

Em termos de utilização de recursos, em comparação com o *NFS*, a utilização de memória foi semelhante, atingindo utilização máxima desde o primeiro nível de carga. Em termos de CPU, o *Ceph* apresentou uma utilização ligeiramente maior. Enquanto o *NFS* apenas apresentou picos de utilização máxima para 75 e 100 utilizadores, o *Ceph* apresentou esses picos adicionalmente para a carga de 50 utilizadores (figura 5.49).

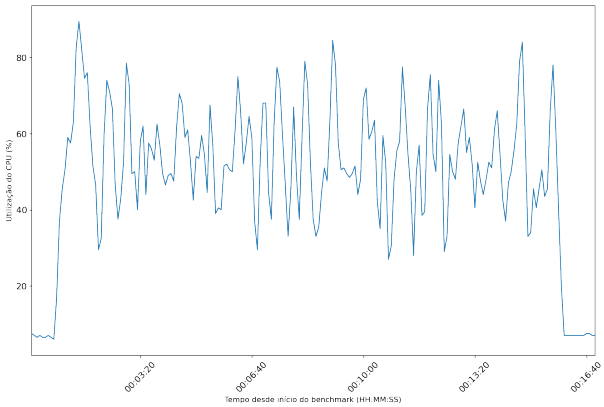

Figura 5.48: Utilização mediana do CPU (%), *NFS*, 50 utilizadores

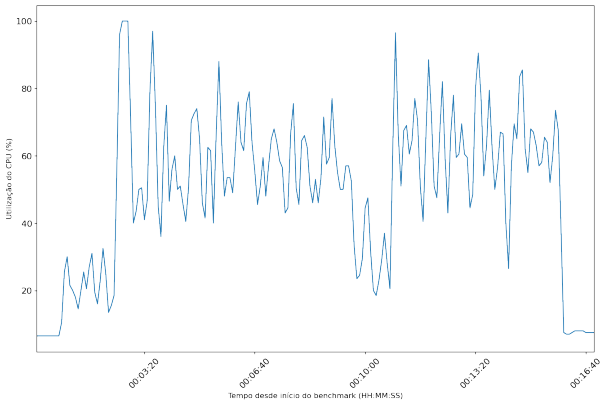

Figura 5.49: Utilização mediana do CPU (%), *Ceph*, 50 utilizadores

Tendo em conta os resultados apresentados, concluímos para operações de escrita, o *storage NFS* apresenta, mais uma vez, melhores resultados. Embora o *Ceph* apresente uma taxa de erro ligeiramente menor para alguns níveis de carga, essa taxa de erro está associada a tempo medianos de resposta bastante superiores e *throughput* inferior.

5.2.2.3 Cenário misto

Por último, para os *benchmarks* na aplicação *Wiki.JS*, iremos analisar os resultados do cenário misto com o *storage Ceph*, e comparar com os resultados com o *storage NFS*. O resumo da execução dos *benchmarks* com *Ceph* está apresentado na tabela 5.14, e o resumo da execução dos *benchmarks* com o *storage NFS* está apresentado na tabela 5.11.

Curiosamente, no caso deste cenário, os resultados para os níveis de carga de 25 e 50 utilizadores são bastante similares, apresentado praticamente os mesmo valores de *throughput* e tempo mediano de resposta, assim como utilização de recursos. Em contraste, para os níveis de carga de 75 e 100 utilizadores, embora os tempos medianos de resposta fossem, mais uma vez, praticamente iguais, os valores de *throughput* foram ligeiramente inferiores no caso do *storage Ceph*, visto que foram obtidos valores próximos de 60 pedidos por segundo, enquanto que no caso do *storage NFS*, foram obtidos valores próximos dos 70 pedidos por segundo. Neste caso, a utilização de CPU foi também similar. Por outro lado,

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	28.07	14	0.007
50 +100%	49.70 +77.06%	20 +42.86%	0.006
75 +200%	60.66 +116.10%	25 +78.57%	1.044
100 +300%	61.14 +117.81%	47 +235.71%	0.069

Tabela 5.14: *Wiki.JS Ceph* - Resumo dos resultados para simulação de utilizador

a utilização do *storage Ceph* atingiu o limite de memória um pouco mais cedo (observável nas figuras 5.50 e 5.51), podendo ser essa a causa desta diferença de resultados.

