

UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA  
INFRAESTRUTURAS DE CENTROS DE DADOS

---

## Wiki.js

---

### Grupo 11

-  
João Linhares (a86618)  
Joel Ferreira (a89982)  
Rui Azevedo (a80789)

20 de fevereiro de 2021

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>8</b>
<b>2</b>	<b>Arquitectura</b>	<b>9</b>
<b>3</b>	<b>Pontos de Configuração</b>	<b>9</b>
<b>4</b>	<b>Configuração inicial</b>	<b>10</b>
4.1	Automatização da Instalação . . . . .	11
4.2	Identificação de <i>SPOFs</i> . . . . .	11
4.3	Testes de carga . . . . .	11
4.3.1	Limitações do Wiki.js . . . . .	12
4.3.2	Teste 1 : <i>Login</i> . . . . .	14
4.3.3	Teste 2: Login + Listagem de Páginas + Criação de páginas . . . . .	21
4.3.4	Teste 3: Login + Pesquisa + Obtenção da Página Pesquisada . . . . .	27
4.3.5	Teste 4: Login + Criação de Utilizador . . . . .	32
4.3.6	Teste 5: Login + Consulta de todas as Páginas do Sistema . . . . .	38
4.3.7	Teste 6: Login + Pesquisa + Escolha de Página + Save . . . . .	44
4.4	Análise dos resultados . . . . .	50
<b>5</b>	<b>Arquitetura Implementada</b>	<b>51</b>
5.1	Problemas Encontrados . . . . .	52
5.2	Storage . . . . .	53
5.3	High Availability Cluster . . . . .	54
5.3.1	Arquitetura do High Availability Cluster . . . . .	54
5.3.2	Balanceador de Carga . . . . .	55
5.3.3	Cluster . . . . .	56
5.4	Web Servers e Política de Balanceamento . . . . .	59
5.4.1	Balanceador de Carga . . . . .	59

5.4.2	Web Servers . . . . .	60
5.5	Testes de Carga . . . . .	61
5.5.1	Teste 1 : <i>Login</i> . . . . .	62
5.5.2	Teste 2: Login + Listagem de Páginas + Criação de páginas . . . . .	65
5.5.3	Teste 3: Login + Pesquisa + Obtenção da Página Pesquisada . . . . .	68
5.5.4	Teste 4: Login + Criação de Utilizador . . . . .	72
5.5.5	Teste 5: Login + Consulta de todas as Páginas do Sistema . . . . .	77
5.5.6	Teste 6: Login + Pesquisa + Escolha de Página + Save . . . . .	80
5.6	Análise dos resultados . . . . .	85
<b>6</b>	<b>Conclusão</b>	<b>87</b>
<b>7</b>	<b>Anexos</b>	<b>88</b>

# Listas de Figuras

1	Arquitectura base do Wiki.js . . . . .	9
2	Configuração inicial . . . . .	10
3	Erro Detetado no Response Body. . . . .	13
4	Solução para o problema encontrado. . . . .	14
5	Estatísticas do <i>login</i> com 100 <i>threads</i> . . . . .	16
6	<i>Response Times Over Time</i> do <i>login</i> com 100 <i>threads</i> . . . . .	16
7	Estatísticas do <i>login</i> com 200 <i>threads</i> . . . . .	17
8	<i>Response Times Over Time</i> do <i>login</i> com 200 <i>threads</i> . . . . .	17
9	Estatísticas do <i>login</i> com 300 <i>threads</i> . . . . .	18
10	<i>Response Times Over Time</i> do <i>login</i> para 300 <i>threads</i> . . . . .	18
11	Estatísticas do <i>login</i> com 400 <i>threads</i> . . . . .	19
12	<i>Response Times Over Time</i> do <i>login</i> com 400 <i>threads</i> . . . . .	19
13	Estatísticas do <i>login</i> com 500 <i>threads</i> . . . . .	20
14	<i>Response Times Over Time</i> do <i>login</i> para 500 <i>threads</i> . . . . .	20
15	Estatísticas do Teste 2 para 100 <i>threads</i> . . . . .	22
16	<i>Response Times Over Time</i> do Teste 2 para 100 <i>threads</i> . . . . .	22
17	Estatísticas do Teste 2 para 200 <i>threads</i> . . . . .	23
18	<i>Response Times Over Time</i> do Teste 2 para 200 <i>threads</i> . . . . .	23
19	Estatísticas do Teste 2 para 300 <i>threads</i> . . . . .	24
20	<i>Response Times Over Time</i> do Teste 2 para 300 <i>threads</i> . . . . .	24
21	Estatísticas do Teste 2 para 400 <i>threads</i> . . . . .	25
22	<i>Response Times Over Time</i> do Teste 2 para 400 <i>threads</i> . . . . .	25
23	Estatísticas do Teste 2 para 500 <i>threads</i> . . . . .	26
24	<i>Response Times Over Time</i> do Teste 2 para 500 <i>threads</i> . . . . .	26
25	Pedidos feitos no teste 3. . . . .	27
26	Estatísticas do Teste 3 para 100 <i>threads</i> . . . . .	28
27	Estatísticas do Teste 3 para 200 <i>threads</i> . . . . .	29

28	Estatísticas do Teste 3 para 300 <i>threads</i> . . . . .	30
29	Estatísticas do Teste 3 para 400 <i>threads</i> . . . . .	31
30	Estatísticas do Teste 3 para 500 <i>threads</i> . . . . .	32
31	Pedidos feitos no teste 4. . . . .	33
32	Estatísticas do Teste 4 para 100 <i>threads</i> . . . . .	34
33	Estatísticas do Teste 4 para 200 <i>threads</i> . . . . .	35
34	Estatísticas do Teste 4 para 300 <i>threads</i> . . . . .	36
35	Estatísticas do Teste 4 para 400 <i>threads</i> . . . . .	37
36	Estatísticas do Teste 4 para 500 <i>threads</i> . . . . .	38
37	Pedidos feitos no teste 5. . . . .	39
38	Estatísticas do Teste 5 para 100 <i>threads</i> . . . . .	40
39	Estatísticas do Teste 5 para 200 <i>threads</i> . . . . .	41
40	Estatísticas do Teste 5 para 300 <i>threads</i> . . . . .	42
41	Estatísticas do Teste 5 para 400 <i>threads</i> . . . . .	43
42	Estatísticas do Teste 5 para 500 <i>threads</i> . . . . .	44
43	Pedidos feitos no teste 6. . . . .	45
44	Estatísticas do Teste 6 para 100 <i>threads</i> . . . . .	46
45	Estatísticas do Teste 6 para 200 <i>threads</i> . . . . .	47
46	Estatísticas do Teste 6 para 300 <i>threads</i> . . . . .	48
47	Estatísticas do Teste 6 para 400 <i>threads</i> . . . . .	49
48	Estatísticas do Teste 6 para 500 <i>threads</i> . . . . .	50
49	Arquitetura implementada na Google Cloud Platform. . . . .	52
50	High Availability do Wiki.js. . . . .	53
51	Arquitetura da Storage Partilhada. . . . .	54
52	Arquitetura do High Availability Cluster. . . . .	55
53	Balanceador Interno sobre o Cluster. . . . .	56
54	Página do Cluster com os nodos existentes. . . . .	57
55	Nodo Wikijs 2. . . . .	57
56	Grupo de Recursos no Cluster. . . . .	58

57	Recurso do Filesystem. . . . .	58
58	Recurso do PostgreSQL. . . . .	59
59	<i>Google Cloud Balancer</i> sobre 4 instâncias da aplicação <i>Wikijs</i> . . . . .	60
60	Arquitetura do <i>front-end</i> . . . . .	61
61	Estatísticas da arquitectura implementada com o Teste 1 para 100 <i>threads</i> . .	62
62	Estatísticas da arquitectura implementada com o Teste 1 para 200 <i>threads</i> . .	63
63	Estatísticas da arquitectura implementada com o Teste 1 para 300 <i>threads</i> . .	63
64	Estatísticas da arquitectura implementada com o Teste 1 para 400 <i>threads</i> . .	64
65	Estatísticas da arquitectura implementada com o Teste 1 para 500 <i>threads</i> . .	64
66	Estatísticas da arquitectura implementada com o Teste 2 para 100 <i>threads</i> . .	65
67	Estatísticas da arquitectura implementada com o Teste 2 para 200 <i>threads</i> . .	66
68	Estatísticas da arquitectura implementada com o Teste 2 para 300 <i>threads</i> . .	66
69	Estatísticas da arquitectura implementada com o Teste 2 para 400 <i>threads</i> . .	67
70	Estatísticas da arquitectura implementada com o Teste 2 para 500 <i>threads</i> . .	67
71	Estatísticas da arquitectura implementada com o Teste 3 para 100 <i>threads</i> . .	68
72	Estatísticas da arquitectura implementada com o Teste 3 para 200 <i>threads</i> . .	69
73	Estatísticas da arquitectura implementada com o Teste 3 para 300 <i>threads</i> . .	70
74	Estatísticas da arquitectura implementada com o Teste 3 para 400 <i>threads</i> . .	71
75	Estatísticas da arquitectura implementada com o Teste 3 para 500 <i>threads</i> . .	72
76	Estatísticas da arquitectura implementada com o Teste 4 para 100 <i>threads</i> . .	73
77	Estatísticas da arquitectura implementada com o Teste 4 para 200 <i>threads</i> . .	74
78	Estatísticas da arquitectura implementada com o Teste 4 para 300 <i>threads</i> . .	75
79	Estatísticas da arquitectura implementada com o Teste 4 para 400 <i>threads</i> . .	76
80	Estatísticas da arquitectura implementada com o Teste 4 para 500 <i>threads</i> . .	77
81	Estatísticas da arquitectura implementada com o Teste 5 para 100 <i>threads</i> . .	78
82	Estatísticas da arquitectura implementada com o Teste 5 para 200 <i>threads</i> . .	78
83	Estatísticas da arquitectura implementada com o Teste 5 para 300 <i>threads</i> . .	79
84	Estatísticas da arquitectura implementada com o Teste 5 para 400 <i>threads</i> . .	79
85	Estatísticas da arquitectura implementada com o Teste 5 para 500 <i>threads</i> . .	80

86	Estatísticas da arquitectura implementada com o Teste 6 para 100 <i>threads</i>	.. .	81
87	Estatísticas da arquitectura implementada com o Teste 6 para 200 <i>threads</i>	.. .	82
88	Estatísticas da arquitectura implementada com o Teste 6 para 300 <i>threads</i>	.. .	83
89	Estatísticas da arquitectura implementada com o Teste 6 para 400 <i>threads</i>	.. .	84
90	Estatísticas da arquitectura implementada com o Teste 6 para 500 <i>threads</i>	.. .	85

## **Lista de Tabelas**

1	Desempenho Login . . . . .	15
---	----------------------------	----

# 1 Introdução

Num mundo onde a interconectividade se tornou um bem adquirido, vários desafios têm vindo a surgir relativos à forma como as infraestruturas de centros de dados dão resposta a este número crescente de utilizadores. Consequentemente, as indústrias de IT têm que se adaptar de maneira a fornecer serviços de alta disponibilidade, escaláveis e tolerantes a faltas.

É neste contexto que surge o caso de estudo sobre o *Wiki.js*, uma plataforma capaz de gerar páginas *wiki*. É pretendido, com este relatório, apresentar toda a fase de planeamento e operacionalização de uma instalação da aplicação que garanta alta disponibilidade e tolerância a faltas.

Dado isto, numa primeira fase irá ser apresentada uma configuração *naive* do sistema, e, a partir daí, ir eliminando os pontos únicos de falha (*SPOFs*). Ainda numa fase inicial, serão realizados testes de carga com o fim de descobrir quais as operações críticas do sistema.

Por fim, e tendo em conta toda a análise anterior, irá ser apresentada uma configuração de alta disponibilidade e tolerante a faltas, capaz de dar resposta a um número elevado de clientes em tempo útil. Inclusive, serão feitos testes de carga a esta última configuração com o objectivo de analisar o desempenho da mesma.

## 2 Arquitectura

A *Wiki.js* é um motor de criação de páginas *wiki open-source*. A arquitectura base da aplicação é bastante simples, sendo composta apenas por três camadas lógicas, nomeadamente, camada de apresentação, camada de lógica de negócio e camada de persistência de dados.

O *frontend* e *backend* foram desenvolvidos, respectivamente, com as *frameworks* *Vue.js* e *Node.js*. Quanto à camada de persistência de dados, a *Wiki.js* é flexível quanto à escolha do sistema de base de dados a usar. Os sistemas suportados são os seguintes: *MariaDB*, *PostgreSQL*, *MySQL*, *MS SQL Server* e *SQLite*. Neste caso de estudo irá ser usado o sistema *PostgreSQL*.

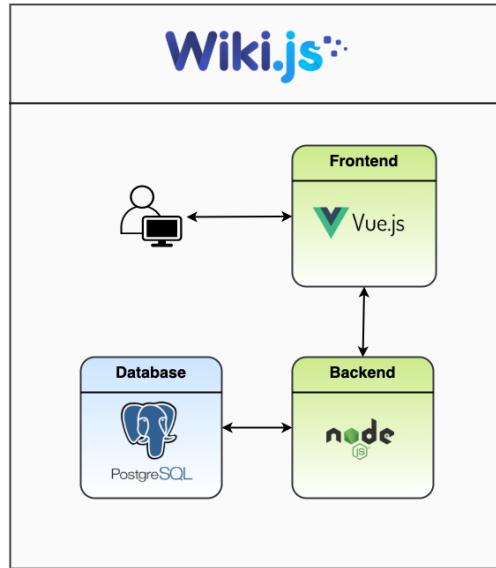


Figura 1: Arquitectura base do *Wiki.js*

## 3 Pontos de Configuração

O ficheiro *config.yml* é o ponto de configuração da aplicação. É possível, com isto, escolher o sistema da base de dados, bem como definir o utilizador e a base de dados a usar. Para além disto, ainda é possível alterar muitas outras configurações, como por exemplo, a porta onde o servidor irá ficar à escuta, a pasta onde a aplicação irá guardar ficheiros temporários, definir os tipos de comunicação possíveis, e em particular, uma configuração que permite definir o serviço de base de dados como de alta disponibilidade, isto é, se existem várias instâncias da base de dados.

## 4 Configuração inicial

Numa configuração inicial, houve a dúvida na escolha entre duas configurações. Uma seria instalar a plataforma apenas numa máquina virtual, tendo o sistema de base de dados e a aplicação em si a correrem na mesma máquina. A outra opção, era implementar o *frontend* e *backend* na mesma máquina virtual e o sistema de base de dados numa outra máquina. O grupo optou pela segunda opção pelo simples motivo de se ter já uma latência na comunicação entre as duas máquinas para teste.

As máquinas escolhidas têm as mesmas características, nomeadamente:

- Tipo de máquina: *n1-standard-4*
- Sistema Operativo: *Ubuntu 20.04 LTS*
- Processador: 4 *vCPUs*
- Memória: 15GB
- Zona: *europe-west1-b*
- Disco: 10GB

Dado isto, é pretendido que a partir da arquitectura apresentada na Figura 2, e através de incrementais refinamentos arquitecturais, chegar a uma configuração final com alta disponibilidade e sem pontos únicos de falhas.

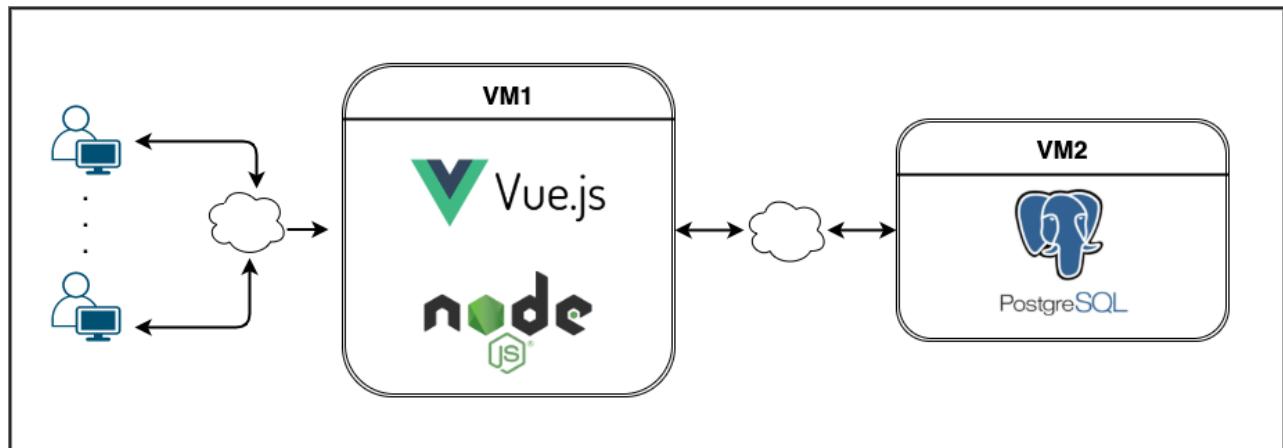


Figura 2: Configuração inicial

## 4.1 Automatização da Instalação

Para a configuração inicial decidimos aplicar alguns dos conhecimentos adquiridos na unidade curricular de *System Deployment & Benchmarking* e automatizar a instalação do Wiki.js através da ferramenta *Ansible*.

Foi então criado um *playbook* que tem dois *roles* (base de dados e wiki.js) que configura toda os servidores aplicacionais para comunicarem com uma base de dados em PostgreSQL.

Com o recurso a esta ferramenta a instalação foi bastante simples, podendo usufruir ainda de termos aprofundado os conhecimentos adquiridos noutras unidades curriculares. O *playbook* e os *roles*, podem ser encontrados na secção de Anexos.

