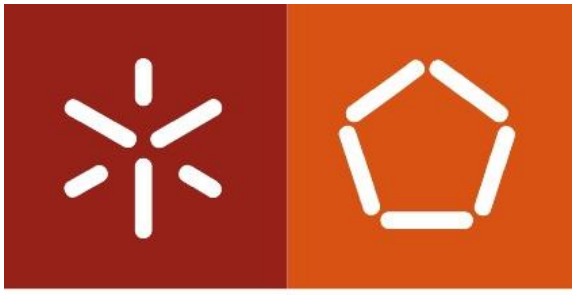

Aplicações e Sistemas de Computação em Nuvem

Eduardo Coelho, Henrique Neto, Maria João Moreira, Carlos Preto, Pedro Veloso
{pg47164, pg47238, a89540, pg47089, pg47578}alunos.uminho.pt

Universidade do Minho
Mestrado em Engenharia Informática

11 de janeiro de 2022

Índice

1 Introdução

Este trabalho prático tem como objetivo principal a caraterização, análise, instalação, monitorização e avaliação da aplicação *Wiki.js*, utilizando para o efeito os conteúdos lecionados na unidade curricular Arquitetura de Sistemas de Computação em Nuvem.

Na primeira parte do projeto, iremos discutir a arquitetura e padrões de distribuição utilizados na aplicação. Para além disso, iremos analisar cada um dos seus componentes, com o objetivo de identificar possíveis pontos críticos da aplicação e eventuais soluções.

Na segunda parte iremos explicar como utilizámos a ferramenta Ansible para permitir a instalação automática e configurável dos diversos componentes da aplicação num ambiente de computação em nuvem, nomeadamente, na Google Cloud Platform.

A terceira tarefa consiste em dotar a aplicação de ferramentas de monitorização que permitam observar a aplicação num ambiente de testes e/ou produção.

Na quarta e última tarefa, iremos utilizar ferramentas de avaliação que nos permitam avaliar a aplicação de forma realista e com múltiplos testes que simulem diferentes comportamentos de utilizadores.

2 Arquitetura e Componentes da aplicação

2.1 Arquitetura

A aplicação *Wiki.js* segue uma arquitetura multi-camada, na medida em que os seus componentes estão distribuídos por camadas distintas. Os dois componentes principais da aplicação são a base de dados e o servidor *Wiki.js*. Para além disso, opcionalmente, e para redes mais complexas e avançadas, pode-se utilizar um *reverse proxy* como, por exemplo, o *NginX* e o *Apache*. Esta arquitetura *proxy-server* permite que os pedidos realizados pelos utilizadores possam ser processados pelo servidor *Wiki.js*, sem nunca haver ligação direta entre os mesmos. Assim, o componente de *reverse-proxy* permite "esconder"o servidor dos clientes, aumentando a segurança da aplicação. Quanto à base de dados, a aplicação dá a liberdade do utilizador escolher o sistema de gestão de bases de dados que pretende utilizar como, por exemplo, *postgres*, *mysql*, *mariadb*, entre outros.

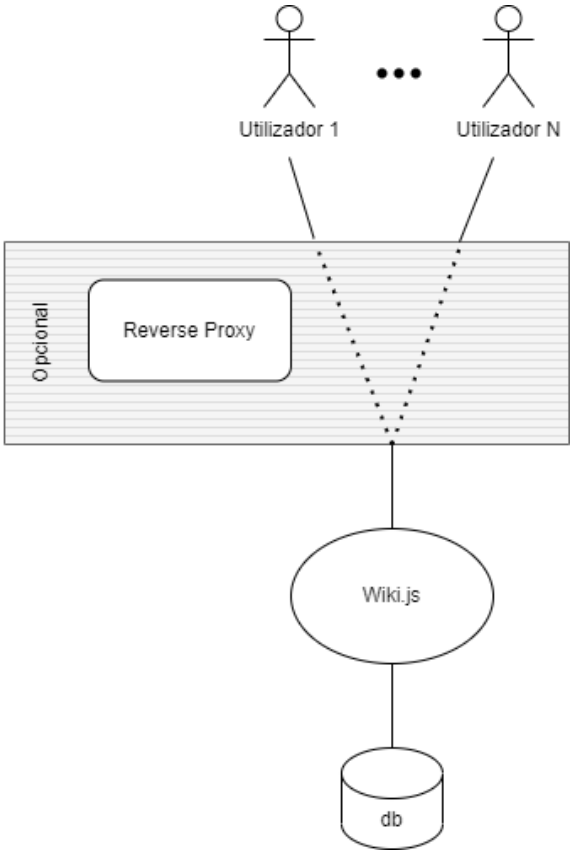

Figura 2.1: Arquitetura da aplicação

2.2 Componentes e pontos críticos da Aplicação

Analisando a arquitetura da aplicação, é facilmente percetível que cada um dos componentes desta pode ser visto como um ponto crítico, dependendo das operações realizadas e do contexto.

2.2.1 *Reverse-proxy*

O *reverse-proxy* funciona como intermediário entre os utilizadores e o servidor, controlando assim os acessos ao *Wiki.js*, o que irá contribuir para o aumento da segurança da aplicação. Porém, no caso de haver um único *reverse-proxy*, quando houver vários utilizadores a realizar pedidos à aplicação, levará a uma sobrecarga deste, visto que terá de tratar cada um desses pedidos, antes de os reencaminhar para o servidor. Tal problema pode ser apaziguado através da utilização de vários *reverse-proxies*.

2.2.2 *Wiki.js*

O *Wiki.js* funciona como o servidor da aplicação. Como tal, o objetivo deste componente é processar os pedidos recebidos dos clientes (ou do *proxy*) e aceder à base de dados caso seja necessário. Deste modo, tendo um único servidor a processar todos os pedidos, é natural que este fique sobrecarregado, causando um *bottleneck* no desempenho da aplicação. Uma solução possível para evitar este problema é a utilização de múltiplos servidores, distribuindo os pedidos feitos pelos clientes e/ou *proxies* pelos mesmos.

A Base de Dados é responsável por armazenar os dados e providenciar a informação requisitada pelo servidor. No caso de haver uma única Base de Dados, quantos mais pedidos o servidor enviar a esta, mais lento será o processo de providenciar a informação pretendida, levando assim a um congestionamento na aplicação. A solução para tal problema será criar mais que uma base de dados.

3 Instalação da aplicação

Tal como se referiu na secção anterior, de modo a garantir o funcionamento da aplicação é necessário executá-la, associada a um motor de base de dados compatível com a aplicação – *MySQL*, *MariaDB*, *MS SQL*, *SQLite* ou *PostgreSQL*. Ao garantir o acesso a estes dois componentes, é possível usufruir da totalidade das funcionalidades oferecidas pela aplicação, contudo poderá ocorrer problemas de performance, à mediada que o número de utilizadores aumente.

Com o aumento dos utilizadores da aplicação, é expectável que a performance da aplicação comece a degradar numa instalação simples do *Wiki.js*. Assim, de modo a conservar a qualidade e viabilidade do serviço poderá ser necessário melhorar a instalação da aplicação e da base de dados da qual depende. Estas melhorias variam conforme o volume de utilizadores em questão, podendo estas variar entre soluções simples como ajustes de configurações e soluções mais complexas como por exemplo replicação de componentes da aplicação e da base de dados.

Ao replicar as funcionalidades da aplicação para diferentes instâncias é necessário garantir que os utilizadores se distribuam de forma a possibilitar a uma boa qualidade de serviço de forma diminuir carga atual sobre cada instância. Para fazer esta distribuição, recorreu-se à utilização de um *reverse proxy*. O *reverse proxy* é colocado antes das instâncias aplicacionais e distribui as sessões dos utilizadores pelas várias instâncias da aplicação. Desta forma, o ponto de acesso ao serviço passa a ser este componente. O *reverse proxy* usado no projeto foi o *NGINX* devido á sua popularidade e ás recomendações de instalação da equipa do *WikiJS*.