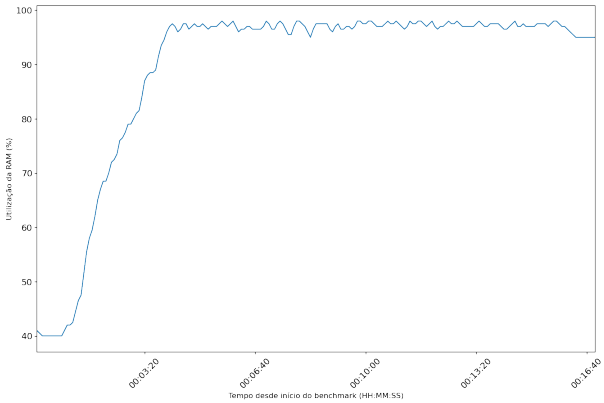

Figura 5.50: Utilização mediana de memória (%), *NFS*, 75 utilizadores

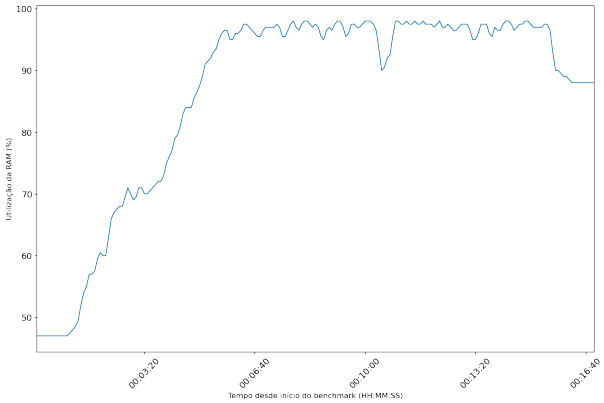

Figura 5.51: Utilização mediana de memória (%), *Ceph*, 75 utilizadores

Tendo em conta esta análise, para um cenário composto tanto por operações de escrita como de operações de leitura, as diferenças em termos de *performance* acabam por ser bastante menores do que foi verificado num cenário com apenas leituras ou apenas escritas. Desta forma, talvez seja benéfica a utilização do *Ceph* como cliente de *storage*, visto que oferece mais resiliência em termos de armazenamento, sacrificando marginalmente o *throughput* para cargas maiores de utilizadores, comparativamente com *NFS*. Contudo, se considerarmos puramente a performance do sistema, o *NFS* revelou-se uma opção claramente melhor para a *Wiki.JS* do que para o *NextCloud*.

5.3 *PeerTube*

5.3.1 *NFS*

5.3.1.1 Cenário de leituras

Após observar os resultados obtidos e alguma pesquisa ao modo de como a aplicação transfere ficheiros, chegamos à conclusão que os resultados obtidos para o cenário de leitura, que nosso caso corresponde ao *download* de ficheiros, são inválidos. Isto deve-se ao facto de a aplicação confirmar o

sucesso do pedido do *download* antes de efetivamente efetuar o download. Como tal, a aplicação de *benchmarking* ao receber a resposta positiva do pedido encerra o canal de comunicação, nunca efetuando o *download*.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	2.5	7	0.0

Tabela 5.15: *PeerTube NFS* - Resumo dos resultados para cenário de leituras

5.3.1.2 Cenário de escritas

Em relação ao cenário de escritas, que neste caso corresponde ao *upload* de vídeos, os resultados obtidos, tal como no cenário de leituras são inconclusivos. Ao contrário da situação anterior, conseguimos efetuar alguns pedidos com sucesso, mas independentemente do número de utilizadores obtivemos uma taxa de erro de 50%. Esta percentagem elevada de erros deve-se, essencialmente, a erros internos do servidor aplicacional derivado de não conseguir lidar com a carga (bastante baixa) a que foi submetido.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	1.26	328	49.74

Tabela 5.16: *PeerTube NFS* - Resumo dos resultados para cenário de escritas

5.3.1.3 Cenário misto

Neste cenário, tal como é de esperar, temos uma conjugação dos problemas anteriores, mantendo a taxa de erro dos pedidos de escrita apesar de estes diminuírem em número.