## 4.2 Identificação de *SPOFs*

Um dos objectivos de uma infraestrutura é garantir que a falha de certos componentes da topologia seja completamente transparente aos utilizadores e, para além disso, continuar a fornecer o serviço caso tal aconteça. É de elevada importância, para atingir tal objectivo, eliminar os pontos únicos de falha, acrescentando redundância aos componentes do sistema.

Dada a configuração inicial, é fácil perceber que todos os componentes do sistema podem ser considerados pontos únicos de falha, uma vez que a falha de um deles compromete o correcto funcionamento do sistema, pois os componentes têm dependências entre eles. No caso de falha do servidor que aloca a lógica de negócios, toda comunicação com o resto do sistema fica comprometido. Quanto à base de dados, caso esta não esteja operacional, a aplicação não consegue devolver dados ao cliente, o que torna o sistema improdutivo. Inevitavelmente, os servidores *Web* são um *SPOF* pois, uma vez que não estejam funcionais, o utilizador não consegue ter acesso à visualização da aplicação.

## 4.3 Testes de carga

Para medir o desempenho da infraestrutura criada, foram criados diversos testes de carga usando as ferramentas *Selenium* e *JMeter*.

O *Selenium* é uma *framework* que permite realizar testes funcionais a uma aplicação *Web*. Esta ferramenta permite usar uma linguagem de programação para simular um determinado comportamento de um utilizador através da *interface* da aplicação.

O *Apache JMeter* é uma ferramenta que permite fazer testes de carga com um número elevado de *threads* a correr em paralelo. Esta funcionalidade é o que torna o *JMeter* uma ferramenta mais poderosa em relação ao *Selenium* pois, é possível fazer testes com um número elevado de pedidos uma vez que não se está a depender do número de *threads* da máquina que

está a criar os testes.

Uma vez escolhidas as ferramentas para testes de carga, resta apenas definir quais os testes a fazer em cada uma das ferramentas. No caso do *JMeter*, e uma vez que o mesmo é capaz de gerar uma *dashboard* com os dados de teste obtidos, foram extraídos os gráficos considerados mais importantes, particularmente, irão ser apresentadas as percentagens de pedidos efectuados com sucesso e insucesso, número de pedidos por tempo, *hits* por segundo, e uma visão geral dos tempo de resposta.

#### 4.3.1 Limitações do Wiki.js

Durante a realização dos testes de carga, deparamo-nos com um problema. Este problema foi detetado através da análise aos testes efetuados com o *View Results Tree Listener* do JMeter.

Durante a análise dos resultados dos testes usando a GUI do JMeter reparamos que todos os *status codes* dos pedidos POST /graphql eram 200 (Sucesso), mas que no *Response Body* existiam erros que não estavam a ser reportados no *Summary Report* e então reparamos que a percentagem de erro no *Summary Report* poderia estar errada, pois os erros dos pedidos feitos ao GraphQL não estavam a ser reportados. Posto isto em cada pedido GraphQL foi adicionado um *Response Assertion*, para garantir que caso o *Response Body* contivesse o texto "*errors*":, esse fosse reportado. Após esta alteração reparamos que cerca de 95 por cento dos pedidos de login efetuados não eram completados com sucesso. A mensagem de erro e as alterações feitas para deteção de erros no GraphQL são visíveis na figura seguinte:

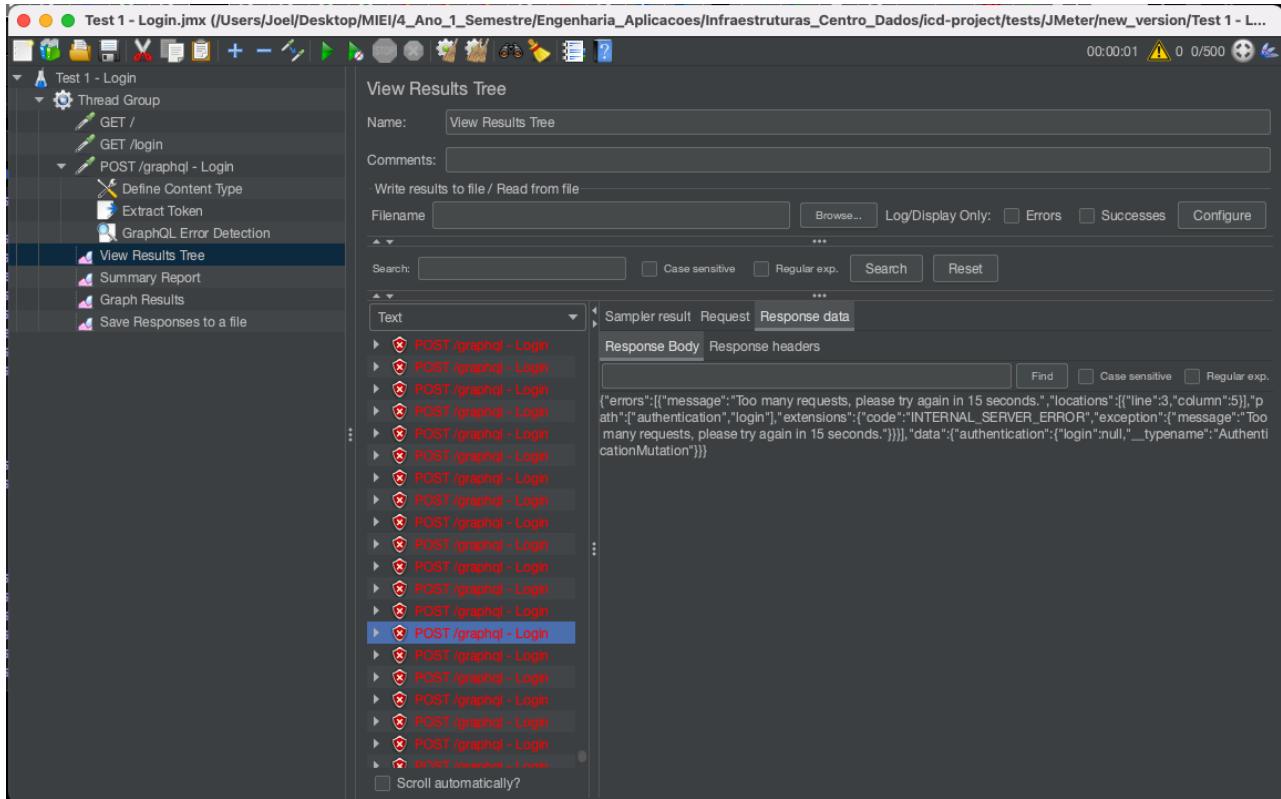


Figura 3: Erro Detetado no Response Body.

O problema consistia numa limitação da aplicação Wiki.js nos pedidos GraphQL na invocação consecutiva de *queries* e *mutations*. No caso do teste de Login, que irá ser apresentado à frente, reparamos que apenas 5 por cento dos pedidos eram concluídos com sucesso, sendo os restantes barrados por existir essa limitação.

Após bastantes horas, descobrimos alguns ficheiros na diretoria do servidor da aplicação que nos ajudaram a identificar o problema. Na diretoria *graphql/directives* do servidor, reparamos que estava a ser utilizado o package *graph-ql-rate-limit-directive*, que permite definir limitações de invocações nos pedidos GraphQL. Depois disso, na diretoria *graphql/schemas* abrimos o *schema authentication* e reparamos que o limite era de 5 pedidos consecutivos sobre esse recurso, pelo que bastou alterar esse valor para o número de pedidos máximo que queríamos executar consecutivamente que é igual ao número de threads máximo utilizado nos testes.

```

# -----
# MUTATIONS
# -----


type AuthenticationMutation {
  login(
    username: String!
    password: String!
    strategy: String!
  ): AuthenticationLoginResponse @rateLimit(limit: 500, duration: 60)

  loginTFA(
    continuationToken: String!
    securityCode: String!
  ): AuthenticationLoginResponse @rateLimit(limit: 5, duration: 60)

  loginChangePassword(
    continuationToken: String!
    newPassword: String!
  ): AuthenticationLoginResponse @rateLimit(limit: 5, duration: 60)

  register(
    email: String!
    password: String!
  )
}

```

Figura 4: Solução para o problema encontrado.

Após esta resolução, todos os testes de login correram sem qualquer tipo de erro de limitação de recursos.

#### 4.3.2 Teste 1 : *Login*

Para testar o serviço de *login* foram criados testes usando o *Selenium* e o *JMeter*.

A tabela abaixo apresenta os valores obtidos através do *Selenium*. É de notar que o tempo obtido é a soma do tempo de carregamento da página inicial mais o tempo da operação.

É possível observar que os tempos vão-se degradando à medida que o número de clientes aumenta. Quando existem 16 *threads* a fazer o pedido de *login* a espera é de, em média, 14 segundos, o que é bastante para apenas uma simples operação.

Tabela 1: Desempenho Login

Threads	#1	#2	#3	Média
1	3.38	3.37	3.40	3.38
2	3.79	5.74	3.95	4.49
4	5.13	5.35	5.40	5.29
8	7.71	7.53	7.49	7.57
16	14.36	13.75	13.94	14.01

Os testes apresentados abaixo, referentes ao *JMeter*, envolvem três tipos de pedido. O primeiro é um simples *GET* da página principal da *Wiki.js*. O segundo é um *GET* da página de *login* e, por fim, um pedido de *POST* com as credências do utilizador.

É de notar, através da análise dos resultados, que os tempos mínimos e máximos de resposta vão aumentando drasticamente à medida que o número de *threads* que realizam pedidos aumenta. Quando o número de *threads* é inferior a 200, não existem erros nos pedidos, isto é, todos os pedidos realizados são atendidos pelo servidor, no entanto, comparando os valores de 100 e 200 *threads*, o tempo de resposta é o dobro. Quando o número de *threads* é de 300 e 400, começam a haver pedidos que não são atendidos pelo servidor e, por sua vez, os tempos de resposta chegam a atingir os 2 minutos, o que é impensável em qualquer sistema. Nos testes realizados com 500 *threads*, verificou-se um comportamento irregular. O tempo de resposta neste caso é inferior aos testes realizados com 300 e 400 *threads* e o número de erros é zero, o que pode ser indicativo que, de alguma maneira, houverem erros que não foram detectados pelo *JMeter*.

Quanto aos gráficos de tempos de resposta sobre o tempo, pode-se observar que vão havendo umas lacunas entre os pedidos de *GET* e o pedido de *POST*. Isto pode-se dever ao facto do servidor estar a tentar dar resposta aos primeiros pedidos de *GET* e, por isso, não efectuar os pedidos de *POST* enquanto os anteriores não forem executados.

Em síntese, e apesar de se terem feito testes até 500 *threads*, pode-se concluir que o servidor não está a dar resposta em tempo útil, nem para apenas 100 *threads*. Observa-se que no caso das 100 *threads* o tempo médio de espera é de 11 segundos e que este valor, vai aumentando chegando aos 24 segundos