Adicionalmente, esta replicação de funcionalidades aplicacionais pode por sua vez causar pontos de estrangulamento ou falhas em outros componentes já existentes, nomeadamente a base de dados da aplicação. Desta forma é necessário melhorar a instalação deste componente, na qual se optou pela replicação. Foi implementada uma replicação *master-slave*, disponível pelo próprio sistema de gestão *mysql*, que consiste em um conjunto de base de dados secundárias (*slaves*) que atualizam os seus dados de acordo com a base de dados principal (*master*). A utilização deste tipo de replicação possibilita distribuir os pedidos de leitura pelos vários servidores de bases de dados distintos. Contudo, existe apenas um servidor para realizar pedido de escrita, mantendo desta forma a existência de um ponto de falha. Ao utilizar esta abordagem para a replicação da base de dados, foi necessário também instalar um *proxy* para as base de dados cuja função principal é realizar um *read-write split*, ou seja, reencaminhar para os servidores *slaves* os pedidos de leitura e para o servidor *master* os pedidos de escrita. Sendo assim, as instâncias aplicacionais passam a encaminhar as suas *queries* para este *proxy*, o que agrava o ponto de falha referido anteriormente.

Por fim, com as novas alterações á instalação é possível satisfazer um volume muito maior de utilizadores com o serviço do *Wiki.js*, através de um número ar-

bitrário de instâncias, que podem estar implementadas sobre um número arbitrário de servidores.

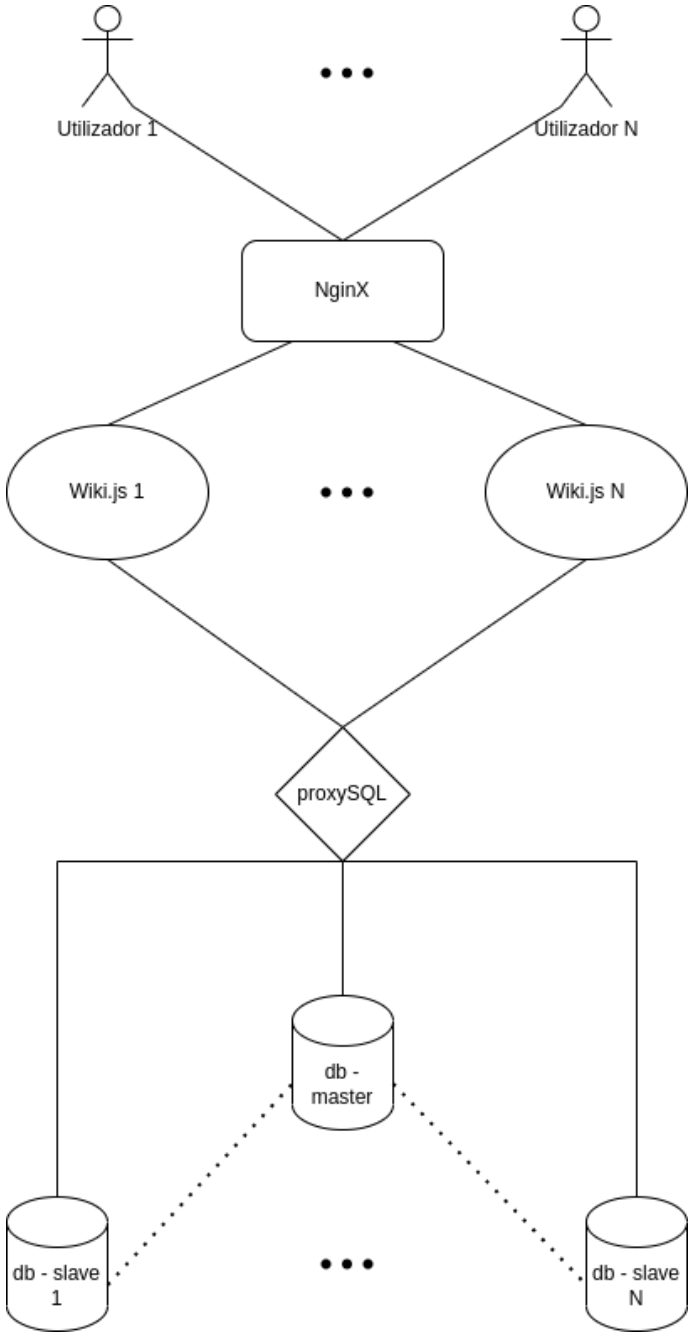

Figura 3.1: Esquema de implementação novo

De modo a garantir a instalação automática da aplicação e dos seus respetivos complementos, recorreu-se à ferramenta *ansible*. Esta ferramenta, permite expressar os diferentes servidores que vão contemplar cada componente, bem como os diferentes passos a executar para as instalar (através de roles). Posteriormente reúne toda a informação num *playbook* a ser executado.

3.1 Docker

Uma vez que existem vários componentes necessários para o funcionamento da aplicação implementados sobre *docker containers*, foi necessário instalar o *docker* nos servidores necessários. Assim, foi criado um *role* em *ansible* responsável por executar as instalações e configurações necessárias.

```
- name: Add Docker GPG apt Key
  apt_key:
    url: https://download.docker.com/linux/ubuntu/gpg
    state: present

- name: Add Docker Repository
  apt_repository:
    repo: deb https://download.docker.com/linux/ubuntu bionic stable
    state: present

- name: Update apt and install docker-ce
  apt: update_cache=yes name=docker-ce state=latest
```

Figura 3.2: Excerto do role *docker*

3.2 Reverse Proxy – NGINX

O *reverse proxy* do *NGINX* disponibiliza uma imagem official *docker*. Desta forma, para instalar este componente é simplesmente necessário montar o ficheiro de configuração do programa ao *docker container*. Este ficheiro de configurações é por sua vez gerado a partir de uma *template* pelo *ansible*, e configura no serviço um servidor *http* que redireciona as sessões para a instância do serviço com menor conexões através de um conceito da aplicação denominado de *upstream*. Esta gestão poder ser modificada a partir das variavéis do *ansible*, podendo esta variar antes conforme o tempo de acesso á instância, fila ou endereço do utilizador.

```
- name: Create NGIN X config file
  template:
    src: "default.conf"
    dest: "./default.conf"

- name: Run NGIN X
  docker_container:
    name: nginx
    image: nginx
    state: started
    restart_policy: unless-stopped
    networks:
      - name: "{{ net_work_docker_name }}"
    volumes:
      - ./default.conf:/etc/nginx/nginx.conf
    ports:
      - "{{ http_port }}:{{ ngix_port }}"
      - "{{ app_port}}:{{ app_port }}"
```

Figura 3.3: Excerto do role *proxy*

3.3 Aplicação – *Wiki.js*

Para instalar a aplicação recorreu-se à sua imagem official do *Wiki.js* para o *docker*. Assim, para executá-la deve-se indicar a imagem(*requarks/wiki*) a utilizar, bem como o seguinte conjunto de variáveis de ambiente, relativas à base de dados:

- DB_TYPE: variável que indica o motor de base de dados escolhido.Assim, de acordo com o escolhido, indica-se "mysql";
- DB_HOST: endereço ip do servidor que hospeda a base de dados. Uma vez que pode haver a existência de várias bases de dados, deve-se indicar o endereço ip do proxy referente à base de dados;
- DB_PORT: porta em que a base de dados escuta. No caso do proxySQL esta porta é 16033;
- DB_USER: usuário que tem acesso à base de dados;
- DB_PASS: password válida do usuário indicado;
- DB_NAME: nome da base de dados criada em todos os servidores de base de dados.

Este processo é executado, recorrendo ao módulo "*docker_container*" existente em *ansible*, como demonstra a figura 3.4.

```
- name: Run wiki
  docker_container:
    name: wiki
    image: requarks/wiki:2
    state: started
    restart_policy: unless-stopped
    networks:
      - name: "{{ net_work_docker_name }}"
    ports:
      - "8080:3000"
    env:
      DB_TYPE: mysql
      DB_HOST: "{{ db_ip }}"
      DB_PORT: "3306"
      DB_USER: "{{ db_user }}"
      DB_PASS: "{{ db_password }}"
      DB_NAME: "{{ db_name }}"
```

Figura 3.4: Excerto do role *app*

3.4 Proxy para a base de dados – proxySQL

De modo a gerir as várias bases de dados existentes, bem como realizar um *read-write split*, foi necessário instalar um *proxy* na conexão com a base de dados. Assim, instala-se, em *ansible*, o *proxy* da seguinte forma:

```
- name: Create proxy configuration file
  template: src=proxysql.cnf dest=~/proxysql.cnf
  become_user: "{{ common_username }}"

- name: Run ProxySQL
  docker_container:
    name: proxySQL
    image: proxysql/proxysql
    state: started
    restart_policy: unless-stopped
    networks:
      - name: "{{ net_work_docker_name }}"
    volumes:
      - /home/{{ common_username }}/proxysql.cnf:/etc/proxysql.cnf
    ports:
      - "16032:6032"
      - "16033:6033"
      - "16070:6070"
```

Figura 3.5: Excerto do role *proxySQL*

Ao observar a figura, nota-se a conceção de um volume durante a criação deste *container*. Este processo dá-se devido à necessidade de conservar os dados do *proxy* de forma permanente. Desta forma, sempre que o serviço seja reiniciado, continua a preservar a informação referentes às bases de dados existentes, bem como o processo que deve seguir para reencaminhar os vários acessos. Contudo, para que o *pro'xySQL* obtenha esta informação, é necessário que após a primeira execução, seja realizada a sua configuração.