# Utilizadores	*Throughput* (pedidos/s)	Tempo mediano de resposta (ms)	Erro (%)
25	1.49	8	7.46

Tabela 5.17: *PeerTube NFS* - Resumo dos resultados para cenário de simulação do utilizador

Devido à elevada taxa de erro no cenário de escritas que podemos observar na tabela 5.16, e aos resultados não conclusivos do cenário de leituras, concluímos que não faria sentido efetuar testes com um número maior de utilizadores.

5.3.2 *Ceph*

Ao efetuarmos os testes com este *backend* de armazenamento obtivemos os mesmos problemas, confirmando que é um problema interno da aplicação em causa. Mesmo no cenário de escrita, obtivemos resultados muito semelhantes no que toca às taxas de erro que tínhamos obtido com o *NFS*. Corroborando, assim, que neste caso o fator impeditivo é a aplicação e não o armazenamento.

Capítulo 6

Discussão

O objetivo deste capítulo é apresentar algumas conclusões genéricas que foram possíveis de retirar ao longo da execução do trabalho, bem como um resumo do que foi possível concluir durante a análise de resultados, relativamente a cada uma das aplicações.

Relativamente ao esforço associado à instalação e utilização de *storage* em *Kubernetes*, o *NFS* tem claramente vantagem sobre o *Ceph*, tanto pela facilidade de instalação da estrutura necessária, que está relacionada à menor complexidade inerente a este mecanismo de *storage*, como pela facilidade de utilizar em *Kubernetes* através de aprovisionamento dinâmico de volumes. Tal como foi dito no capítulo 2, a forma mais útil de utilizar *storage* em *Kubernetes* é através de *Storage Classes* visto que estas permitem criar PVs[1] e PVCs[2] dinamicamente e conforme necessário, o que permite maximizar os recursos disponíveis e utilizá-los de uma forma mais eficiente. No caso do *NFS* isto foi fácil de fazer, visto que existem *provisioners online* que funcionam sem quaisquer problemas. No caso do *Ceph*, esta situação já não se verifica, visto que a maioria dos *provisioners* disponíveis *online* já são antigos e não funcionam com as versões atuais do *Kubernetes* e *Ceph*, pelo que foi necessário criar os PVs e PVCs manualmente, algo não muito aconselhável no paradigma de *Kubernetes*. É importante referir que esta situação aplica-se especialmente ao tipo de *storage* do *Ceph* "*Filesystem*", já que no tipo de "*Block Storage*" existem *provisioners* possíveis de utilizar, contudo as aplicações testadas, tal como muitas outras no mercado, precisam de *storage* "*Filesystem*" e não "*Block Storage*" (pois é necessário fazer a sua montagem com a propriedade de "*ReadWriteMany*"). No entanto, esta dificuldade não invalida, de todo, a aplicação de *Ceph* como *backend* de *storage*; contudo, é algo a ter em conta neste tipo de contexto.

Ainda relativamente ao *Ceph*, durante a fase de otimização das configurações do mesmo de forma a extrair a melhor performance possível, o grupo apercebeu-se que este estava a ter débitos muito inferiores ao que se esperava. Acabámos por perceber que a razão disto acontecer tinha a ver com o facto de

[1] Persistent Volumes
[2] Persistent Volume Claims

algumas das máquinas pertencentes ao *cluster Kubernetes*, máquinas essas onde os volumes do *Ceph* estavam a ser montados (para serem utilizadores pelos *pods*), estarem a utilizar a *driver* baseada em *FUSE*, o que afetava gravemente a *performance* da *storage*. A partir do momento em que se instalou a *kernel driver* nas máquinas, e consequentemente os *mounts* passaram a ser feitos com a mesma, os débitos aumentaram para, pelo menos, o dobro. Assim, conclui-se que, exceto em situações em que a utilização da *driver FUSE* é intencional, é útil garantir que os volumes do *Ceph* estão a ser montados com a *kernel driver* em vez da *driver FUSE*, devido aos claros ganhos de performance.