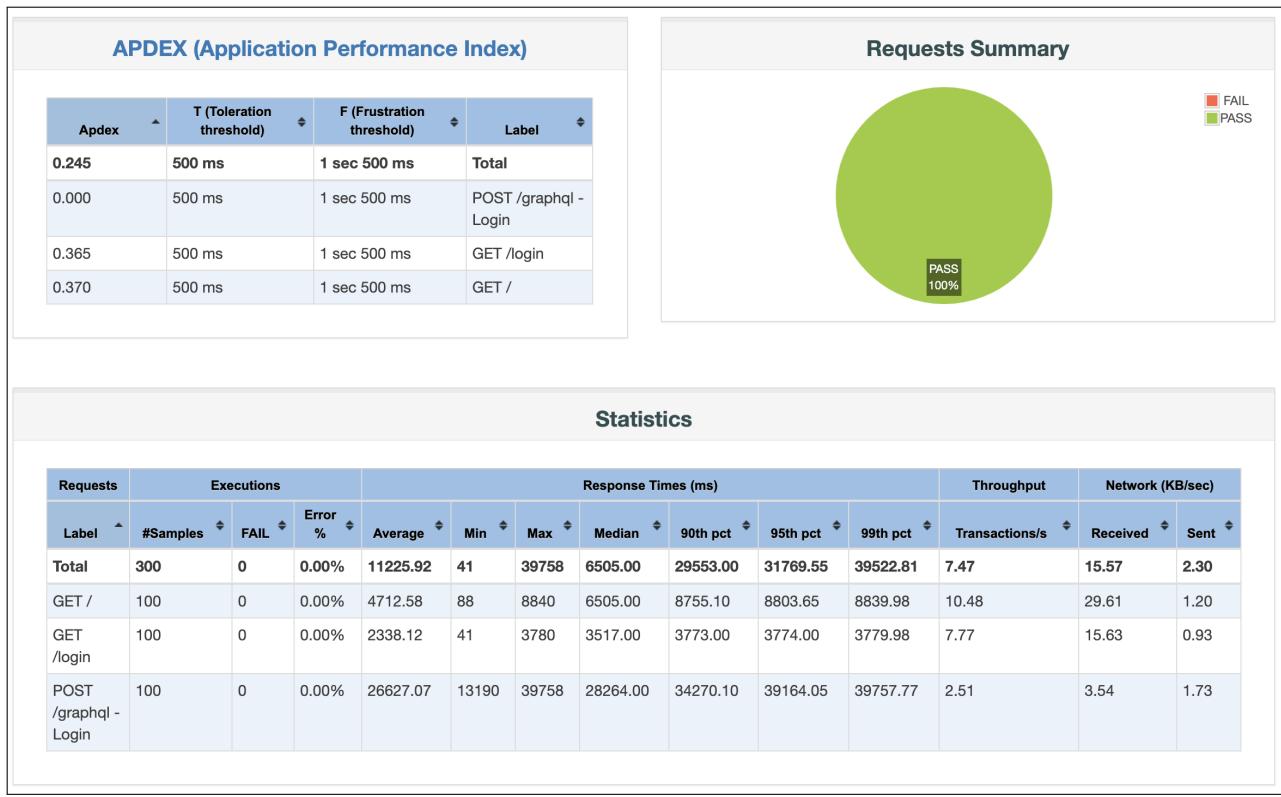


Figura 5: Estatísticas do *login* com 100 threads

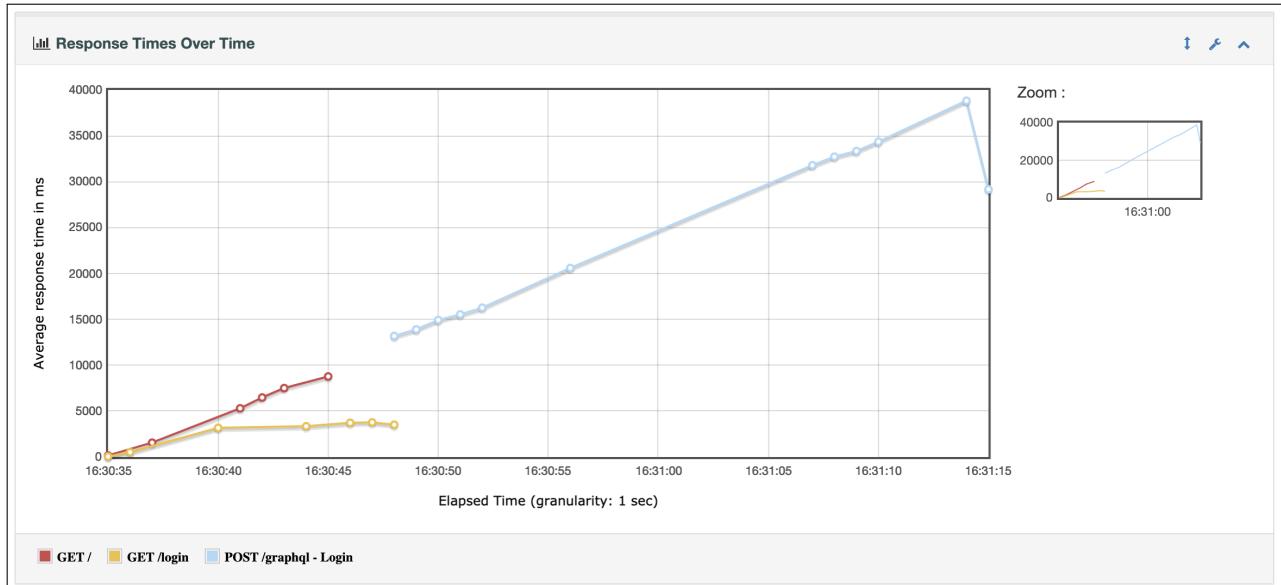


Figura 6: Response Times Over Time do *login* com 100 threads

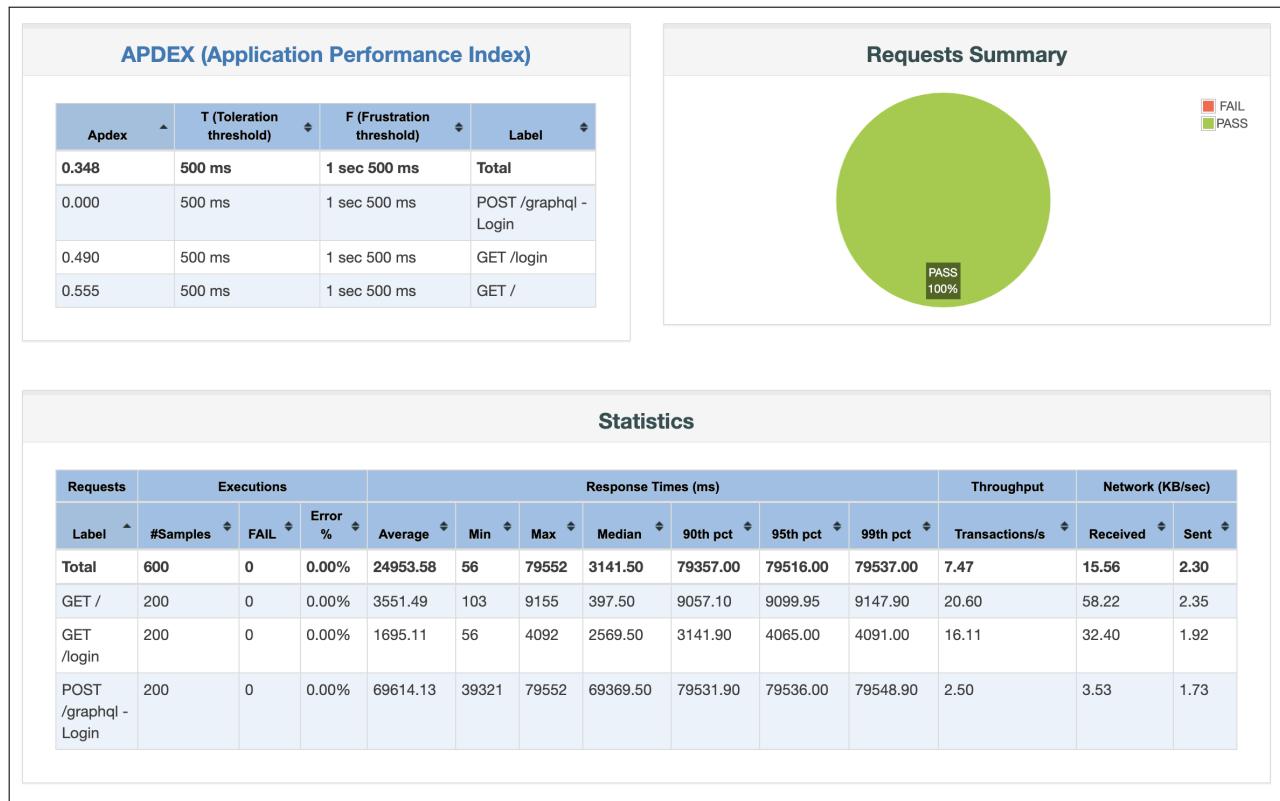
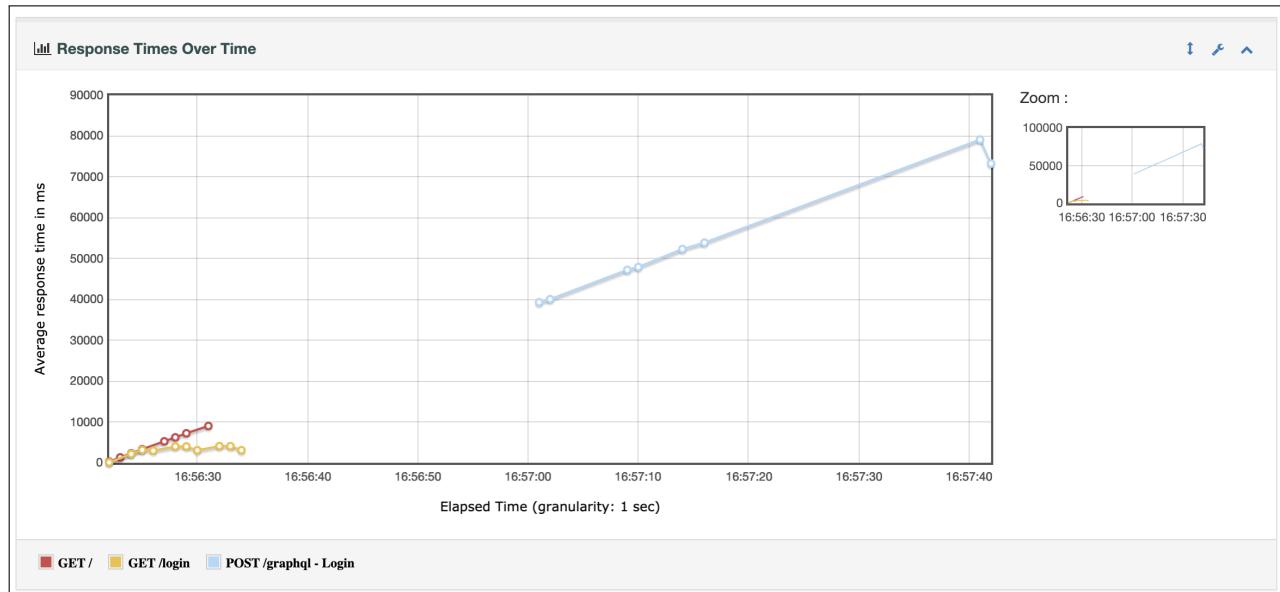