Para realizar a configuração do *proxySQL*, é necessário indicar quais os endereços dos servidores e a respetiva porta, onde se encontram as bases de dados para leitura (*slaves*) ou para escrita (*master*). Para além disto adiciona-se o servidor de base de dados usado como *master*, na lista de servidores para leitura, porém com um peso inferior, implicando a sua utilização apenas nos casos em que nenhum dos *slaves* se encontrem operacionais.

Posteriormente, deve ser adicionado, seguindo o mesmo padrão, a informação referente aos utilizadores que tem acesso à base de dados nos diferentes servidores, bem como a informação do utilizador de monitorização de cada uma delas.

3.5 Base de dados – MySQL

Como motor de base de dados utilizou-se o *MySQL*. Assim, instalou-se, como se observa na figura 3.6, o mesmo em todos os servidores responsáveis por suportar o serviço, independentemente do papel a executar (*slave* ou *master*).

Apesar de neste trabalho, ter-se realizado a instalação do motor de base de dados na maquina nativa, seria preferível que a instalação deste acontece-se através de um *docker conteiner*, uma vez que permitiria garantir que todas as bases de dados têm a mesma configuração, para além de permitir hospedar mais do que servidor de base de dados por *host*.

```
- name: Install Mysql packages
  apt:
    name: "{{ item }}"
    state: latest
  loop: "{{ db_packages }}"
```

Figura 3.6: Excerto dos roles *db*

Após a instalação, modificou-se o ficheiro que guarda as configurações do *MySQL* (*/etc/mysql/mysql.conf.d/mysqld.cnf*) de modo a alterar o *bind-address* e a acrescentar as informações necessárias para a replicação da base de dados (estes dados podem diferenciar consoante o papel a executar).

Após a modificação do conteúdo do ficheiro, foi necessário reiniciar o serviço do *MySQL*, bem como criar um usuário, com todas as permissões, capaz de aceder e configurar a base de dados.

```
- name: restart mysql
  service:
    name: mysql
    state: restarted

- name: Cleanup users with same name accounts
  mysql_user:
    name: "{{ replica_user }}"
    host_all: yes
    state: absent

- name: Criar Usuario base
  mysql_user:
    name: "{{ db_user }}"
    password: "{{ db_password }}"
    priv: '*.*:ALL,GRANT'
    state: present
```

Figura 3.7: Excerto dos roles *db*

Após realizar este processo, em todos os servidores de base de dados, executa-se a criação de outros usuários, utilizados, por exemplo, pelos *slaves* para atualizar a sua cópia dos dados, ou pelo *proxySQL*, para monitorizar o estado dos servidores.

3.5.1 Master

Depois de realizar o processo inicial, é necessário que no servidor de base de dados *master*, execute um conjunto de comandos, de modo a obter o nome do ficheiro de *logs* e a sua posição a utilizar na replicação. Esta informação deve ser passada para o *host* que invocou a execução do *playbook*, para ser utilizada posteriormente, nos servidores de base de dados salves.

```
- name: get pos log bin
  shell: grep -oP '\s+\K(\d+)' temp.txt > pos.txt
  become_user: "{{ common_username }}"

- name: ansible copy file from remote to local.
  fetch:
    src: ~/name.txt
    dest: ~/
    flat: yes
  become_user: "{{ common_username }}"

- name: ansible copy file from remote to local.
  fetch:
    src: ~/pos.txt
    dest: ~/
    flat: yes
  become_user: "{{ common_username }}"

- name: ansible copy file from remote to local.
  fetch:
    src: ~/ip.txt
    dest: ~/
    flat: yes
  become_user: "{{ common_username }}"
```

Figura 3.8: Excerto dos role *db_master*

Após a replicação estar ativa em todos os servidores de base de dados *slaves*, deve-se criar no servidor *master* a base de dados *wiki* a ser utilizada pela aplicação.

3.5.2 Slave

No servidor de base de dados *slave*, após os passos iniciais deve-se estabelecer a replicação da base de dados com a *master*. Assim, é necessário obter a informação proveniente do servidor de base de dados *master*, já registada em ficheiros no *host* que executou o *playbook*.

```
bin_log_file_name: "{{ lookup('file', '~/name.txt') }}"
bin_log_file_pos: "{{ lookup('file', '~/pos.txt') }}"
```

Figura 3.9: Excerto dos role *db_slave*

Por fim, deve-se inicia-se o *slave*, com o nome e posição do ficheiro de *logs*, endereço e usuários a utilizar na replicação.

3.6 Agentes da *Google Cloud Platform*

Ao instalar e configurar todos os componentes acima referidos, a aplicação está pronta a ser utilizada (acedendo ao endereço do servidor que executa o *reverse proxy* na porta 80). Contudo, de modo a possibilitar um monitoramento do estado de cada servidor, deve-se instalar um agente em cada servidor. Os agentes utilizados, são disponibilizados pela *Google Cloud Platform* e permitem aceder à informação recorrendo, também, aos seus serviços.

Utiliza-se este serviço específico da *Google Cloud Platform*, devido à comodidade que este proporciona, uma vez que permite aceder a toda a informação através de uma interface gráfica intuitiva e personalizável.

3.7 Adição de complementos em *runtime*

Após a instalação inicial dos componentes da aplicação deve ser possível variar o número de servidores *Wiki.js*, ou o número de bases de dados *slaves*. Assim, criou-se dois *playbook's* distintos responsáveis por adicionar cada um destes componentes.

Para adicionar um novo servidor de *Wiki.js*, deve-se executar os passos demonstrados anteriormente, aos quais se adicionam a carga de trabalho referente à reconfiguração do *reverse proxy* e reinício do serviço.

```
- name: add novos server on conf
  shell: sed -i -E "s/upstream wikijs \{/upstream wikijs {\n\tserver {{ item }}:{{ app_port }};/g" ./default.conf
  loop: "{{ ip_app }}"
```

Figura 3.10: Reconfiguração do *reverse proxy*

Por fim, para acrescentar uma base de dados *slave*, deve-se instalar e configurar todos os componentes necessários (como referido acima), bem como adicionar o estado atual da base de dados a replicar. Posteriormente, reconfigura-se o *proxySQL*, de modo a adicionar o novo servidor à distribuição de carga.

```
- name: create database
  shell: mysql -u{{ db_user }} -p{{ db_password }} -e "CREATE DATABASE {{ db_name }};"

- name: Copiar a base de dados
  ansible.builtin.copy:
    src: "~/{{ db_name }}.sql"
    dest: "~/{{ db_name }}.sql"
  become_user: "{{ common_username }}"

- name: replicate database configurada
  shell: mysql -u{{ db_user }} -p{{ db_password }} {{ db_name }} < /home/{{ common_username }}/{{ db_name }}.sql
```

Figura 3.11: Excerto adicionar *db_slave*

4 Ferramentas de monitorização da aplicação

Para monitorizar e observar a aplicação em ambientes de testes, utilizámos a ferramenta da Google *Cloud Cloud Monitoring agent*. Este agente obtém métricas do sistema e das aplicações das instâncias de VM e envia-as para a cloud, onde podem ser analisadas, através da criação de dashboards. Os resultados obtidos nas figuras seguintes resultaram dos testes de avaliação realizados na próxima fase.