Relativamente aos resultados relativos aos *benchmarks* das aplicações, e começando pelo *PeerTube*, foi a aplicação que menos ilações permitiu retirar, visto que os seus resultados foram maioritariamente inconclusivos ao que diz respeito à *storage*, muito por causa das dificuldades em alcançar uma replicação consistente da mesma e das suas limitações internas, tal como referido ao longo do relatório.

Considerando a análise dos resultados relativos à aplicação *NextCloud*, a primeira conclusão possível de retirar foi que esta não apresentou quaisquer constrangimentos de relevo, ao nível de recursos do sistema (CPU, e memoria). Relativamente ao *NFS* no *NextCloud*, este apresentou resultados bastante aceitáveis, sem taxas de erro significativas e conseguiu suportar a carga a que foi submetido sem grandes quebras de performance, à exceção do cenário de escritas; contudo este quebra também se verificou com o *Ceph*. Relativamente ao *Ceph*, este apresentou melhorias de performance relativamente ao *NFS* em todos os cenários, sendo que no cenário misto (o mais relevante para a análise da aplicação em si, visto que se aproxima mais com a utilizarão real da aplicação) se notaram melhorias significativas tanto no *throughput* como nos tempos de resposta, como aliás é possível ver na figura 5.26 e tabela 5.8, respetivamente. Quanto ao cenário de leitura o aumento de performance foi também muito significativo visto que, com o *Ceph*, o *NextCloud* mostrou uma capacidade de evolução de *throughput* superior à instalação com *NFS*, mantendo a escala dos tempos de resposta, como é possível ver comparando as percentagens de incremento presentes nas tabelas 5.1 e 5.4. Outro elemento onde esta vantagem é bastante clara é nas figuras 5.18 e 5.20, que compara o *throughput* das duas aplicações neste cenário, para 25 e 100 utilizadores. Relativamente ao cenário de escritas, é possível de ver que o *Ceph* permite maior performance, principalmente olhando para as figuras 5.23 e 5.24, onde se comparam os débitos de escritas das duas *storages* e onde é claro que o *Ceph* disponibiliza um débito de escritas largamente superior ao do *NFS*. Por fim, ainda no início da instalação do *NextCloud* foi possível perceber que, pelo menos aparentemente, o *Ceph* permitia melhor performance visto que a aplicação, aquando da instalação, tem um processo inicial de *rsync* para pré-carregar a mesma com alguns documentos e este processo demorava um terço do tempo com *Ceph* (cerca de 5 minutos), comparativamente ao *NFS*, que demorava cerca de 15 a 20 minutos.

Para a aplicação *Wiki.JS*, foi possível concluir que independentemente do *backend* de *storage* utilizado, existiram limitações de CPU, sendo estas mais críticas em operações de escritas para cargas superiores a 50 utilizadores, e também limitações de memória, sendo estas aparentes em todos os cenários, para praticamente todos os níveis de carga. As percentagens de erro foram mais relacionadas com limitações de CPU, visto que foram nestes casos que estes valores de erros subiram consideravelmente. Os erros

relacionados com o esgotamento de memória foram sensivelmente duas ordens de magnitude abaixo dos erros relacionados com a sobre-carga de CPU (observável, por exemplo, para *NFS*, na Tabela 5.9 para leituras, cenário que não atinge o limite de CPU, e na Tabela 5.10 para escritas, cenário que atinge o limite de CPU). Em termos de *backend* de *storage* preferível, esta escolha não é tão imediata como no caso da aplicação *NextCloud*. Para os cenários testados, o *NFS* apresentou resultados melhores em todos os casos, embora por margens pequenas no cenário de simulação do utilizador (evidenciado na Tabela 5.11 para *NFS* e na Tabela 5.14 para *Ceph*). Por essa razão, poderá optar-se pelo *Ceph*, com um *tradeoff* de *performance* ligeiro pela resiliência oferecida pelo *Ceph* em relação ao *NFS*. Estes resultados aplicam-se para cenários semelhantes ao testado no cenário de simulação de utilizador, onde existe a relação de uma escrita para cinco leituras. Caso as operações de leitura sejam bastante superiores a este cenário descrito, a diferença de *performance* poderá começar a ser mais aparente. No entanto, caso a *performance* seja prioridade e não a resiliência dos dados, o *NFS* será a melhor opção.