Figura 7: Estatísticas do *login* com 200 *threads*



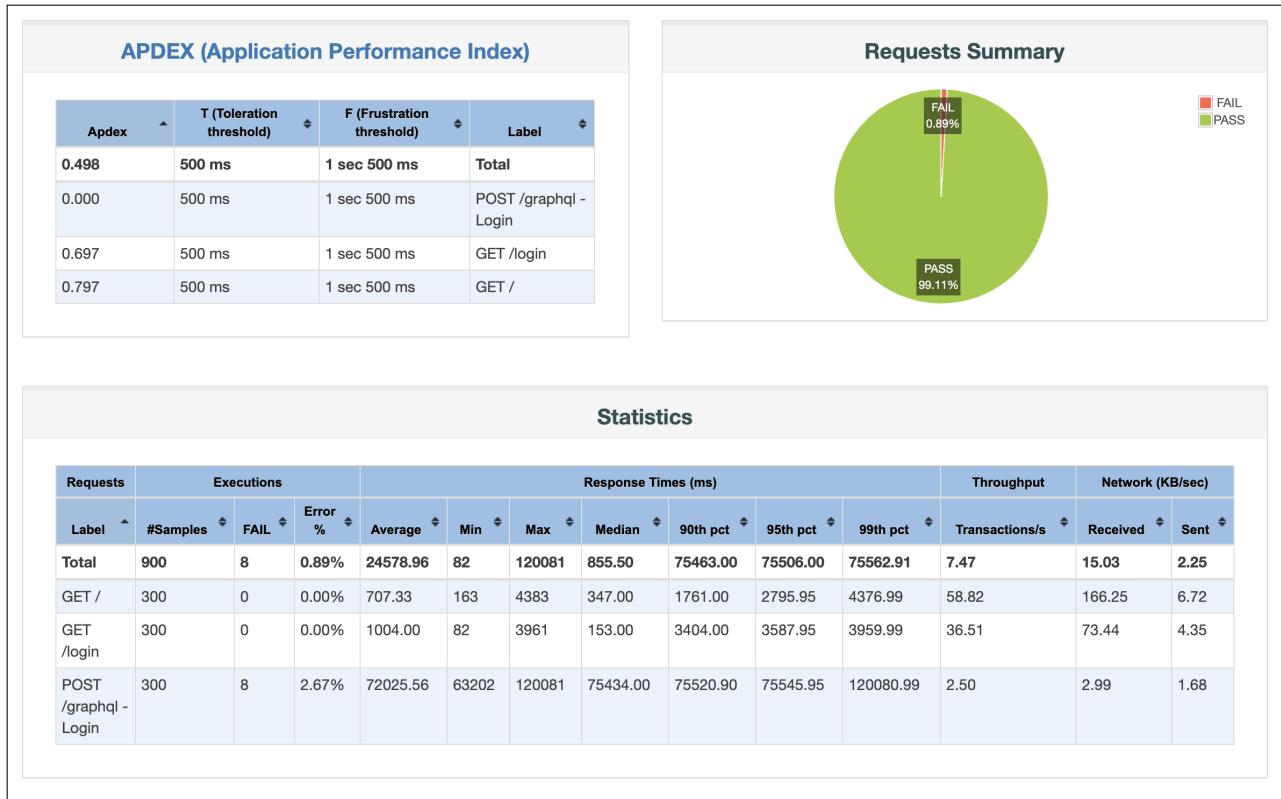


Figura 9: Estatísticas do *login* com 300 threads

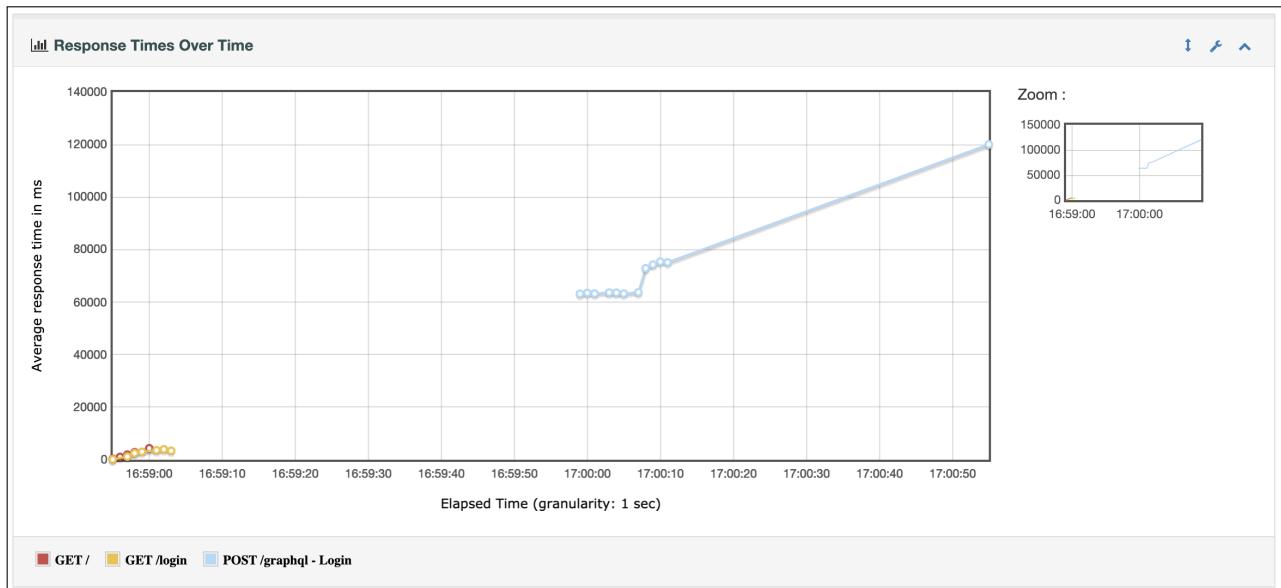


Figura 10: *Response Times Over Time* do *login* para 300 threads

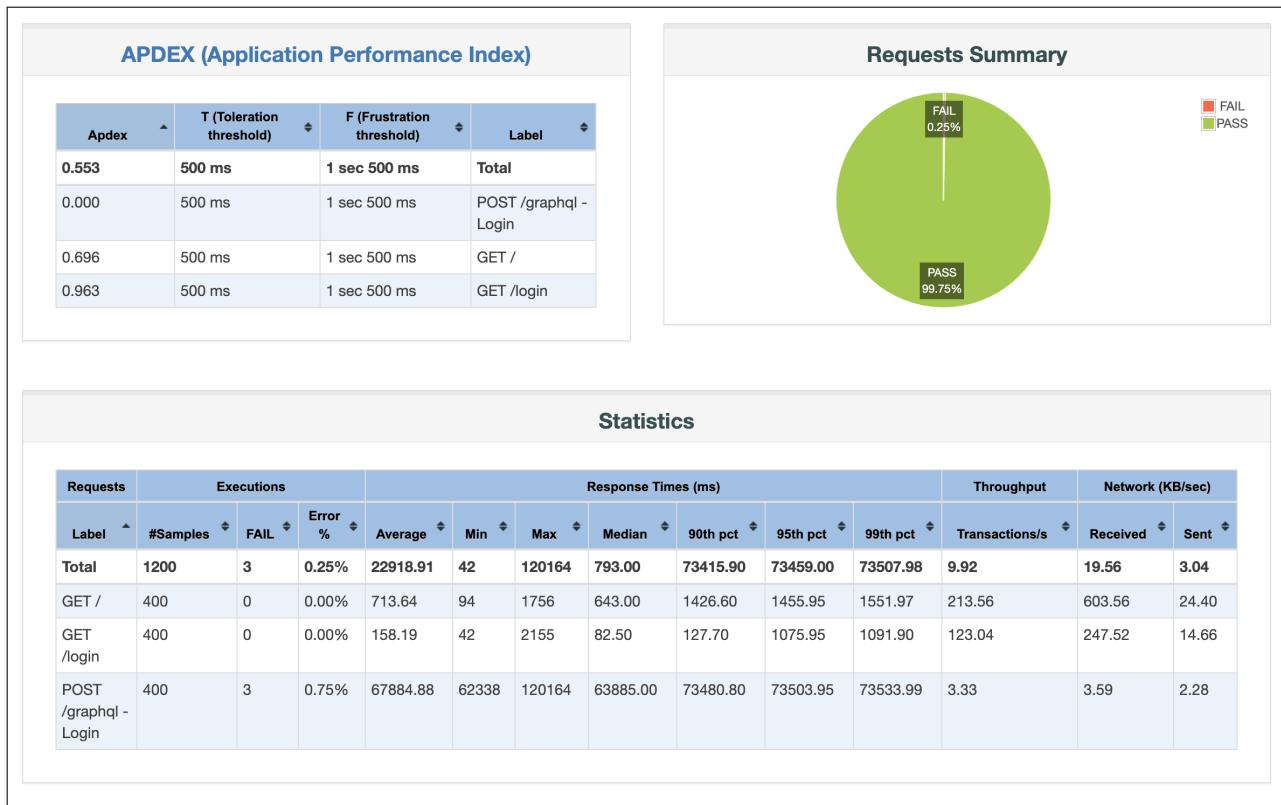


Figura 11: Estatísticas do *login* com 400 threads

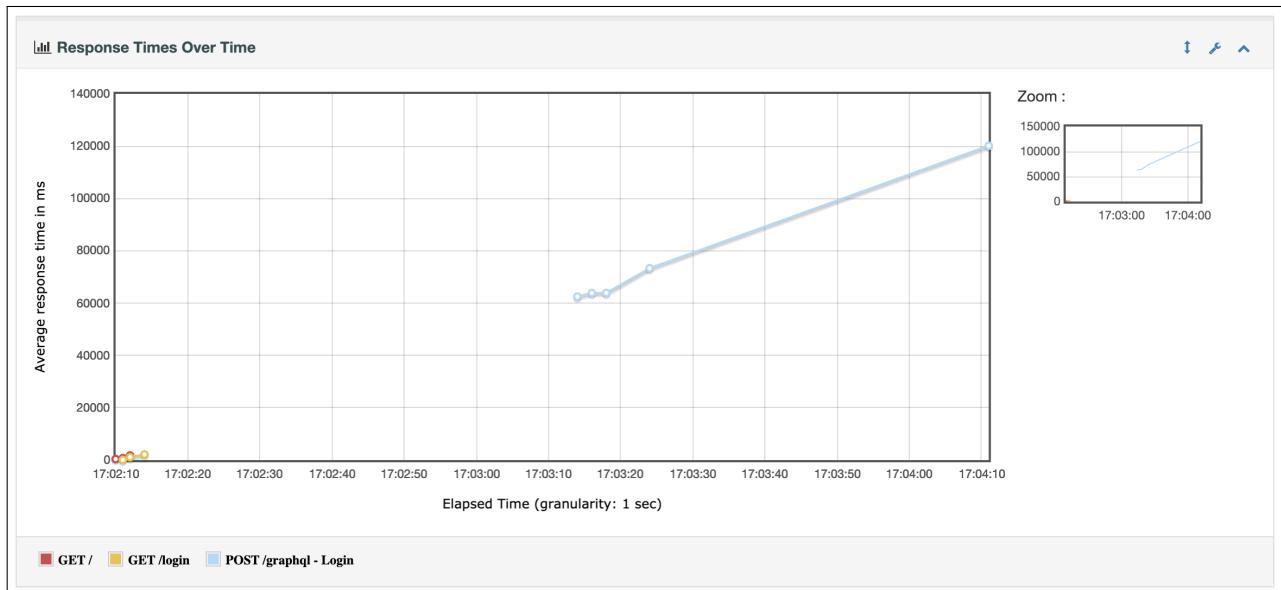


Figura 12: Response Times Over Time do *login* com 400 threads

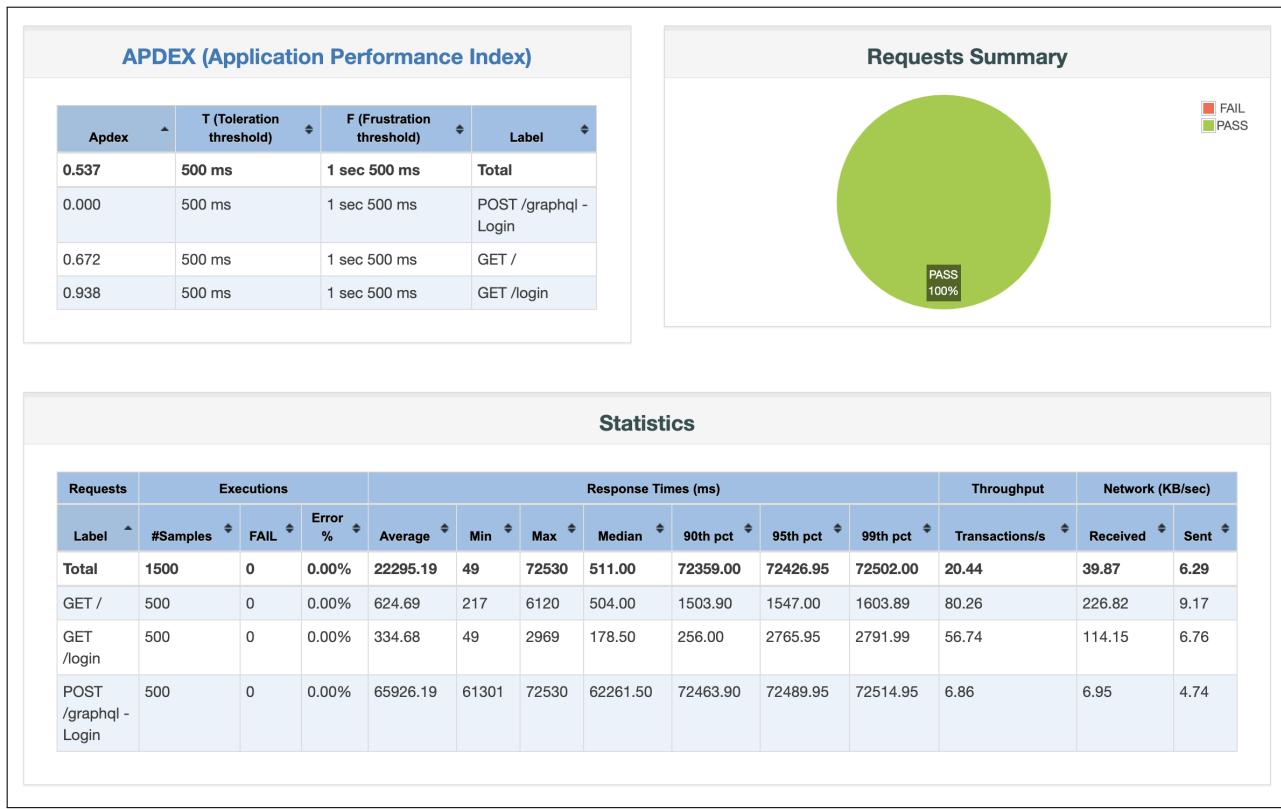


Figura 13: Estatísticas do *login* com 500 threads

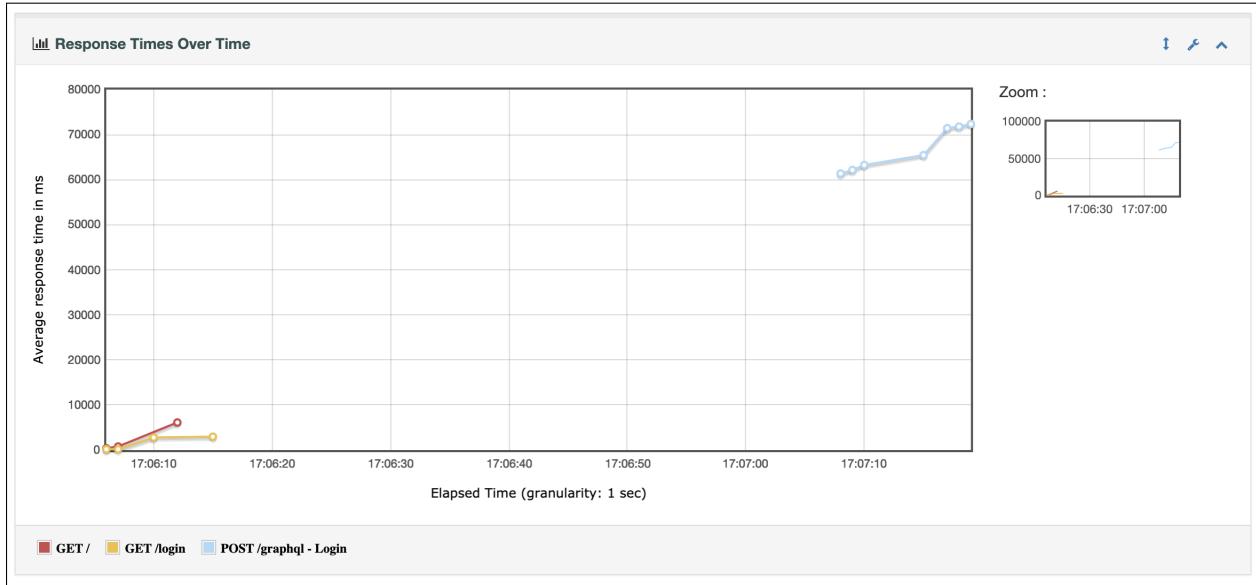


Figura 14: Response Times Over Time do *login* para 500 threads

#### 4.3.3 Teste 2: Login + Listagem de Páginas + Criação de páginas

Este teste, em comparação com o teste anterior, apresenta uma maior carga de trabalho para o servidor e envolve fazer *login* na aplicação, listar as páginas existentes e criar uma página nova.

Numa primeira fase, foi criado um *script* com o *Selenium* que simulava o comportamento descrito anteriormente. No entanto, constatou-se que o teste realizado com o *Selenium* não era muito viável para um número de *threads* maior que 4 devido ao tempo de espera de carregamento das páginas. As funções que o *Selenium* dispõe para as esperas de carregamento de páginas estão descontinuadas logo, a forma para realizar esta espera era com esperas activas, o que torna a medição do desempenho pouco significativa uma vez que as esperas do carregamento das páginas iriam estar limitadas pelo maior tempo de espera. No entanto, este teste realizado deu para constatar, visualmente, que o servidor estava a demorar um tempo considerável a apresentar as páginas aos utilizadores. Dado isto, decidiu-se apresentar apenas os resultados obtidos pelo *JMeter* pois foram considerados mais significativos em relação ao *Selenium*.

Para simular o comportamento deste teste, foi criado um plano de testes que consiste em sete pedidos ao servidor como se pode observar nas imagens posteriores.

Para 100 *threads*, todas as páginas submetidas para criação foram efectivamente criadas mas, no entanto, nem todas as páginas conseguiram ser apresentadas aos utilizadores. O tempo médio de espera, neste caso, é de 20 segundos, o que é bastante significativo.

Ao executar o teste para o restante número de *threads* verificou-se, novamente um comportamento anormal com o *JMeter*. Em todos os testes, à excepção do primeiro, a percentagem de erros na criação de páginas foi de 100%, mas, no entanto, após interrogar a base de dados sobre o número de páginas criadas, observou-se que efectivamente houveram páginas a ser criadas. Para 200 *threads*, foram criadas, como era esperado, 200 páginas, mas, para cada um dos testes posteriores, o número de páginas efectivamente criadas foi, em média, 180.

Posto isto, observa-se uma degradação considerável no desempenho do servidor e um aumento do tempo de resposta dado o número crescente de utilizadores, que no pior registo foi de 2 minutos. Pode-se constatar que o servidor apenas está a conseguir criar uma média de 180 páginas quando o número de pedidos é elevado o que faz com que o sistema esteja a disponibilizar um serviço com baixa disponibilidade.