4.1 Especificações das máquinas

Antes de analisar os resultados provenientes da monitorização da aplicação, é importante realçar as especificações das máquinas que utilizámos para instalar os componentes da aplicação:

- CPU - 4vCPUs com tecnologia das plataformas de CPU Intel Cascade Lake e Ice Lake
- Memória - 3 GB
- Disco - SSD

Todos os testes de performance foram realizados em máquinas com estas caraterísticas, de modo a minimizar a quantidade de variáveis que pudessem influenciar os resultados, isolando o fator de performance para a arquitetura de deployment utilizada (número de componentes, balanceamento de carga, entre outros). Também foi por este mesmo motivo que utilizámos máquinas com caraterísticas superiores às necessárias para a aplicação *Wiki.js*.

4.2 Monitorização geral das máquinas

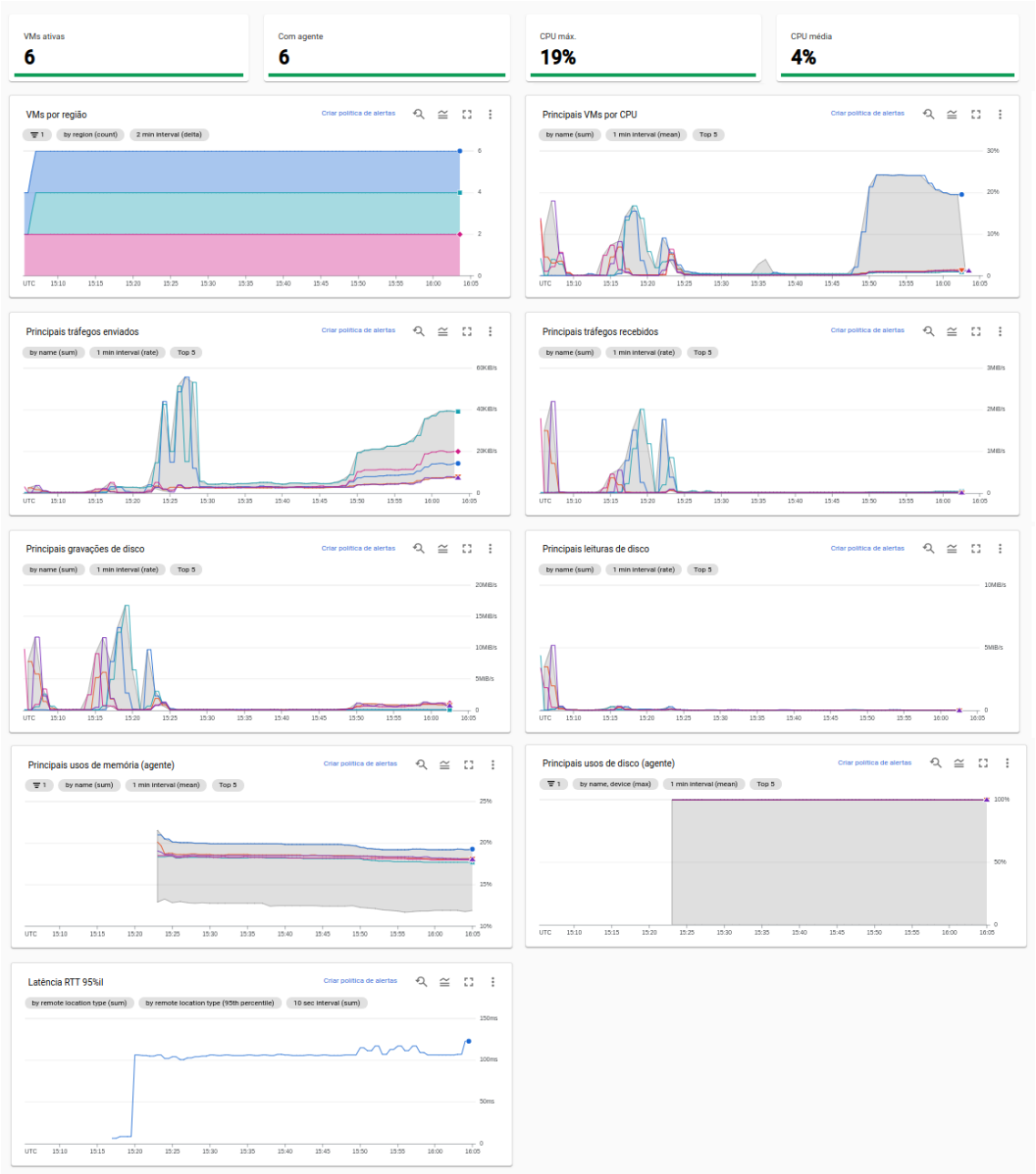

Figura 4.1: Monitorização geral das Máquinas Virtuais

Nesta dashboard geral, podemos verificar o consumo de recursos geral das máquinas onde estão instalados todos os componentes da aplicação. As métricas que decidimos analisar são:

- VM's por região - métrica secundária, mas que nos permite verificar a distribuição das máquinas pelo mundo.

- VM's por CPU - métrica muito importante, uma vez que nos permite verificar que máquinas exigem mais do processador.
- Tráfegos enviados - permite observar a quantidade de dados que cada máquina envia na rede.
- Tráfegos recebidos - permite observar a quantidade de dados que cada máquina recebe na rede.
- Operações I/O de disco - Esta métrica reflete o uso do disco que cada máquina faz.
- Uso da memória - Métrica muito importante para verificar o uso da memória de cada máquina.
- Latência - Métrica muito importante, uma vez que permite, mais facilmente, avaliar a performance da aplicação, na medida em que permite analisar o tempo de resposta da aplicação aos pedidos que lhe são feitos.

Para as dashboards individuais, decidimos medir e analisar as seguintes métricas

- Uso do CPU
- Uso da memória
- Operações I/O do disco
- Tráfego na rede

Selecionámos estas métricas, uma vez que são representativas do uso de recursos computacionais de um computador. Como se trata de uma aplicação web, é importante medir e analisar todo o tráfego na rede, de modo a identificar possíveis problemas como a sua congestão. Para além disso, como a aplicação utiliza bases de dados, medir todas operações que ocorrem no disco é fundamental para avaliar o impacto que as queries executadas nos sistema de gestão de base de dados estão a ter, e conseguirmos identificar possíveis *bottlenecks* neste componente. Quanto ao CPU e memória, são obviamente os recursos vitais para o funcionamento de qualquer aplicação, pelo que é extremamente importante medir os seus usos. Por fim, ao selecionar este conjunto de métricas, podemos comparar, mais facilmente, o uso de recursos computacionais entre diferentes componentes da aplicação e entre diferentes *deployments*, como vai ser visto na próxima etapa.

4.3 Monitorização da máquina virtual que contém a base de dados Master

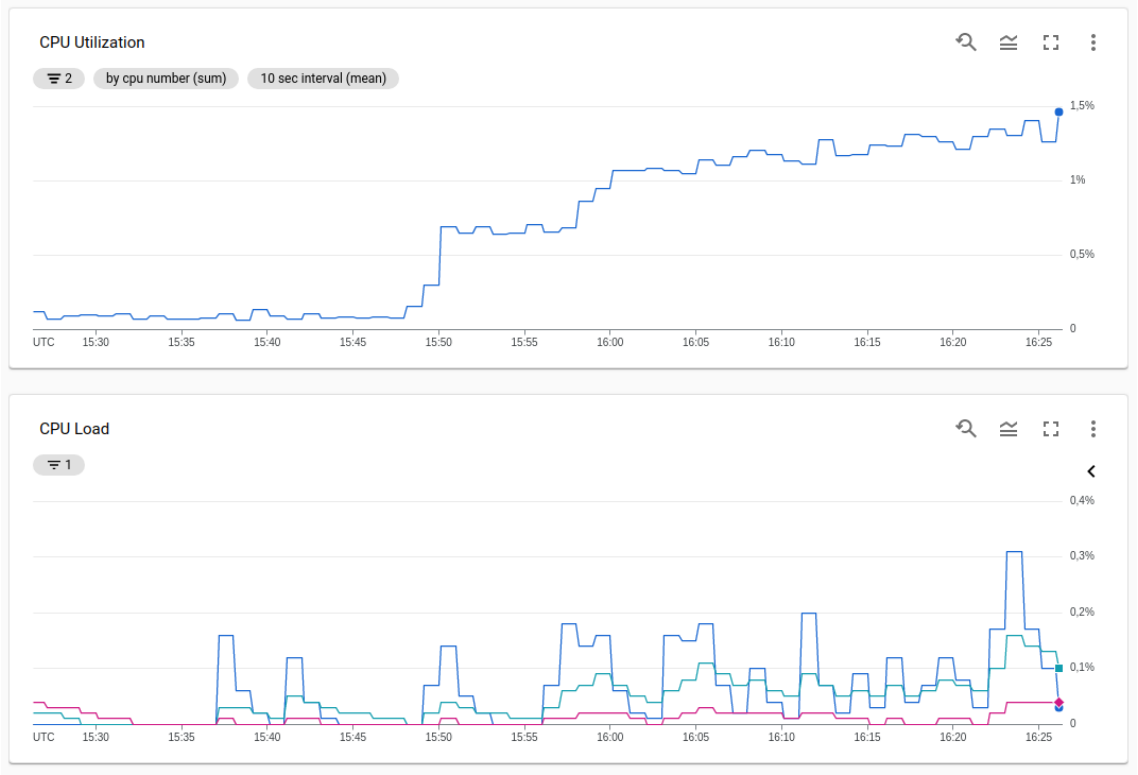

Figura 4.2: Monitorização da máquina virtual que contém a base de dados Master (CPU)