Capítulo 7

Conclusão

Durante a elaboração do presente trabalho fizemos uma análise de performance a três aplicações, utilizando dois tipos diferentes de *backends* de *storage*. No capítulo anterior retiramos conclusões sobre as aplicações individualmente, sendo que podemos resumir essas conclusões do seguinte modo: na maioria dos cenários reais, é expectável (e necessário) existir a resiliência dos dados a guardar. Assim sendo, é necessário utilizar um *backend* de *storage* que ofereça boa resiliência. Assim sendo, consideramos que o *Ceph* é uma boa escolha, pois a *trade-off* entre a resiliência e a *performance* é, num caso geral, negligível, sendo que, em alguns casos, existem ganhos de *performance*. Assim sendo, e considerando os ganhos de resiliência, o *Ceph* torna-se numa escolha atrativa para estes fins. No entanto, se o único objetivo for performance, poderá ser necessária uma análise caso a caso, para entender quais os potenciais *bottlenecks* das aplicações utilizadas, e assim se concluir qual será o *backend* com a melhor *performance*.

Os testes que executámos não são, claramente, exaustivos. Assim, sugerimos como potencial trabalho futuro, a análise de outros cenários de teste, como por exemplo, executar várias aplicações no *cluster Kubernetes*, que é um cenário que não executámos devido à falta de recursos nas máquinas que nos foram disponibilizadas. Além disso, poderiam ser analisados cenários de teste onde utilizávamos diferentes configurações do *Ceph*, alterando, por exemplo, o nível de replicação, para tentar encontrar um nível de *trade-off* ótimo entre *performance* e resiliência. Poderia também ser analisado um cenário, para cada aplicação, onde as caches internas das aplicações estariam desativadas, o que causaria, em princípio, um maior *stress* nos *backends* de *storage*.

Neste trabalho também nos foi impossível obter os valores de débitos das leituras. Assim sendo, sugerimos que, num trabalho futuro, sejam utilizadas e configuradas ferramentas para obter estes valores, para poderem ser comparados.

Além disto, sugerimos a utilização do *Rook.io*[1] para a instalação dos *backends* de *storage*, pois tornam mais fácil a instalação e configuração dos mesmos num *cluster Kubernetes*.

[1] https://rook.io

Bibliografia

[1] *Volumes - Kubernetes*, https://kubernetes.io/docs/concepts/storage/volumes/#volume-types, Acedido: 01-06-2021.

[2] *Playbooks kubespray - configuração automática de um cluster Kubernetes*, https://github.com/kubernetes-sigs/kubespray, Acedido: 23-05-2021.

[3] *Helm - Docs*, "https://helm.sh/docs", Acedido: 19-05-2021.

[4] *GitHub - Nextcloud - Helm Chart*, "https://github.com/nextcloud/helm/tree/master/charts/nextcloud", Acedido: 15-05-2021.

[5] *Managing Resources for Containers - Kubernetes*, "https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu", Acedido: 19-05-2021.