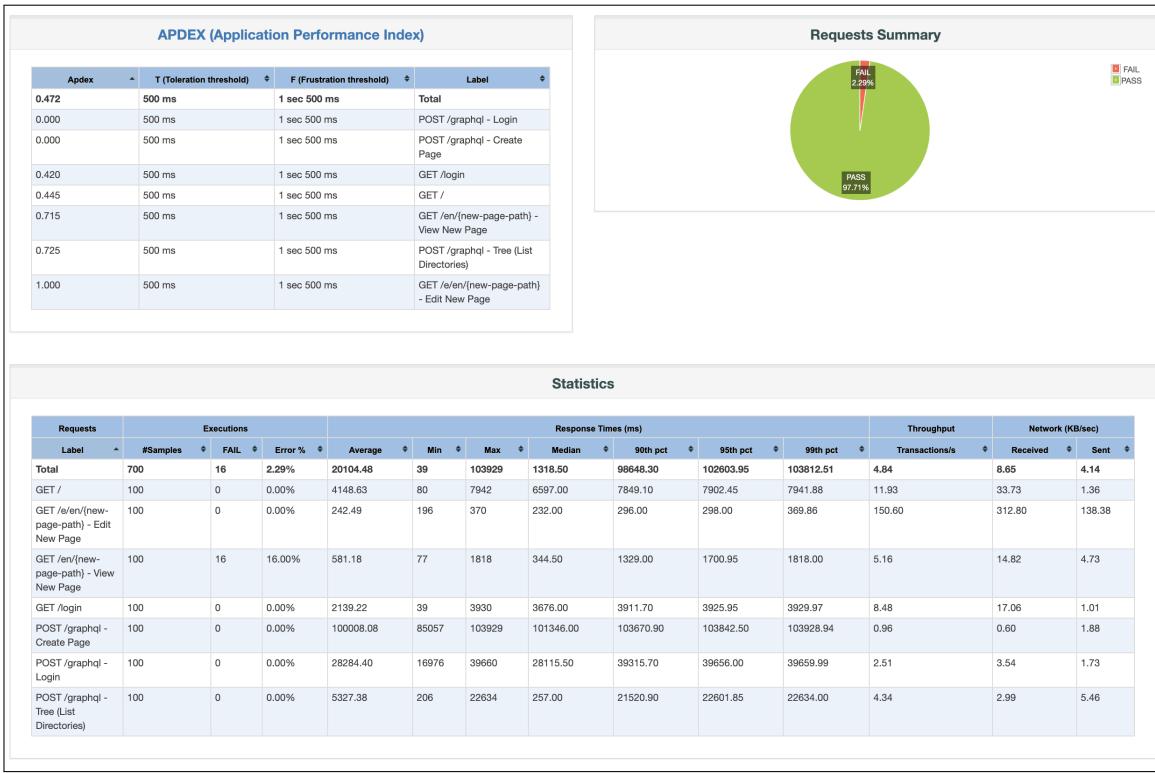


Figura 15: Estatísticas do Teste 2 para 100 threads

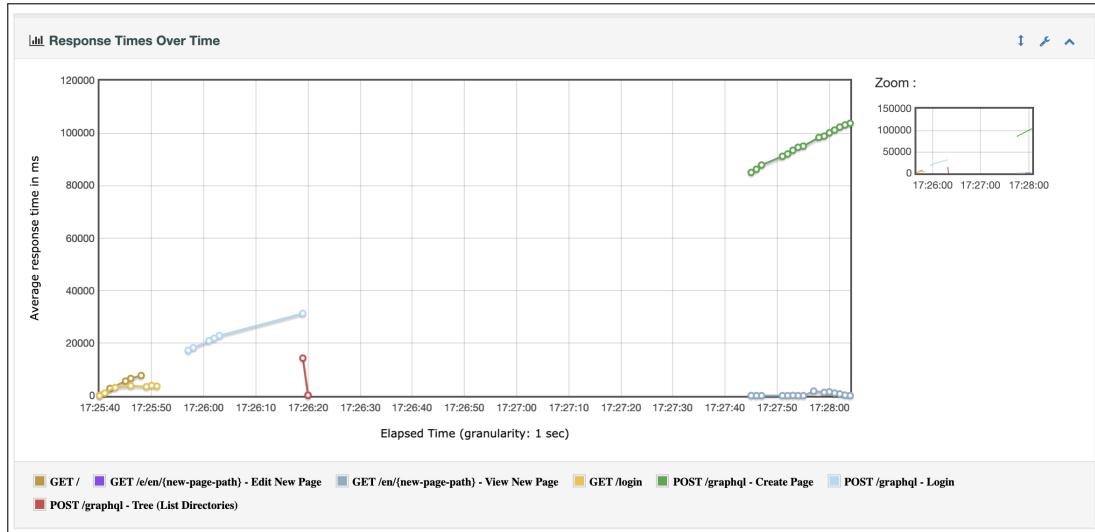


Figura 16: Response Times Over Time do Teste 2 para 100 threads

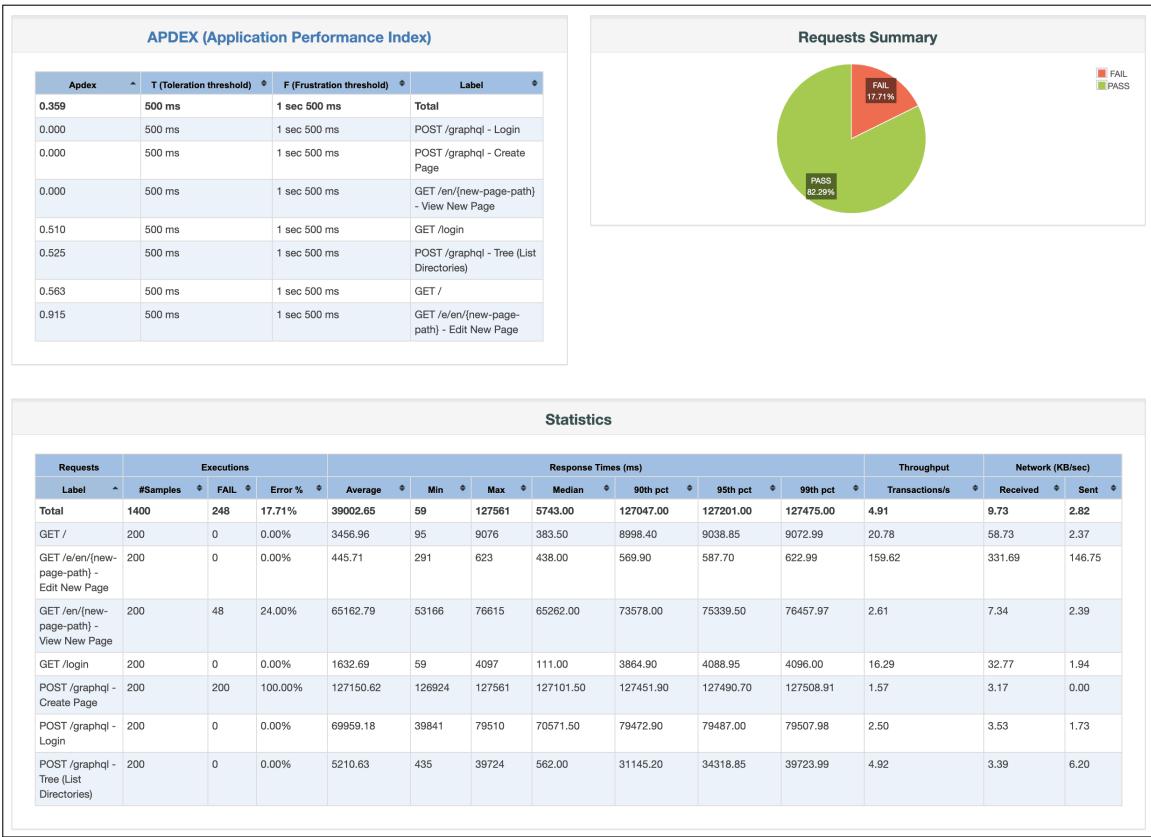


Figura 17: Estatísticas do Teste 2 para 200 threads

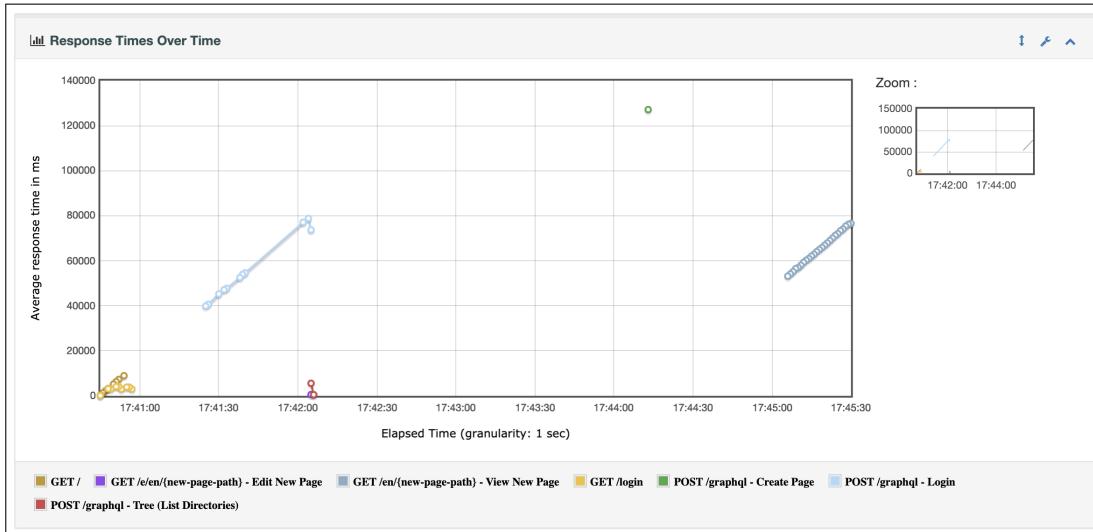


Figura 18: Response Times Over Time do Teste 2 para 200 threads

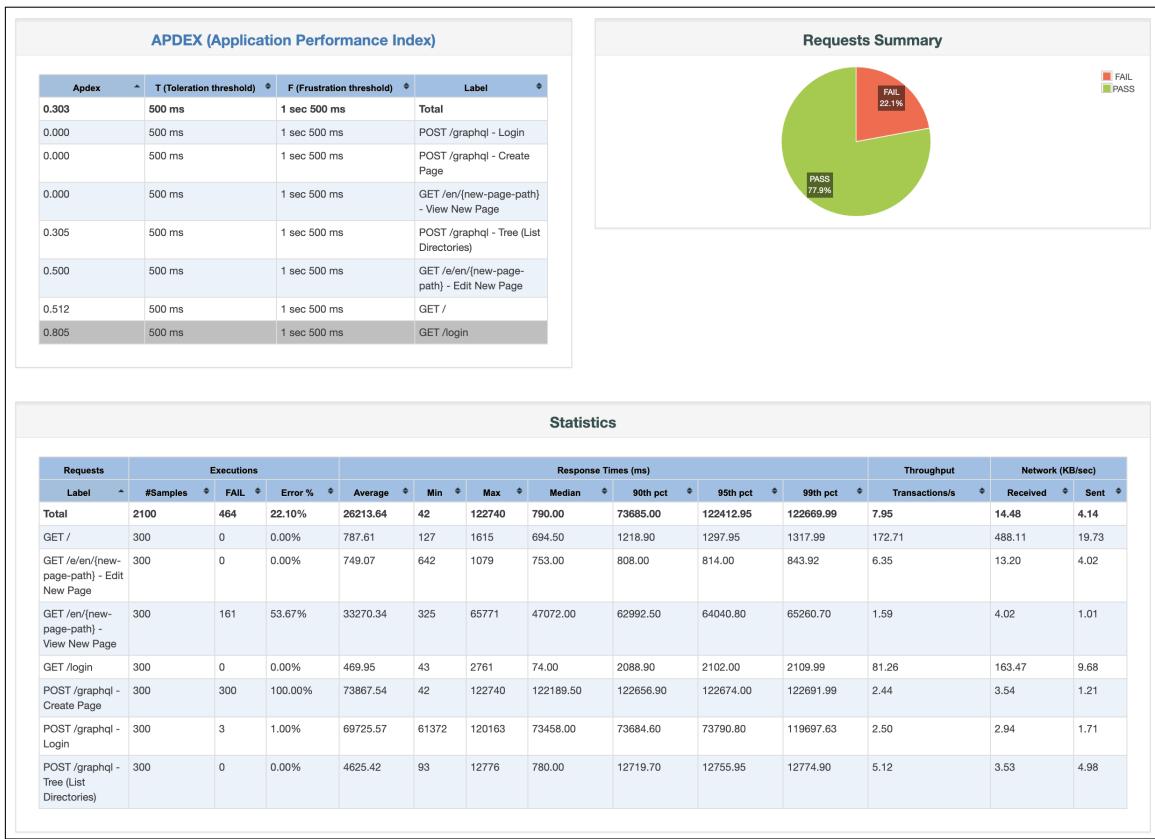


Figura 19: Estatísticas do Teste 2 para 300 threads

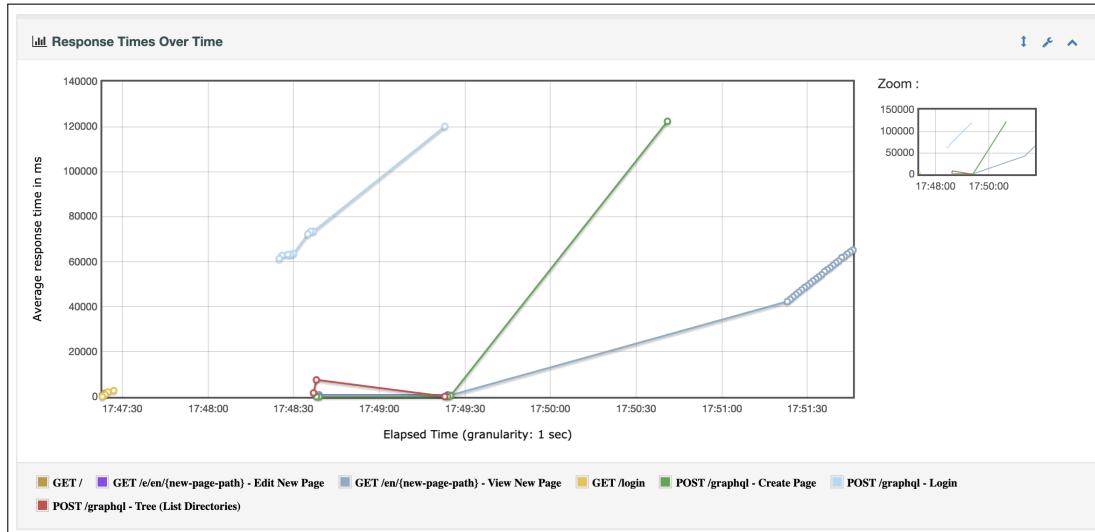


Figura 20: Response Times Over Time do Teste 2 para 300 threads

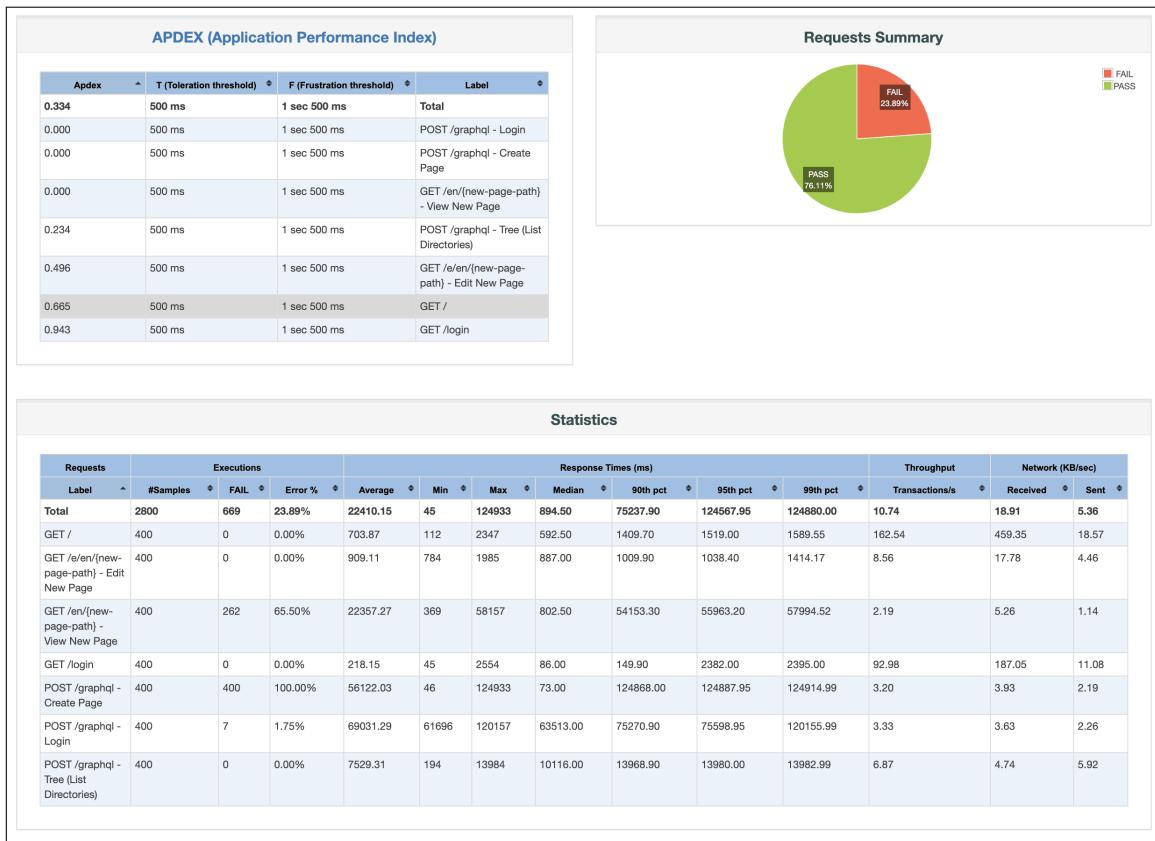


Figura 21: Estatísticas do Teste 2 para 400 threads

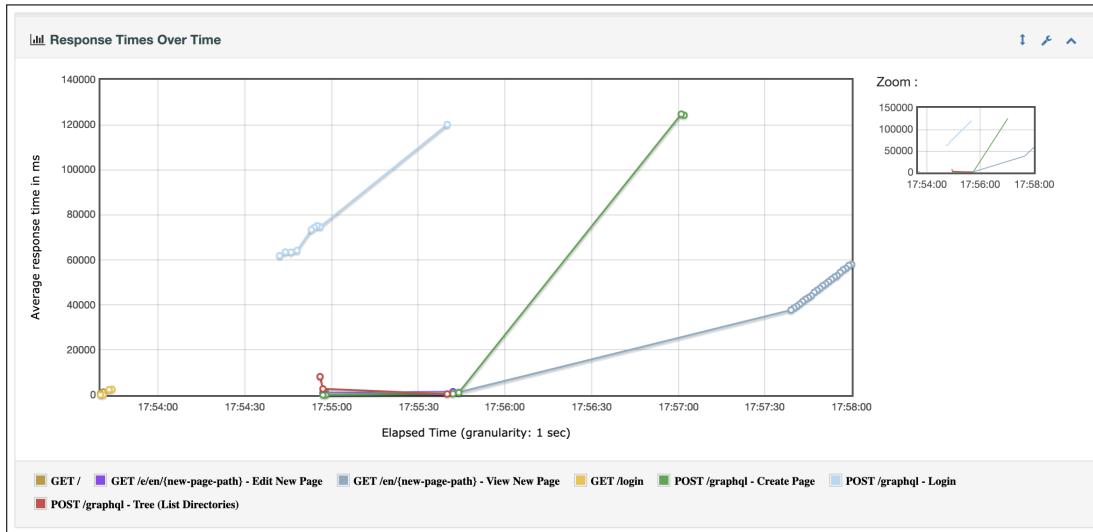


Figura 22: Response Times Over Time do Teste 2 para 400 threads

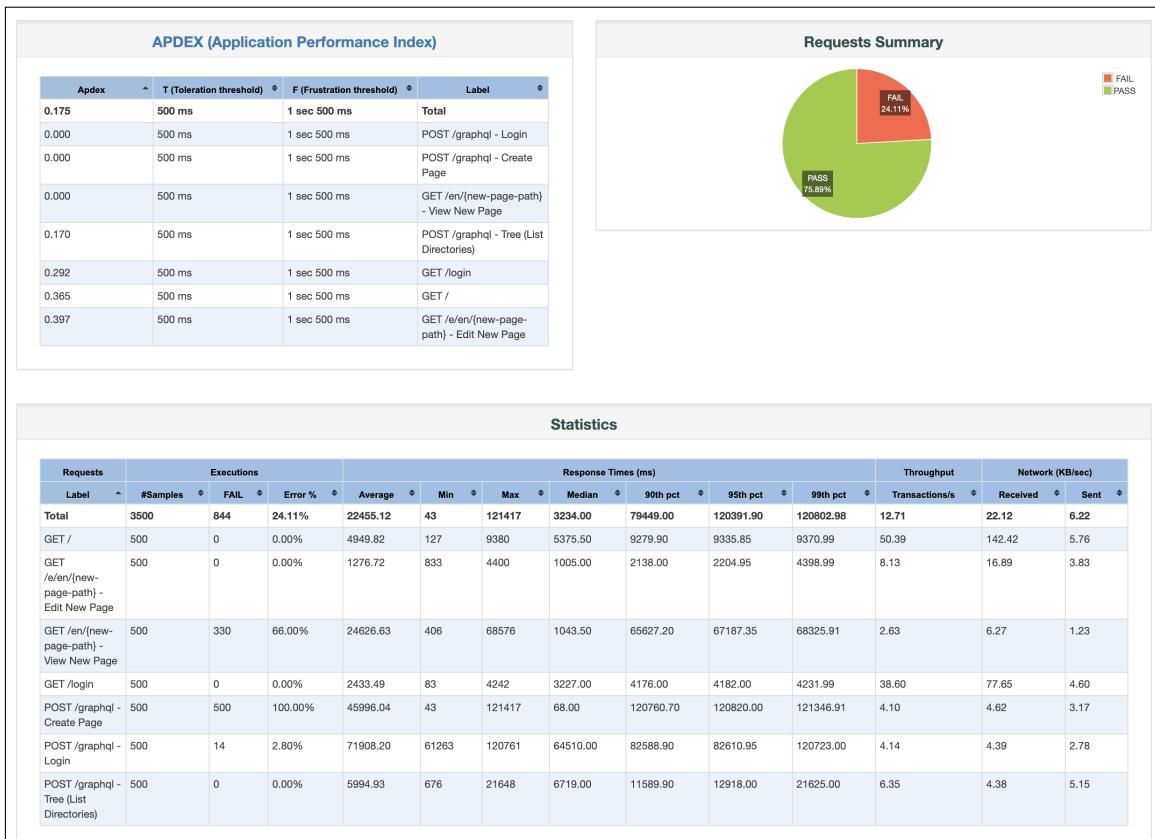


Figura 23: Estatísticas do Teste 2 para 500 threads

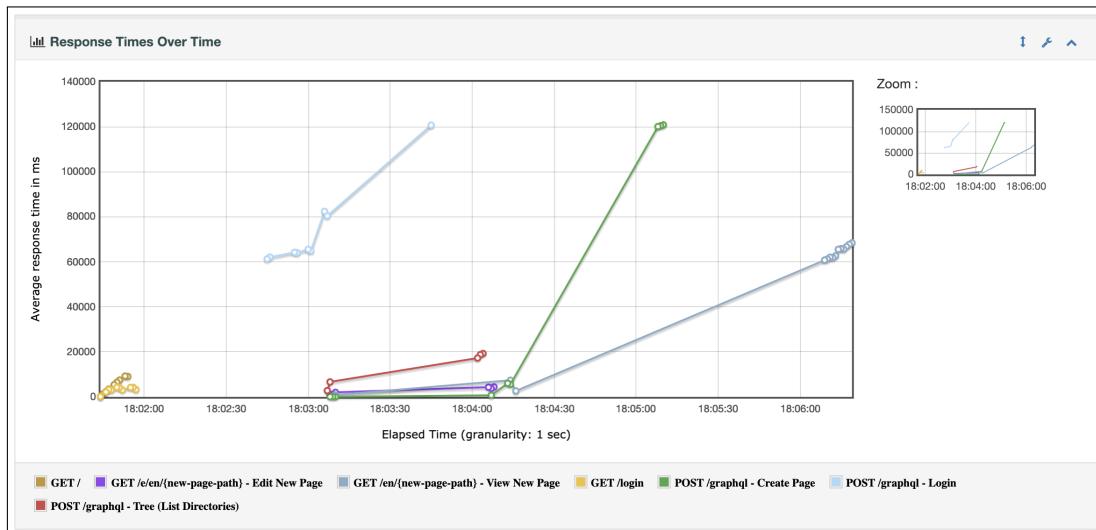


Figura 24: Response Times Over Time do Teste 2 para 500 threads

#### 4.3.4 Teste 3: Login + Pesquisa + Obtenção da Página Pesquisada

Em seguida, foi decidido utilizar a ferramenta *JMeter* para efectuar testes de carga. Este teste consistia em entrar na página inial do *Wikijs* e, em seguida, efectuar o *login* na aplicação, fazer a pesquisa de uma página e por fim obter a mesma página. Os pedidos feitos à aplicação estão apresentados na seguinte imagem:

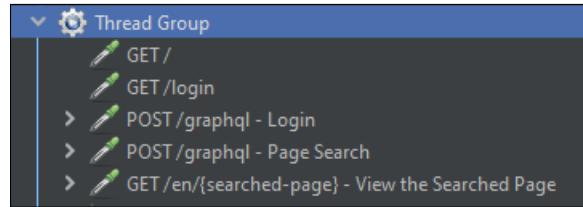


Figura 25: Pedidos feitos no teste 3.

O teste foi executado para 100, 200, 300, 400 e 500 *threads* aos quais os resultados estão apresentados nas figuras 26, 27, 28, 29 e 30 respectivamente.

Analisando estes resultados, podemos concluir que a nossa aplicação consegue responder às quantidades de pedidos exigidas com os vários números de *threads*, havendo uma pequena percentagem de erro apenas com 400 e 500 *threads* no *POST* do *LogIn*.

Por outro lado, apresenta valores baixos *throughput* e valores bastante altos de *response time* o que demonstra que o servidor apesar de responder apresenta dificuldades a processar uma elevada quantidade de pedidos.

Como se pode reparar nas várias figuras o valor médio de *response time* mantém-se entre os 15000 e os 16000 ms a partir de 200 *threads* sendo que os *POSTS* apresentam um elevado peso nestes valores devido a obter maiores valores de *response time*, como por exemplo o *POST LogIn* pode ultrapassar os 70000 ms o que é um tempo enorme.

O *throughput* sobe consoante o número de *threads* sendo que novamente os pedidos *POST* influenciam a média destes valores devido a obterem valores bastante baixos.

É se esperado diminuir os valores de *response time* e aumentar os valores de *throughput* com a nova arquitetura.