Nestes gráficos, conseguimos observar que, na máquina virtual que contém a base de dados master, o processador está com pouca carga, raramente ultrapassando 1% de utilização.

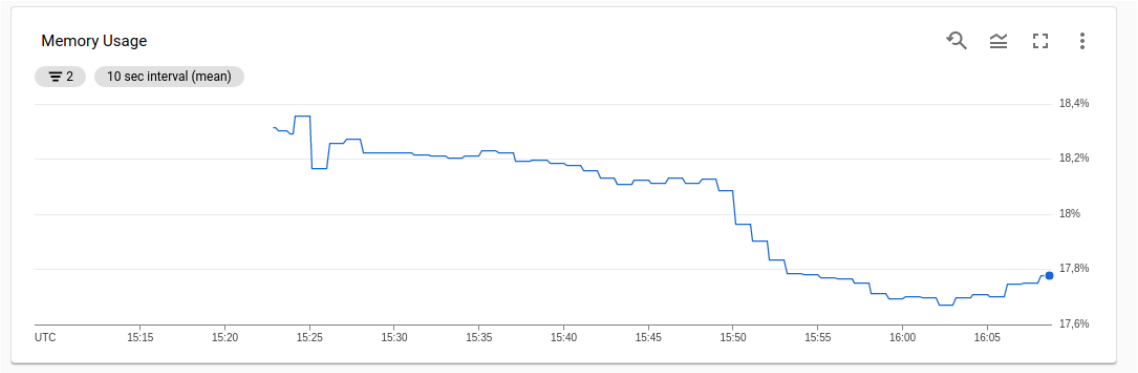

Figura 4.3: Monitorização da máquina virtual que contém a base de dados Master (Uso da memória)

Já no uso da memória, podemos observar que é um recurso muito mais utilizado pela base de dados, uma vez que quase atingiu 20% de utilização (aproximadamente 0,6 GB).

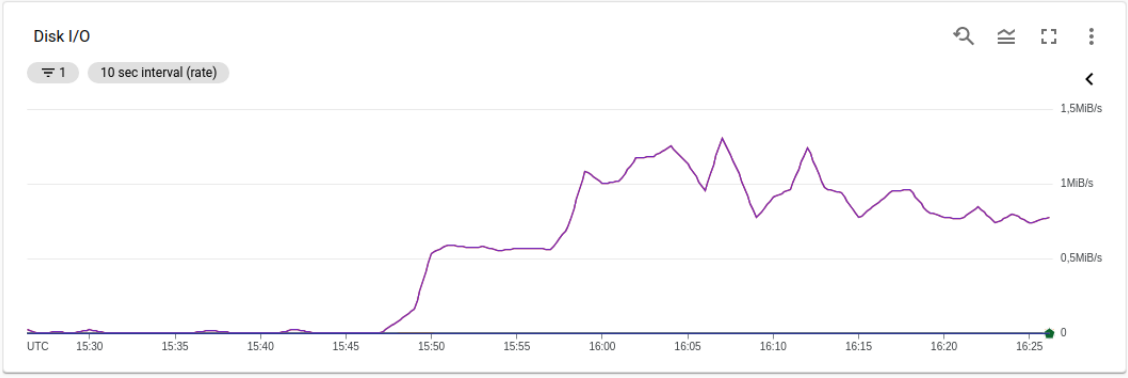

Figura 4.4: Monitorização da máquina virtual que contém a base de dados Master (Disk I/O)

Quanto às operações de I/O no disco, a base de dados master apresentou velocidades de quase 1,5 MB/s.

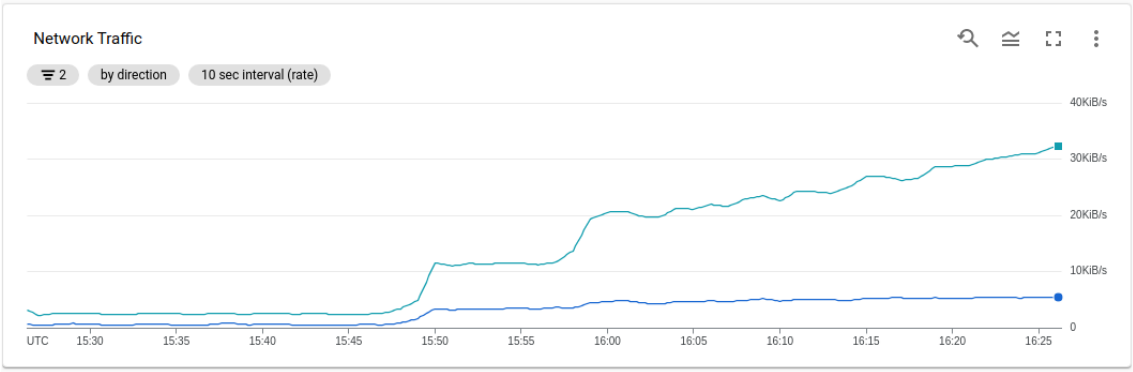

Figura 4.5: Monitorização da máquina virtual que contém a base de dados Master (Tráfego na rede)

Podemos verificar que existiu alguma atividade na rede, crescendo ao longo do tempo, o que faz sentido, uma vez que cresceu com o aumento de utilizadores do sistema.

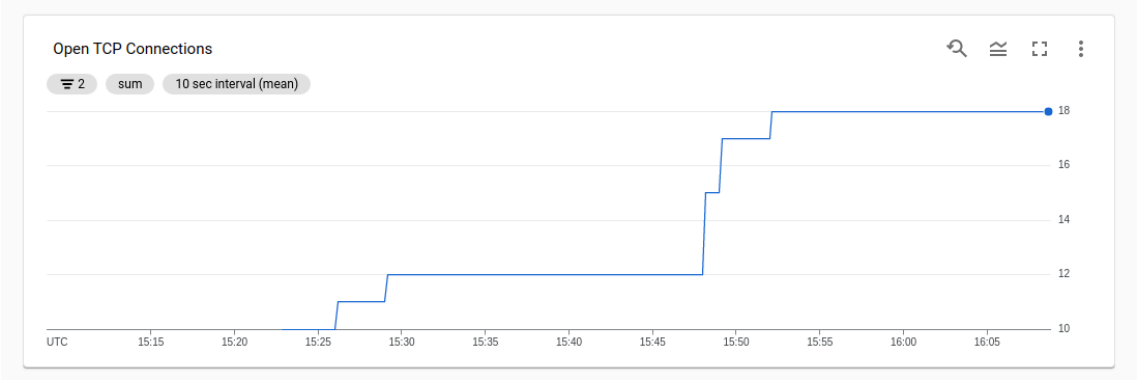

Figura 4.6: Monitorização da máquina virtual que contém a base de dados Master (Conexões TCP)

O mesmo se pode observar para o número de conexões TCP, já que também aumentou com o aumento do número de utilizadores da aplicação.

4.4 Monitorização da máquina virtual que contém o servidor wiki

Figura 4.7: Monitorização da máquina virtual que contém o servidor Wiki (CPU)

Podemos observar que, ao contrário da máquina que contém a base de dados master, a máquina que tem o servidor wiki apresentou um uso bastante maior do processador, aproximando-se de 30% em certas instâncias. Este maior uso pode ser justificado com o superior número de processamento de eventos que o servidor wiki tem de fazer, em relação à base de dados.