Apêndice A

Exemplo da configuração do *Deployment* do *Nextcloud*

Ficheiro *helm/nextcloud.yml*

```
image:
  repository: nextcloud
  tag: 20.0.9-apache
  pullPolicy: IfNotPresent

nameOverride: ""
fullnameOverride: ""

replicaCount: 1

nextcloud:
  host: cloud74
  username: admin
  password: admin
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0

internalDatabase:
  enabled: false

mariadb:
  enabled: true
  db:
    name: nextcloud
    user: nextcloud
    password: passw0rd
```

```
30    replication:
31      enabled: true
32    master:
33      persistence:
34        enabled: true
35        storageClass: nfs-client
36        accessMode: ReadWriteMany
37        size: 10Gi
38    slave:
39      replicas: 2
40      persistence:
41        enabled: false
42
43  redis:
44    enabled: true
45    usePassword: true
46    password: passw0rd
47    master:
48      persistence:
49        enabled: true
50        storageClass: nfs-client
51        accessMode: ReadWriteMany
52        size: 8Gi
53    slave:
54      replicas: 2
55      persistence:
56        enabled: false
57
58  service:
59    type: NodePort
60    port: 8080
61    nodePort: 30000
62
63  persistence:
64    enabled: true
65    storageClass: nfs-client
66    accessMode: ReadWriteMany
67    size: 25Gi
68
69  resources:
70    requests:
71      cpu: 350m
72
73  livenessProbe:
74    enabled: false
75    initialDelaySeconds: 10
76    periodSeconds: 10
77    timeoutSeconds: 5
78    failureThreshold: 3
79    successThreshold: 1
80  readinessProbe:
81    enabled: true
82    initialDelaySeconds: 10
```

```
83      periodSeconds: 30
84      timeoutSeconds: 15
85      failureThreshold: 3
86      successThreshold: 1
87    startupProbe:
88      enabled: false
89      initialDelaySeconds: 30
90      periodSeconds: 10
91      timeoutSeconds: 5
92      failureThreshold: 30
93      successThreshold: 1
94
95    hpa:
96      enabled: true
97      cputhreshold: 55
98      minPods: 1
99      maxPods: 15
```

Apêndice B

Exemplo da configuração do *Deployment* da *Wiki.js*

Ficheiro *helm/wiki.yml*

```
image:
  repository: requarks/wiki
  pullPolicy: IfNotPresent

nameOverride: ""
fullnameOverride: ""

replicaCount: 1

serviceAccount:
  create: false

service:
  type: NodePort
  port: 3000
  nodePort: 30002

postgresql:
  enabled: true
  fullnameOverride: "wiki-postgresql"
  postgresqlUsername: wiki
  postgresqlDatabase: wiki
  postgresqlPassword: wiki
  replication:
    enabled: true
    readReplicas: 2
  readReplicas:
    persistence:
      enabled: false
      size: 30Gi
  persistence:
    enabled: true
```

```
33      storageClass: nfs-client
34      accessMode: ReadWriteMany
35      size: 30Gi
36
37  resources:
38    requests:
39      cpu: 350m
40
41  hpa:
42    enabled: true
43    cputhreshold: 55
44    minPods: 1
45    maxPods: 15
```

Apêndice C

Exemplo da configuração do *Deployment* da *Peertube*

Ficheiro *helm/peertube.yml*

```
1   replicaCount: 1
2
3   image:
4       repository: chocobozzz/peertube
5       pullPolicy: IfNotPresent
6       tag: ""
7
8   imagePullSecrets: []
9   nameOverride: ""
10  fullnameOverride: ""
11
12  serviceAccount:
13      create: false
14
15  service:
16      type: NodePort
17      port: 9000
18      nodePort: 30001
19
20  resources:
21    requests:
22      cpu: 1000m
23
24  autoscaling:
25      enabled: false
26
27  persistence:
28      enabled: true
29      storageClass: nfs-client
```

```
30      accessMode: ReadWriteMany
31      size: 25Gi
32
33  postgresql:
34      enabled: true
35      postgresqlUsername: postgres
36      postgresqlPassword: postgres
37      postgresqlDatabase: peertube_prod
38      replication:
39          enabled: true
40          readReplicas: 2
41      readReplicas:
42          persistence:
43              enabled: false
44              size: 10Gi
45      persistence:
46          enabled: true
47          storageClass: nfs-client
48          accessMode: ReadWriteMany
49          size: 10Gi
50
51  redis:
52    enabled: true
53    usePassword: false
54    master:
55      persistence:
56        enabled: true
57        storageClass: nfs-client
58        accessMode: ReadWriteMany
59        size: 5Gi
60    slave:
61      replicas: 2
62      persistence:
63        enabled: false
64
65  postfix:
66      enabled: false
```