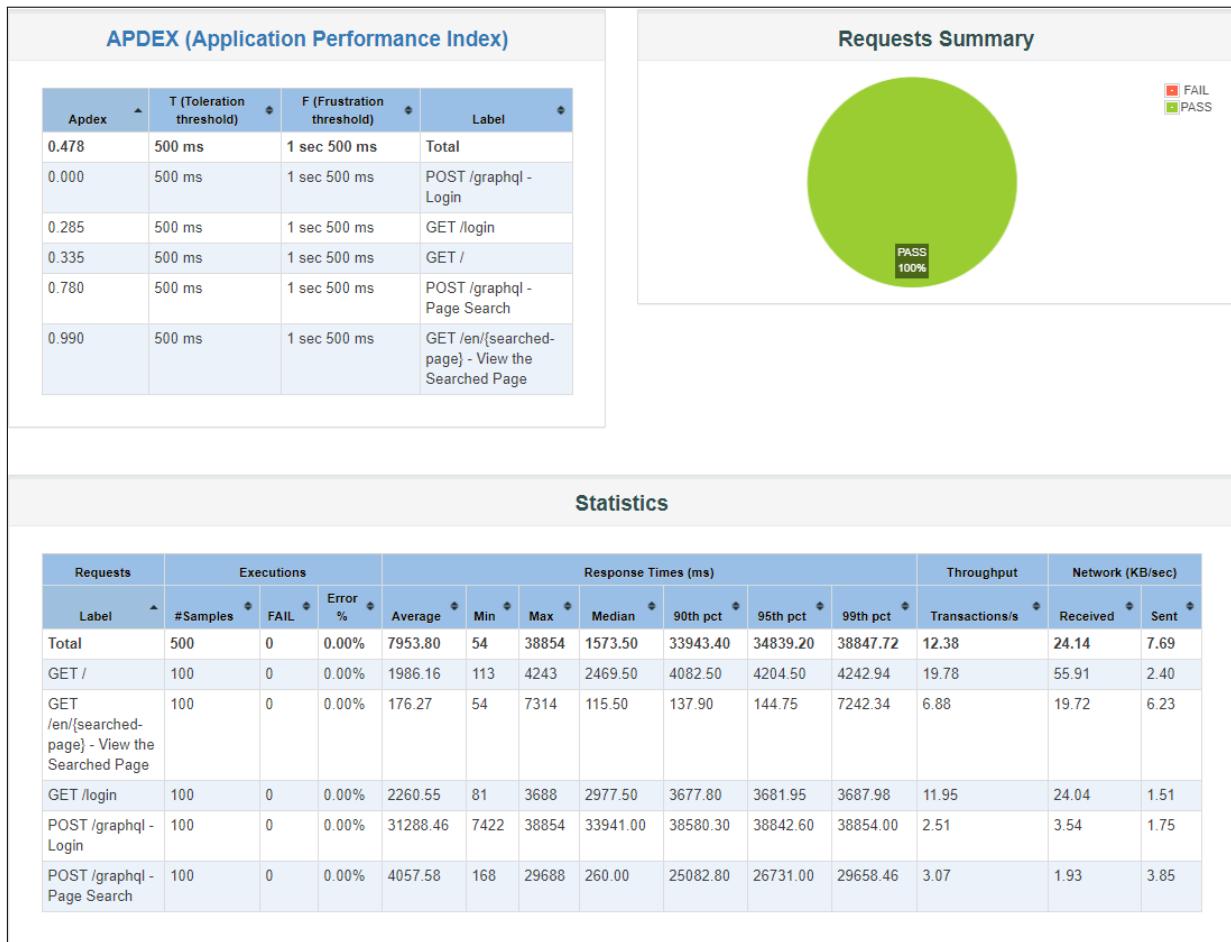


Figura 26: Estatísticas do Teste 3 para 100 threads

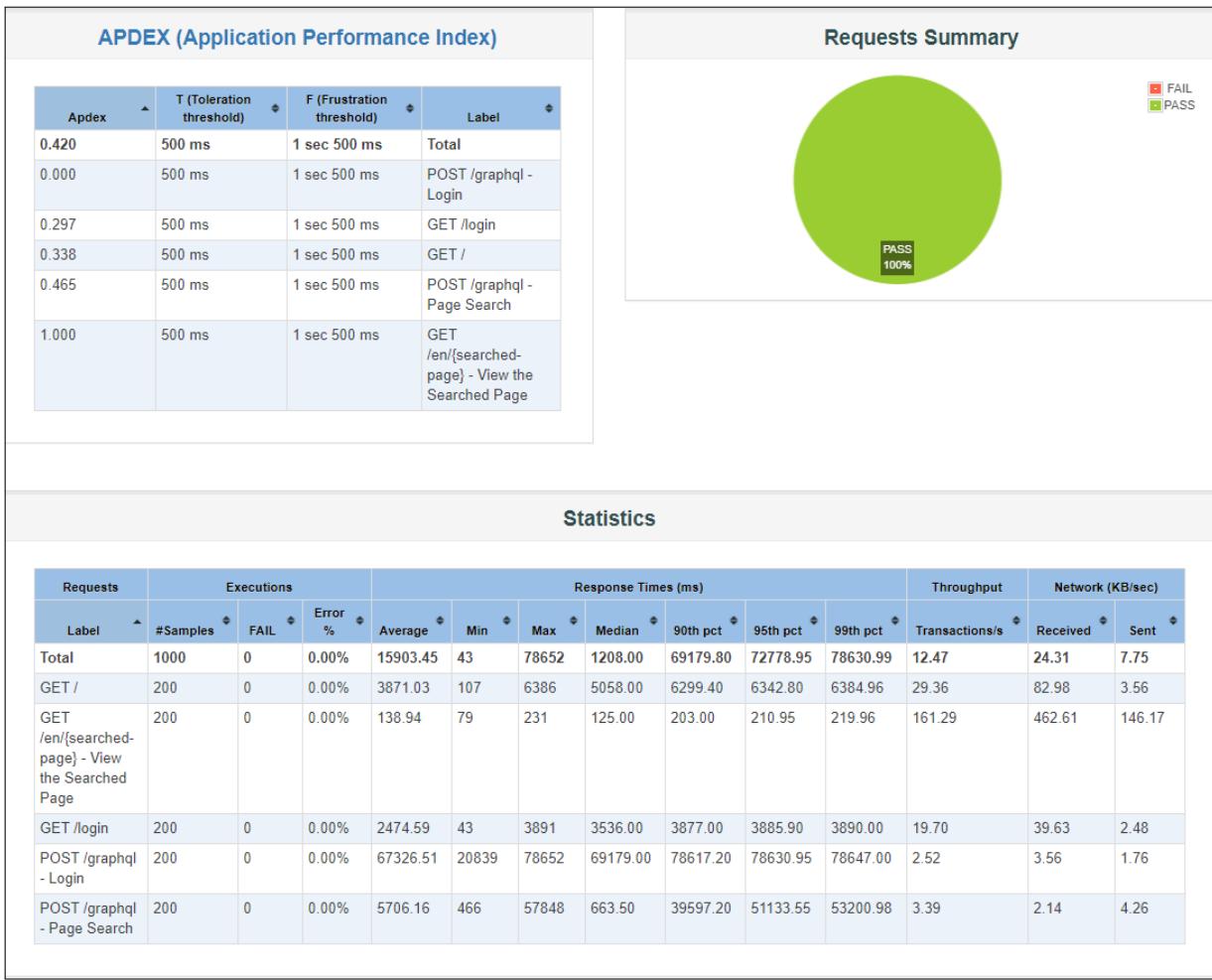


Figura 27: Estatísticas do Teste 3 para 200 threads

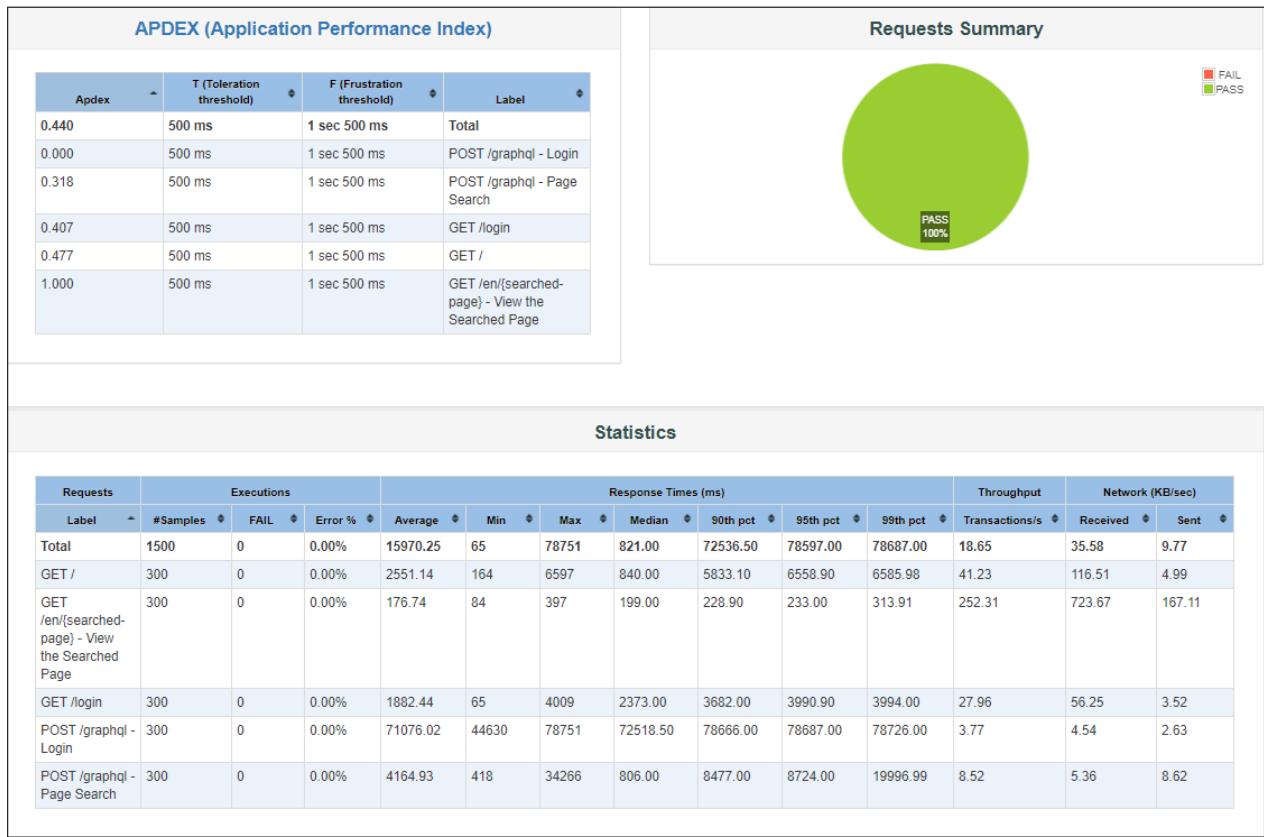


Figura 28: Estatísticas do Teste 3 para 300 threads

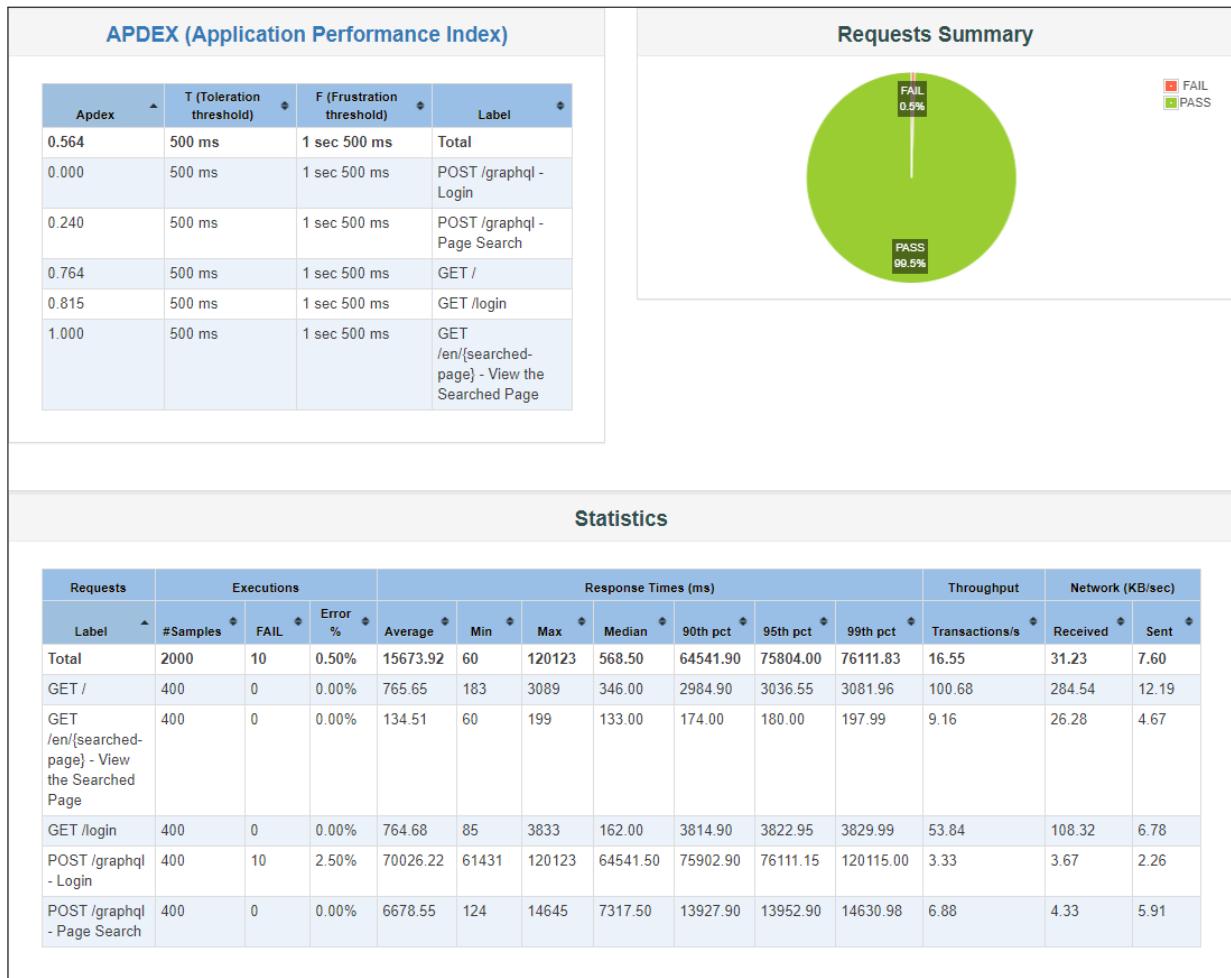


Figura 29: Estatísticas do Teste 3 para 400 threads

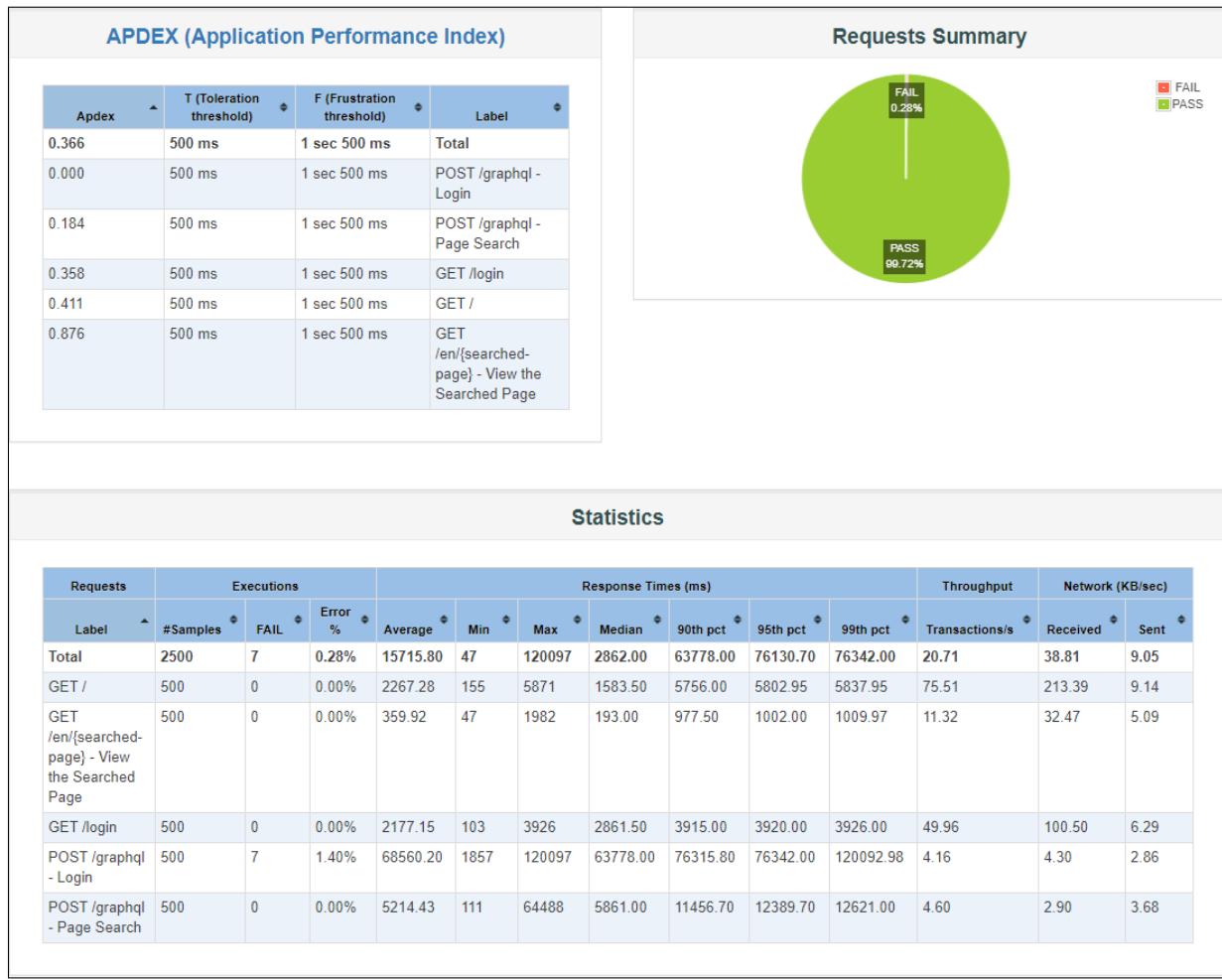


Figura 30: Estatísticas do Teste 3 para 500 threads

#### 4.3.5 Teste 4: Login + Criação de Utilizador

Foi novamente utilizada a ferramenta *JMeter* para efectuar testes de carga. Este teste consiste em entrar na página inicial do *Wikijs* e, em seguida, efectuar o *login* na aplicação como administrador, entrar no seu menu que corresponde a processos de obtenção de grupos e utilizadores e por fim adicionar um utilizador ao sistema. Cada novo utilizador vai conter um nome e um email derivado do número da *thread* e do tempo em que foi executado o pedido com auxílio das funções do *Jmeter* *\_\_time()* e *\_\_threadNum()*. Os pedidos feitos à aplicação podem ser visualizados na imagem a baixo:

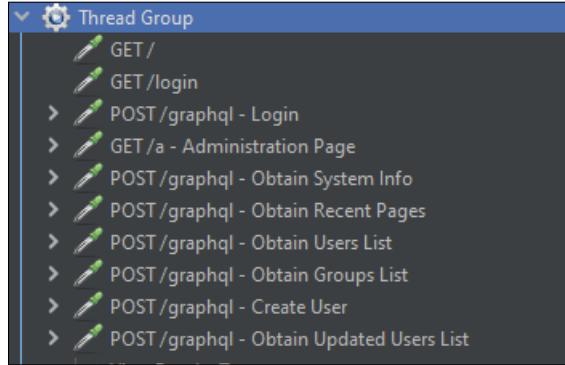


Figura 31: Pedidos feitos no teste 4.