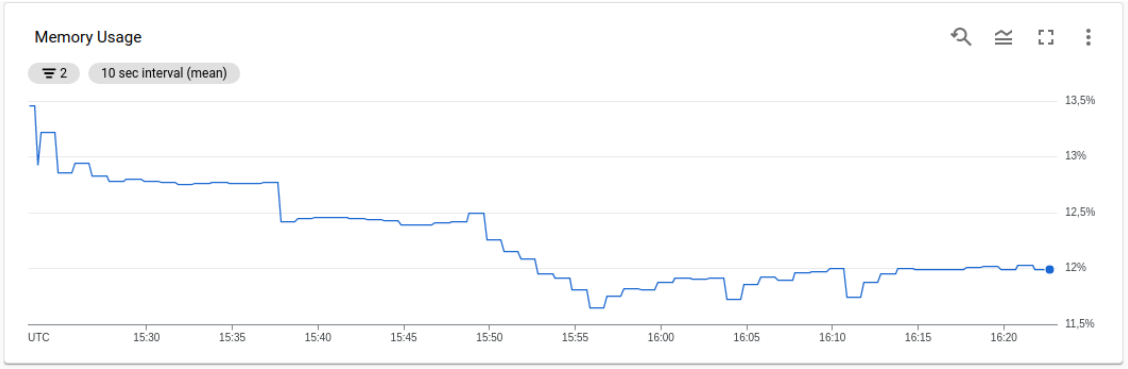

Figura 4.8: Monitorização da máquina virtual que contém o servidor Wiki (Memória)

Quanto ao uso de memória, o servidor da wiki manteu-se, geralmente, à volta de 12.5% (correspondente a 0,38 GB).

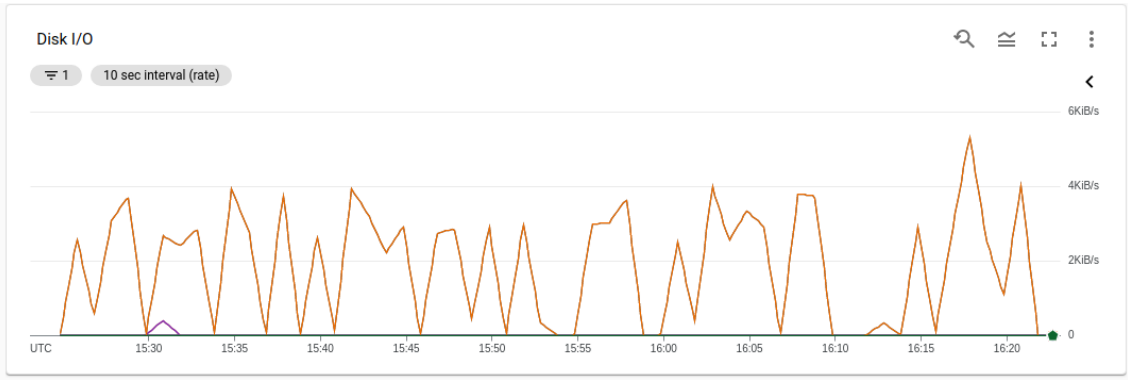

Figura 4.9: Monitorização da máquina virtual que contém o servidor Wiki (I/O Disk)

Ora, como seria de se esperar, a utilização do disco por parte da máquina que contém o servidor wiki é extremamente baixa, estando sempre na ordem de poucos KB/s. Este resultado faz todo o sentido, uma vez que todas as operações de escrita e leitura do disco são feitas pelas bases de dados, que estão em máquinas diferentes.

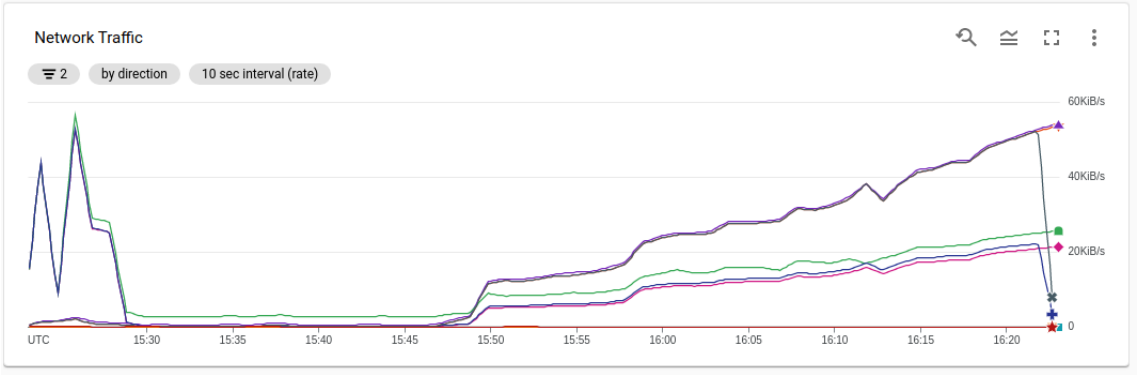

Figura 4.10: Monitorização da máquina virtual que contém o servidor Wiki (Tráfego na rede)

Tal como observámos para a máquina que contém a base de dados, podemos verificar que, para a máquina que contém o servidor da wiki, o uso da rede foi aumentando gradualmente com o aumento do número de utilizadores e pedidos que chegavam ao sistema.

5 Ferramentas de avaliação

Para além das ferramentas de monitorização, é importante considerar um conjunto de ferramentas de avaliação da aplicação, para que se assim se consiga avaliar a aplicação de forma realista.

De maneira a conseguir simular os diferentes comportamentos dos utilizadores, será necessário definir um conjunto de testes automáticos. Para tal, utilizou-se o JMeter, uma ferramenta que irá permitir analisar e avaliar a performance da nossa aplicação.

5.1 Definição dos Testes

Numa primeira fase, foi necessário definir quais os testes mais importantes a considerar para a monitorização do comportamento da nossa aplicação. Como se trata de uma aplicação bastante simples, decidiu-se testar os casos de acesso à página inicial da wikijs, e os casos de criação de novos utilizadores. O primeiro caso representa um HTTP Request utilizando o método GET, enquanto que o segundo caso representa um HTTP Request utilizando o método POST. Desta forma, é possível testar os casos de leitura e escrita na base de dados, respetivamente.

Antes de prosseguir com a análise dos resultados, foi necessário definir o tamanho dos nossos testes, isto é, definir quantos utilizadores virtuais seriam considerados e qual o número de pedidos a serem simulados. Após uma breve reflexão, decidiu-se considerar 8 utilizadores virtuais e um total de 5000 pedidos.De seguida, definiram-se 3 arquiteturas diferentes, para testar o comportamento da aplicação nos diferentes cenários definidos:

- **Arquitetura Básica:** 1 proxy NginX, 1 servidor *Wiki.js*, 1 proxy proxySQL, 1 base de dados master
- **Arquitetura Normal:** 1 proxy NginX, 1 servidor *Wiki.js*, 1 proxy proxySQL, 1 base de dados master, 3 base de dados slave
- **Arquitetura Complexa:** 1 proxy NginX, 2 servidores *Wiki.js*, 1 proxy proxySQL, 1 base de dados master, 3 base de dados slave

A representação de cada uma das arquiteturas pode ser observada na secção dos anexos, no final do presente relatório.

5.2 Acessos Página Inicial

Como foi referido anteriormente, foi produzido um teste automático que irá simular o acesso à página inicial da *Wiki.js*. Este teste foi simulado para cada uma das arqui-

teturas e para cada uma, foram registados os comportamentos da nossa aplicação, sendo que serão destacados os Response Times e a Latency.

5.2.1 Response Times

Como se pode observar na Figura, são apresentados os gráficos obtidos em cada uma das arquiteturas, estando os gráficos ordenados por ordem crescente de complexidade. Percebe-se que quanto mais complexa for a nossa arquitetura, menor serão os tempos de resposta associados a cada pedido, sendo de realçar que o comportamento nos três casos é bastante semelhante, variando apenas a gama de valores de Response Time entre os quais variam. Assim, na arquitetura básica e na arquitetura normal, os response times variam entre 295ms e 320ms, enquanto que na arquitetura complexa, esses valores são relativamente inferiores, devido à presença de um servidor extra, que permite diminuir a carga de trabalho de cada servidor, e por isso diminuir os tempos de resposta.

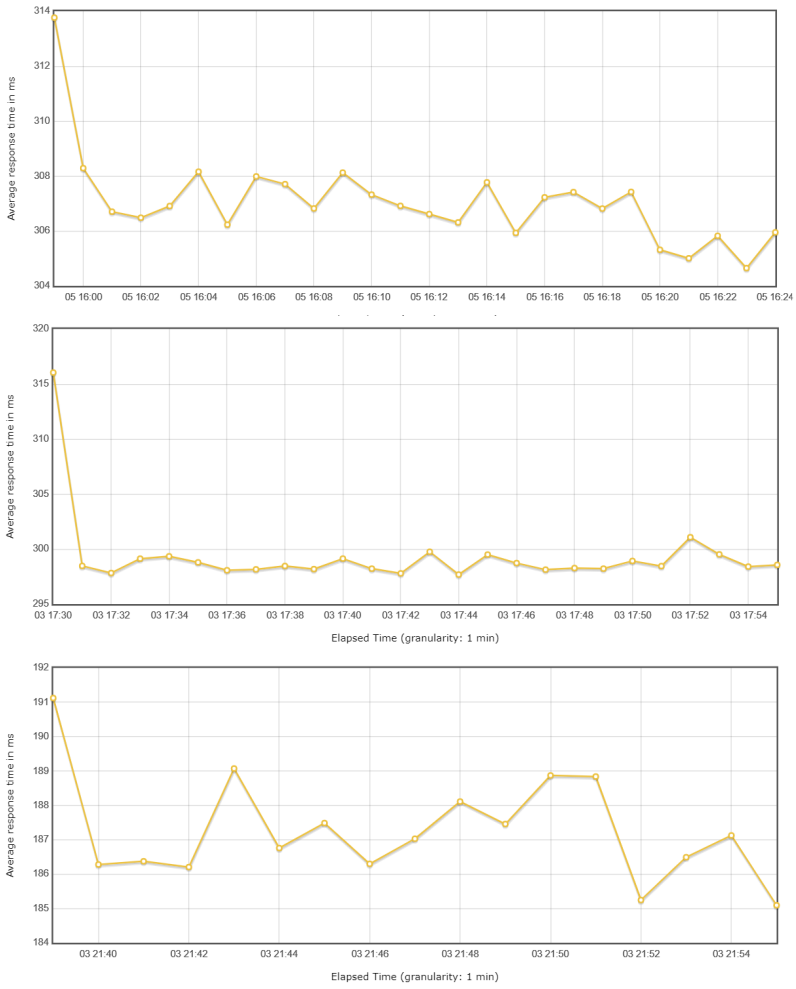

Figura 5.1: Home Page Response Times

5.2.2 Latency Vs Request

Relativamente aos atrasos ao longo do tempo, tal como se pode observar na Figura, onde são apresentados os gráficos obtidos em cada uma das arquiteturas, estando os gráficos ordenados por ordem crescente de complexidade, percebe-se que, para a arquitetura básica e normal, independentemente do aumento do número de pedidos por segundo, os atrasos rondam sempre os 300ms. Já na arquitetura complexa, quantos mais pedidos são realizados por segundo, menor é o atraso médio.

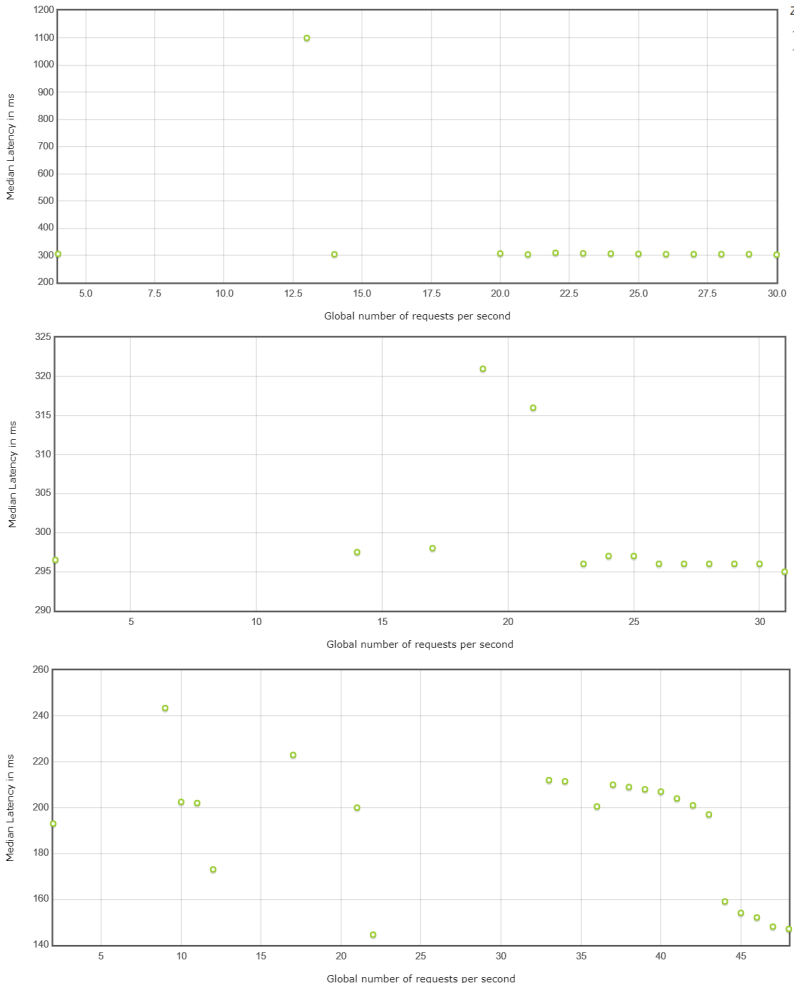

Figura 5.2: Home Page Latency Vs Request

5.3 Criar Utilizadores

De maneira a testar a escrita nas bases de dados, simulou-se a criação de vários utilizadores na aplicação, de maneira a analisar o comportamento desta em situações de várias escritas na base de dados.

5.3.1 Response Times Distribution

Analisando os gráficos obtidos, a primeira conclusão a retirar é que em cada um dos testes, todos os utilizadores foram criados, não havendo por isso falhas. De seguida, comparando os resultados obtidos na arquitetura complexa com as restantes arquiteturas, percebe-se que a presença de um servidor adicional faz com que os tempos de resposta sejam muito inferiores, sendo a maioria dos tempos de resposta valores inferiores a 500ms.

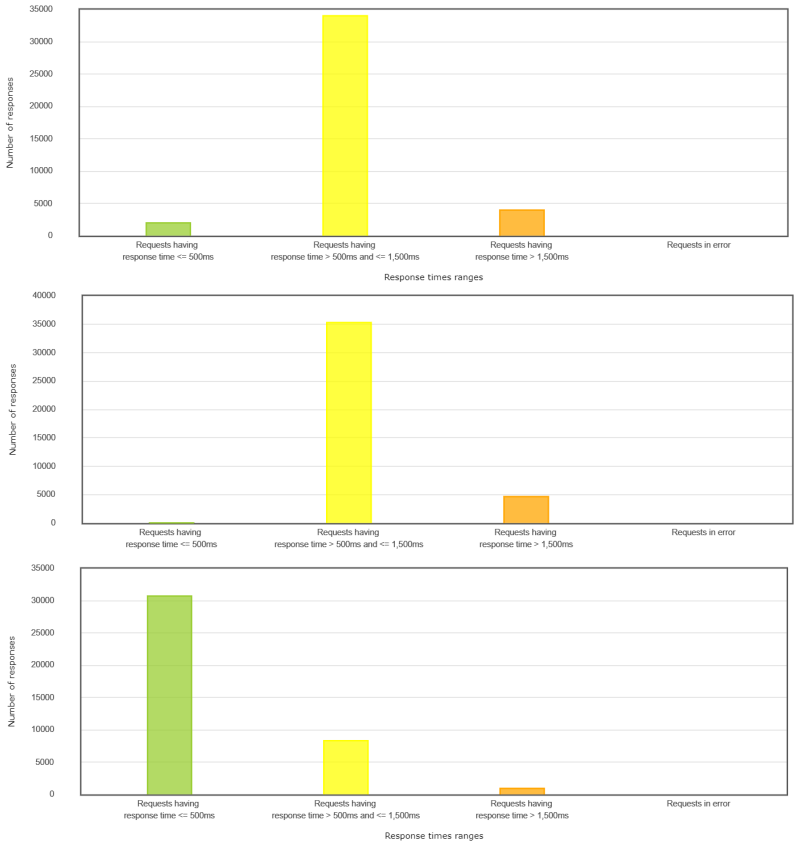

Figura 5.3: Create User Response Times

5.3.2 Latency Over Time

Analisando os atrasos ao longo do tempo, percebe-se que em todas as arquiteturas, à medida que o tempo avança, menor são os atrasos médios das respostas. Tal deve-se ao warmup das caches. Ao longo do tempo, os dados necessários para responder aos pedidos já estão em cache, diminuindo drasticamente a latência da resposta. Além disso, e tal como observado em situações anteriores, os tempos na arquitetura básica e normal são idênticos, e os tempos na arquitetura complexa são relativamente inferiores aos das arquiteturas anteriroes, estabilizando nos 250ms.