O teste foi executado para 100, 200, 300, 400 e 500 *threads* aos quais os resultados estão apresentados nas figuras 32, 33, 34, 35 e 36 respectivamente.

Através destes testes, podemos concluir que a nossa aplicação tem bastantes dificuldades na criação de utilizadores. Apesar de entre 100 e 200 *threads* não apresentar erros, quanto aumentamos a carga sobre o servidor com mais de 200 *threads*, o servidor fica *overbooked* e não consegue processar os pedidos. Como se pode reparar nas figuras com 300 *threads* apresenta uma percentagem de erro de 19.73% e 25% e 21,78% com 400 e 500 *threads* respectivamente. Estes erros caem sobre os pedidos *POST* ao que por consequência, uma elevada quantidade de utilizadores não foram criados como pretendido .

Como se pode reparar nas várias figuras o pedido para efectuar o *POST* do *LogIn* é novamente aquele que mais influencia o tempo de resposta médio tendo em média 70000 ms de *response time*. O *throughput* sobe consoante o número de *threads* sendo que novamente os pedidos *POST* influenciam a média destes valores devido a obterem valores bastante baixos.

Espera-se diminuir os valores de *response time* e aumentar os valores de *throughput* com a nova arquitetura e também diminuir a quantidade de erros para mais utilizadores serem criados com estes testes.