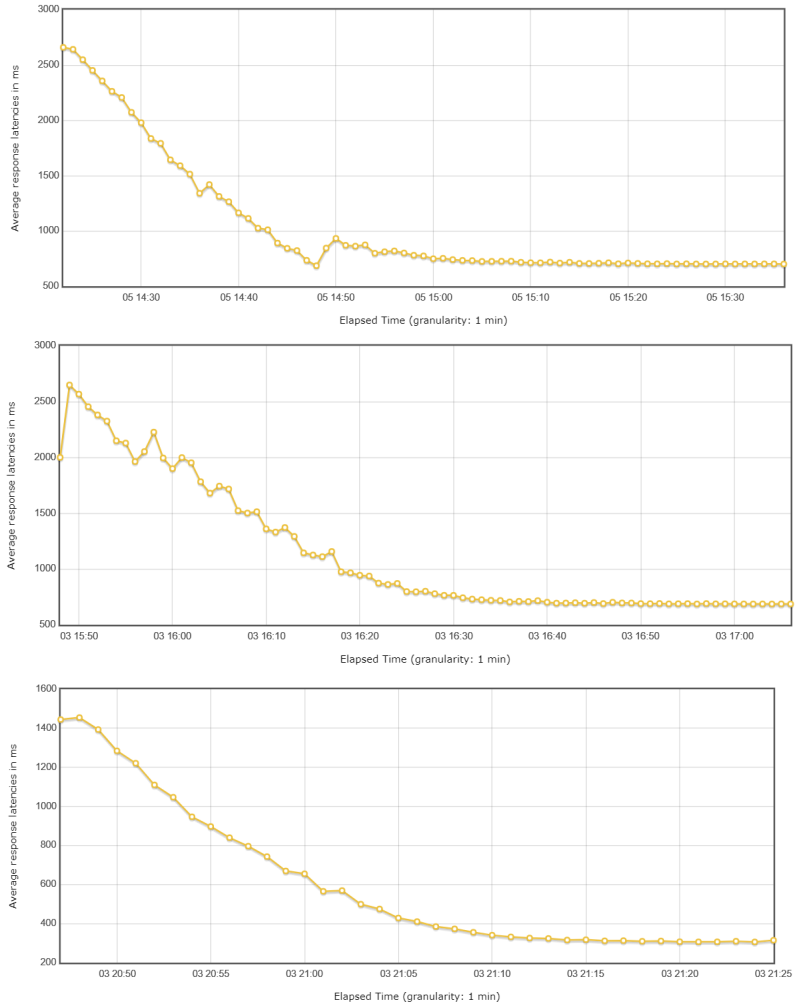

Figura 5.4: Create User Latencies Over Time

6 Conclusão

Neste projeto realizámos a análise da arquitetura da aplicação Wiki.js, e procedemos à sua instalação e monitorização e avaliação de consumo de recursos computacionais.

A maior dificuldade deste trabalho foi a implementação da instalação automática de um número arbitrário de componentes tais como a base de dados o servidor wiki. Na verdade, este requisito obrigou-nos a utilizar aplicações proxy tal como o NginX para efetuar o balanceamento de carga entre vários servidores, e o proxySQL para balancear a carga entre as diferentes bases de dados. Infelizmente, tivémos de escolher entre utilizar bases de dados em containers ou ter replicação da base de dados, uma vez que não conseguimos implementar as duas, em simultâneo.

Utilizar a ferramenta Ansible para a criação de uma instalação automática da aplicação Wiki.js teve as suas vantagens, tais como ter a instalação de todos os componentes num só *package* organizado, e a conveniência de instalar todos os componentes com a mesma ferramenta. Porém, para trabalho futuro e, de modo a melhorar ainda mais o nosso trabalho, poderíamos utilizar ferramentas como o Kubernetes e/ou serviços da Google Cloud que suportam escalonamento automático, configuração automática de containers, bases de dados, entre outros. Uma última vantagem de termos uma instalação automática independente dos serviços da Google é o facto de podermos utilizar esta implementação para efetuar o deployment da aplicação em servidores que não pertencem à Google Cloud, o que se traduz numa enorme vantagem de reutilização.

Nas últimas fases, fomos capazes de testar a performance e consumo de recursos computacionais de todos os componentes da aplicação e máquinas virtuais, através da utilização de ferramentas como o JMeter para a simulação de uso intensivo da aplicação por parte de utilizadores reais, agentes da Google Cloud para obter métricas úteis para análise e observação crítica e as ferramentas da Google Cloud para organizar e visualizar todos os dados de forma organizada e clara, em dashboards.

Em suma, fomos capazes de realizar todas as etapas deste projeto com sucesso e acreditámos que contribuiu para consolidar e aprofundar os nossos conhecimentos e experiência de deployment de software.

A Anexos

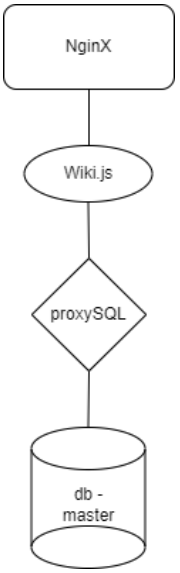

Figura A.1: Arquitetura Básica

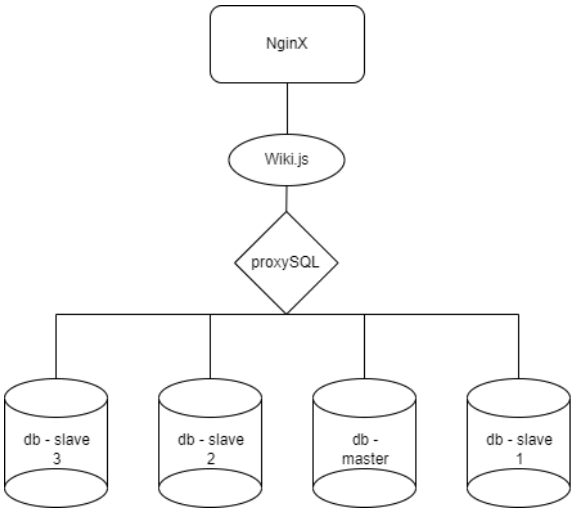

Figura A.2: Arquitetura Normal

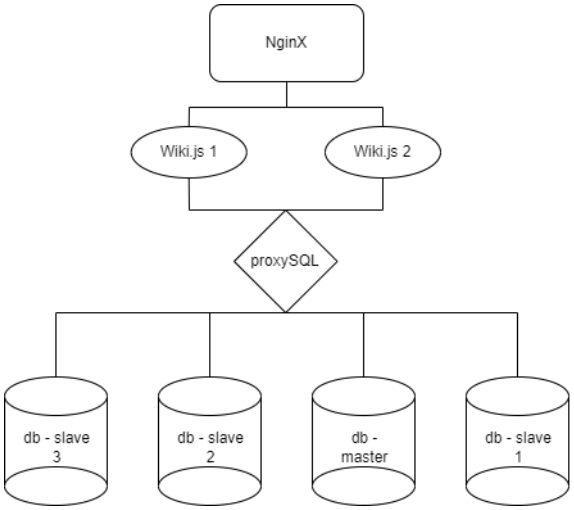

Figura A.3: Arquitetura Complexa