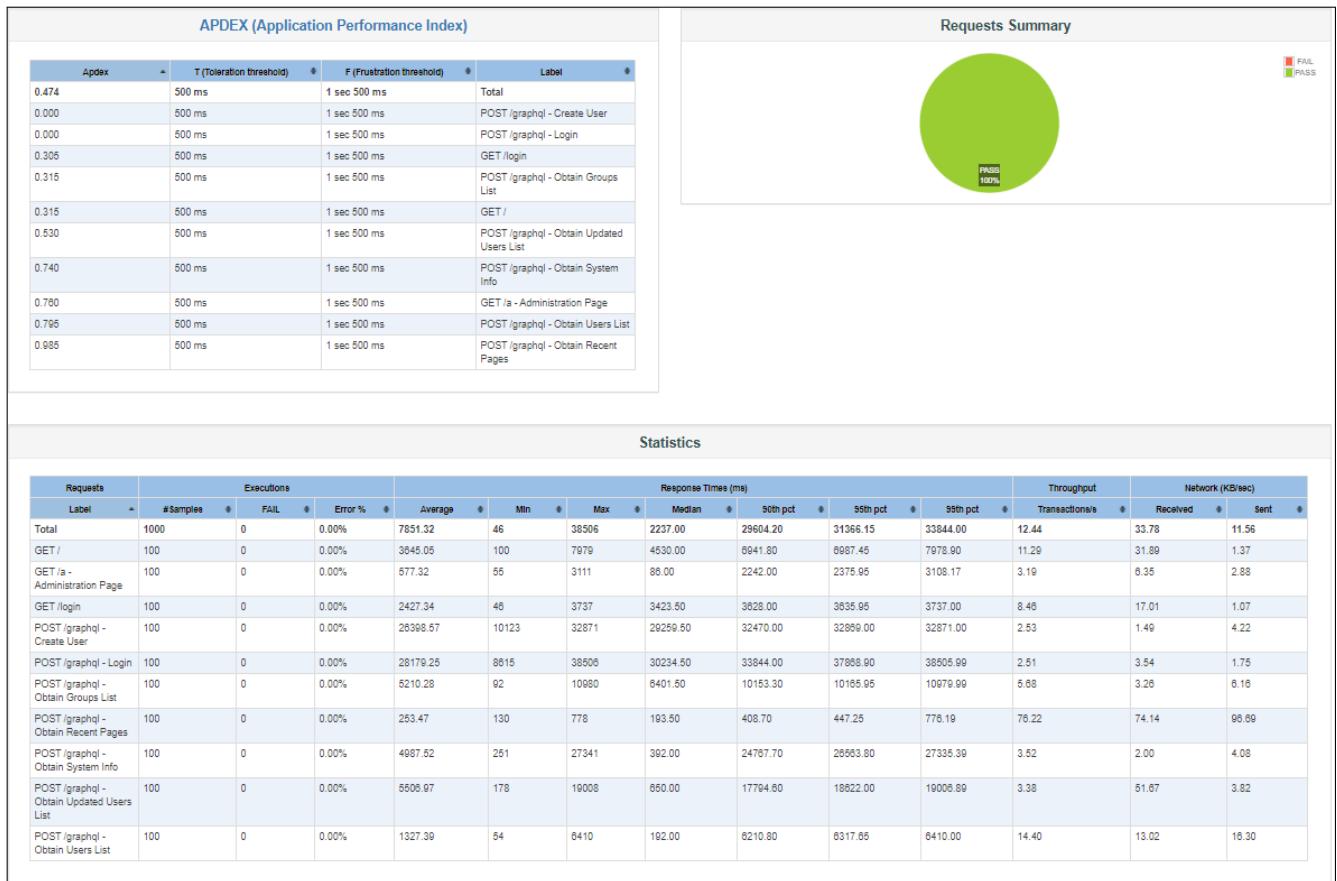


Figura 32: Estatísticas do Teste 4 para 100 threads

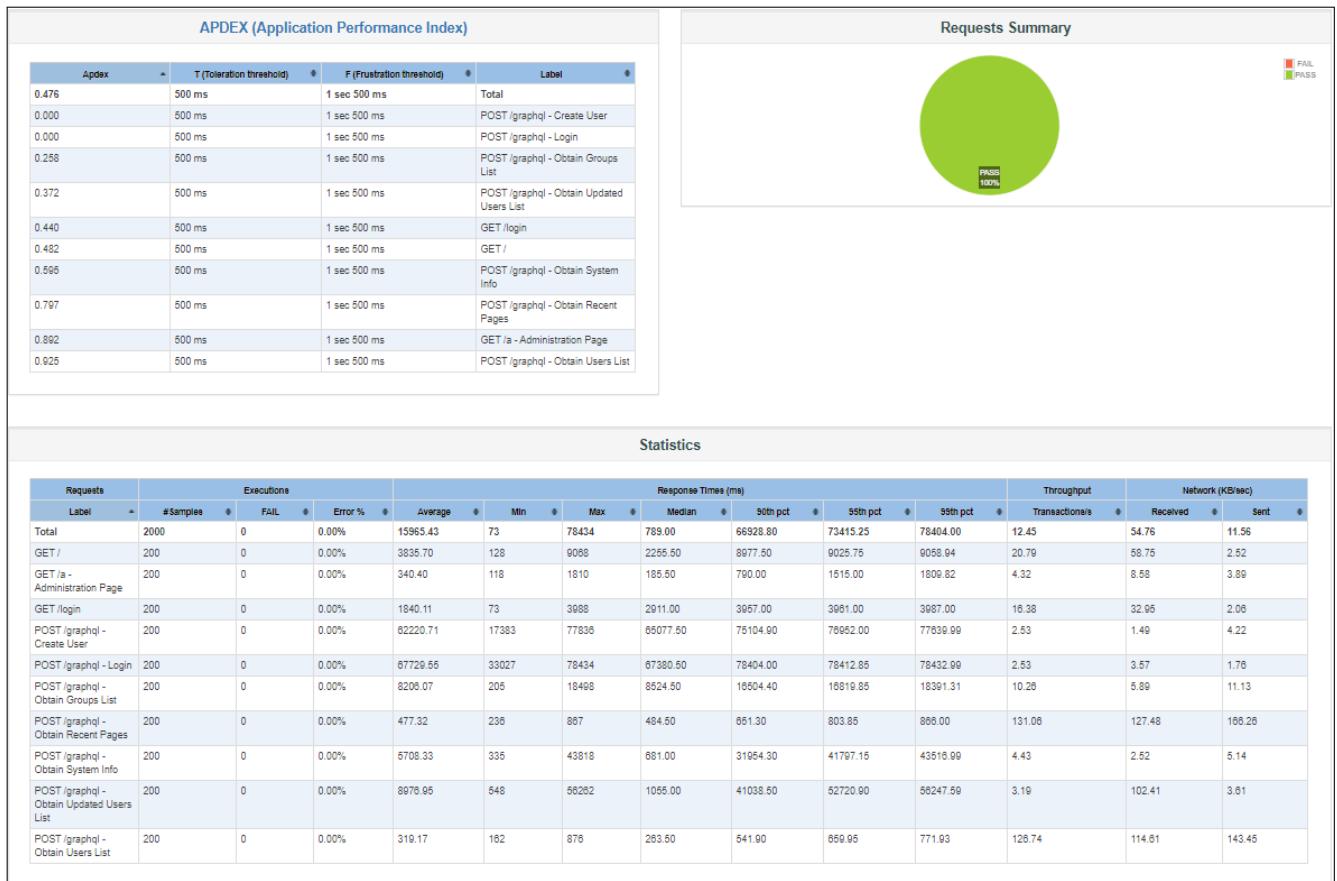


Figura 33: Estatísticas do Teste 4 para 200 threads

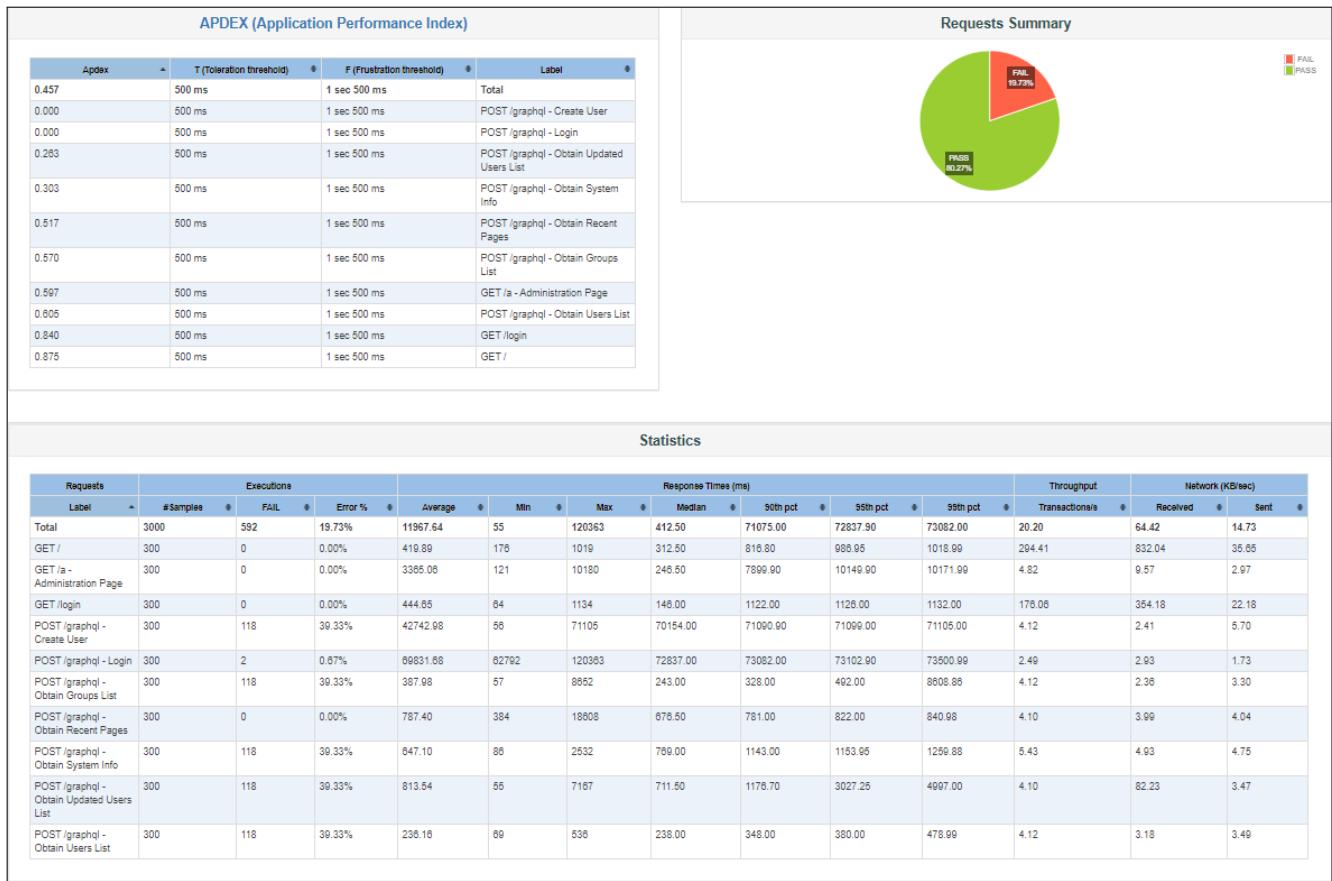


Figura 34: Estatísticas do Teste 4 para 300 threads

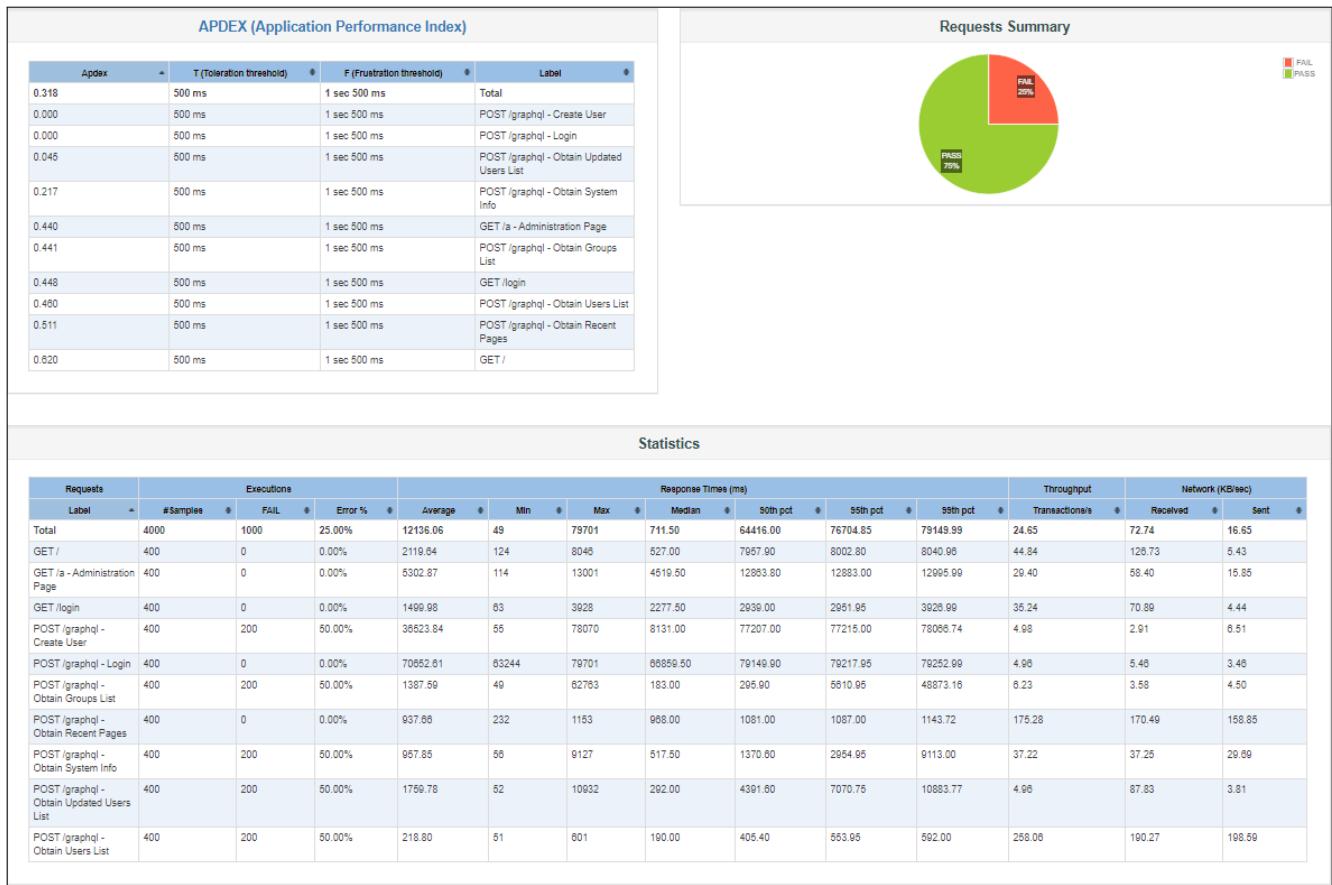


Figura 35: Estatísticas do Teste 4 para 400 threads

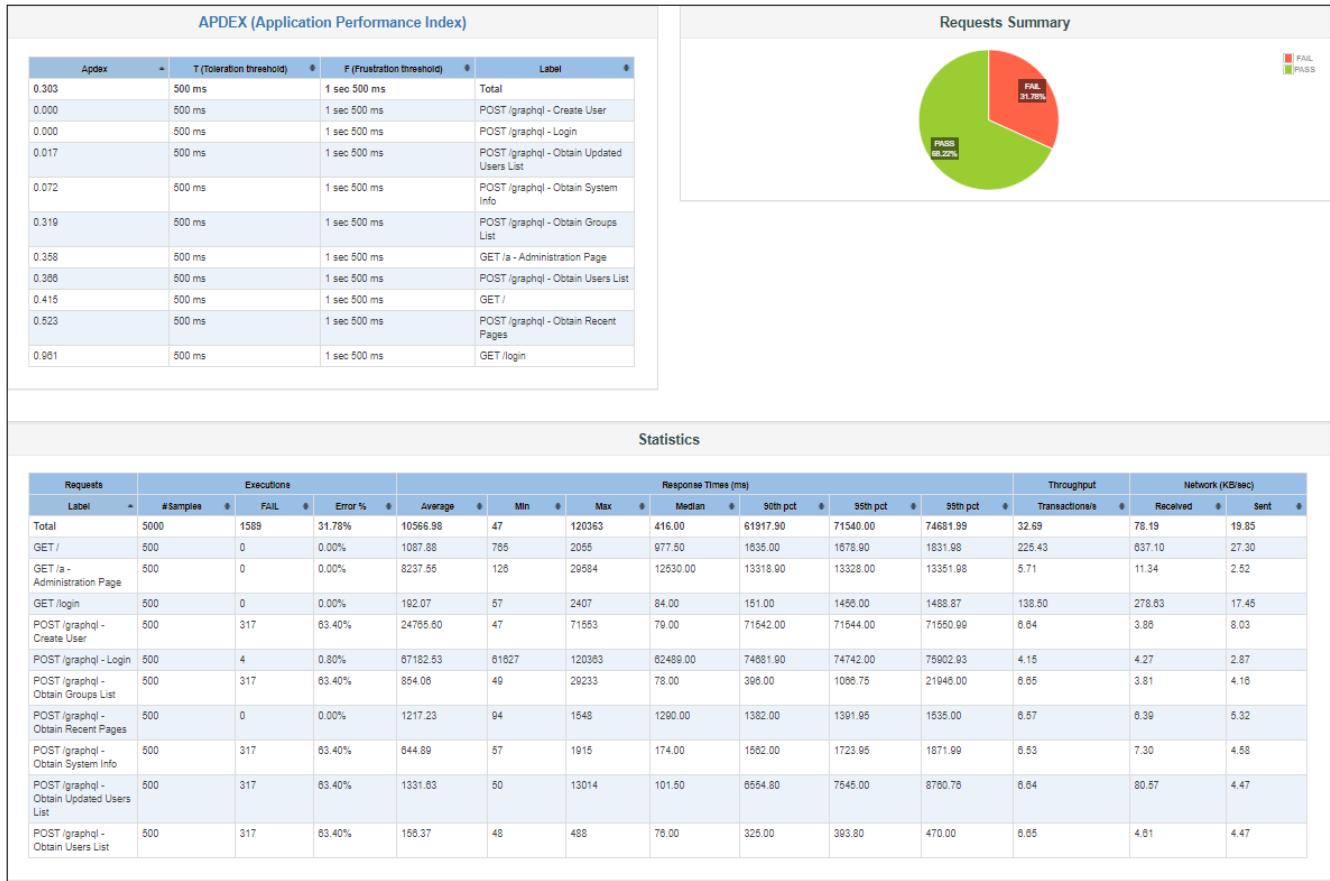


Figura 36: Estatísticas do Teste 4 para 500 threads

#### 4.3.6 Teste 5: Login + Consulta de todas as Páginas do Sistema

Foi novamente utilizada a ferramenta *JMeter* para efectuar testes de carga. Este teste consiste em entrar na página inicial do *Wikijs* e, em seguida, efectuar o *login* na aplicação como administrador, entrar no seu menu que corresponde a processos de obtenção de informação do sistema e por fim obtém todas as páginas existentes na base de dados. Este teste foi feito com mais de 100 páginas existentes no sistema. Os pedidos feitos à aplicação estão demonstrados na imagem a baixo:

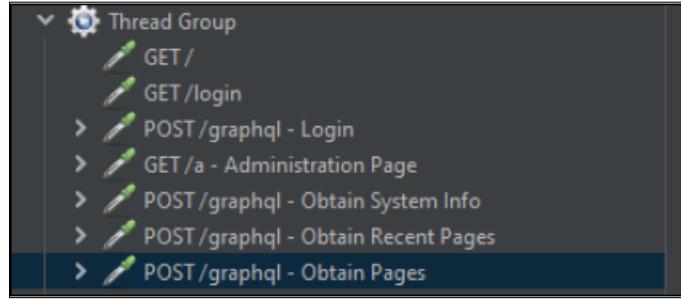


Figura 37: Pedidos feitos no teste 5.

O teste foi executado para 100, 200, 300, 400 e 500 *threads* aos quais os resultados estão apresentados nas figuras 38, 39, 40, 41 e 42 respectivamente.

Como se pode observar pelos resultados obtidos esta arquitetura inicial não tem qualquer problema em suportar a carga de login e consulta de páginas de 100 e 200 threads simultaneamente. No entanto com 300 threads já se podem observar alguns erros (cerca de 6%), sendo que grande parte destes erros ocorrem não durante a operação de login, mas sim durante a operação de obtenção das informações do sistema, nomeadamente número de páginas, número de utilizadores, etc. Com 400 e 500 threads conseguimos observar que mais uma vez a percentagem de erros aumenta uma vez que existe mais sobrecarga sobre o nosso sistema, sendo que a maior fatia da percentagem de erro acontece sempre na obtenção da informação do sistema.

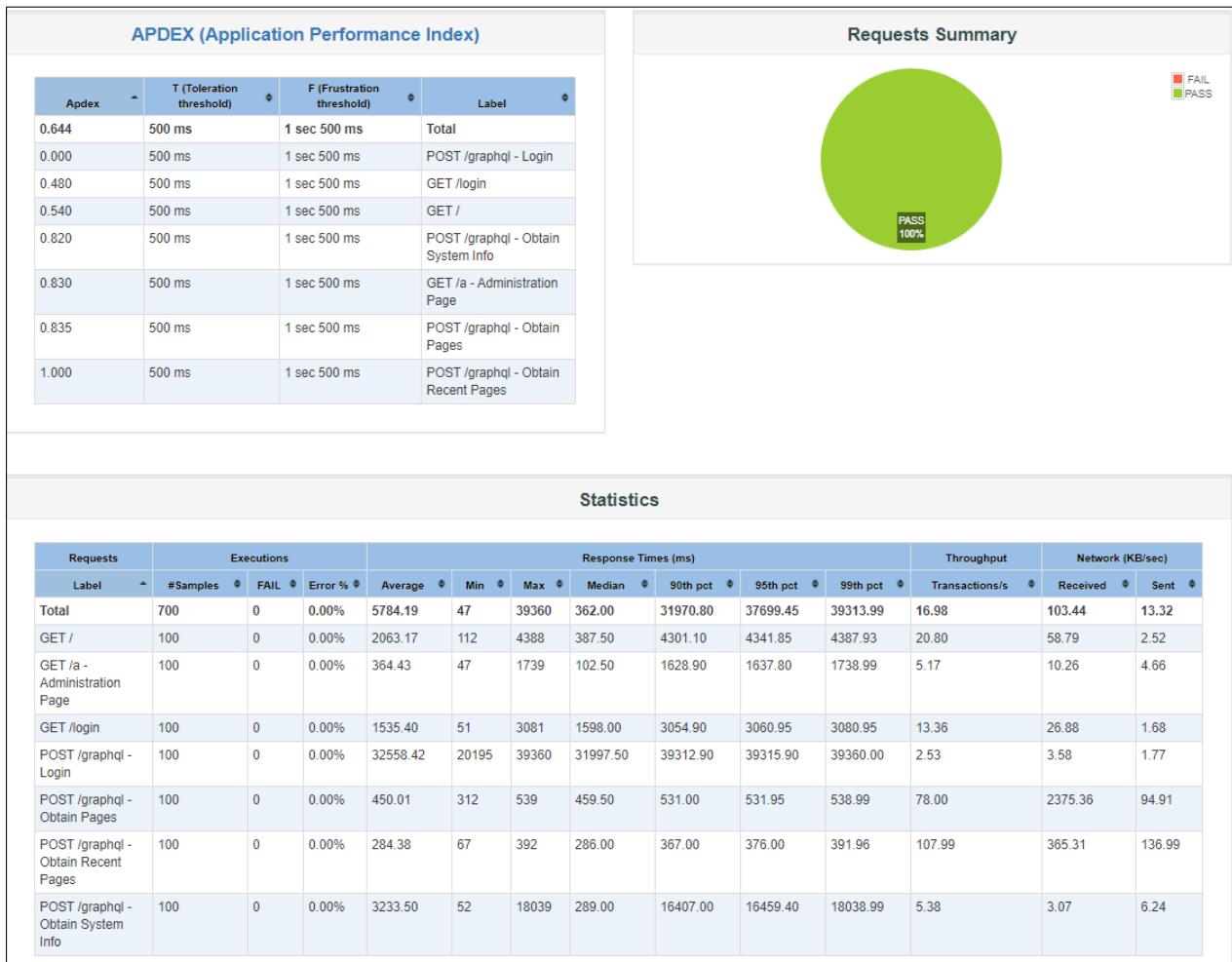


Figura 38: Estatísticas do Teste 5 para 100 threads

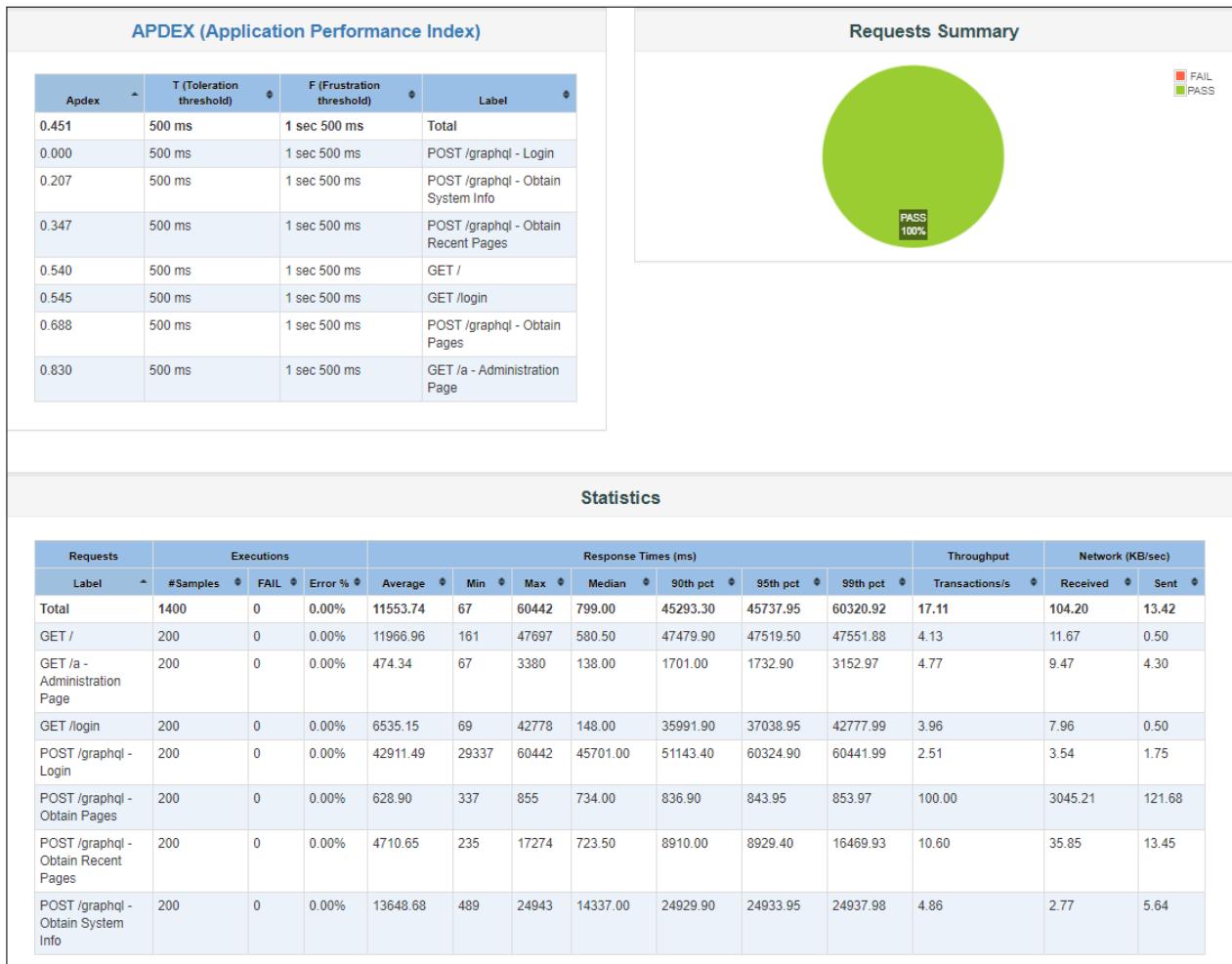


Figura 39: Estatísticas do Teste 5 para 200 threads

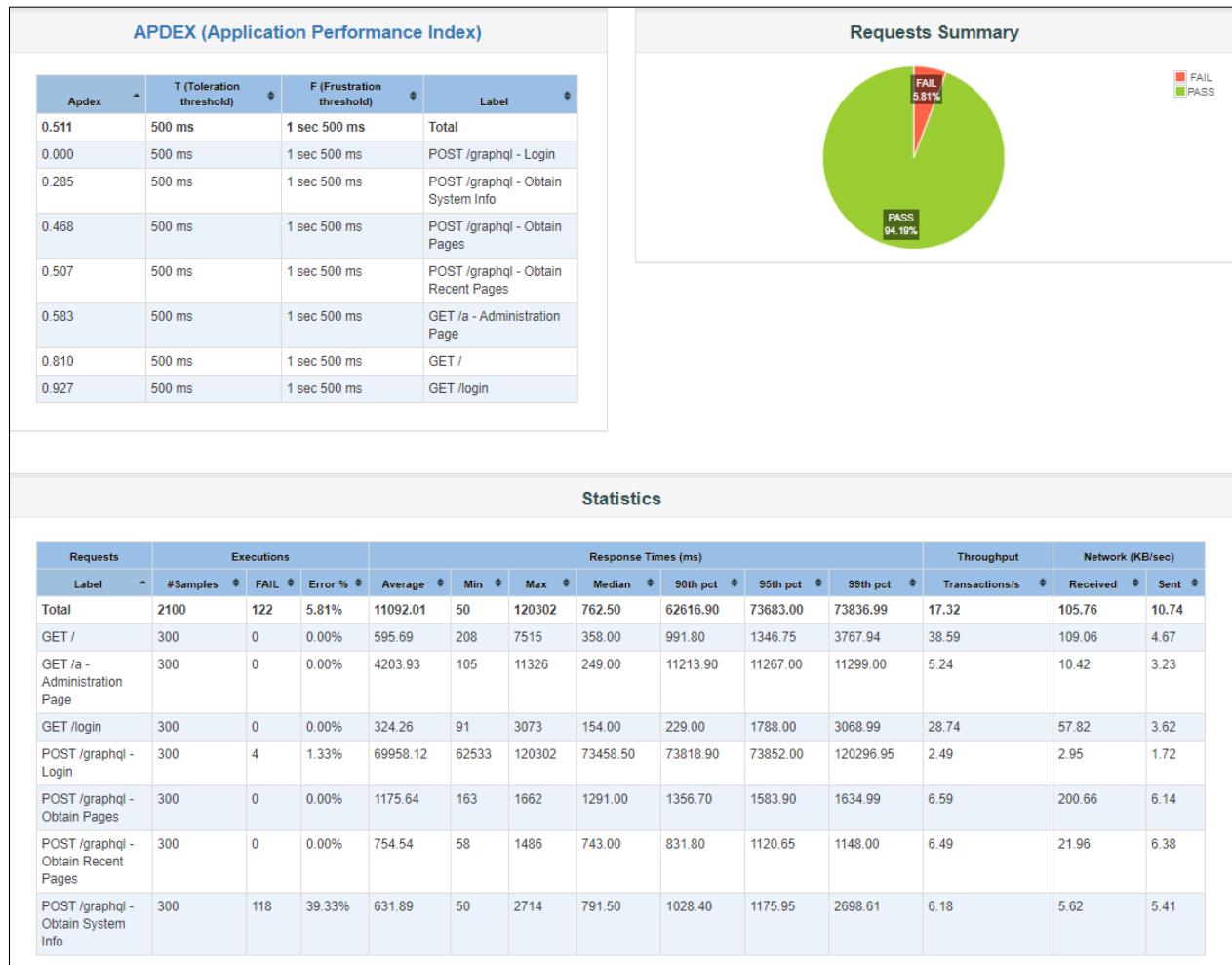


Figura 40: Estatísticas do Teste 5 para 300 threads

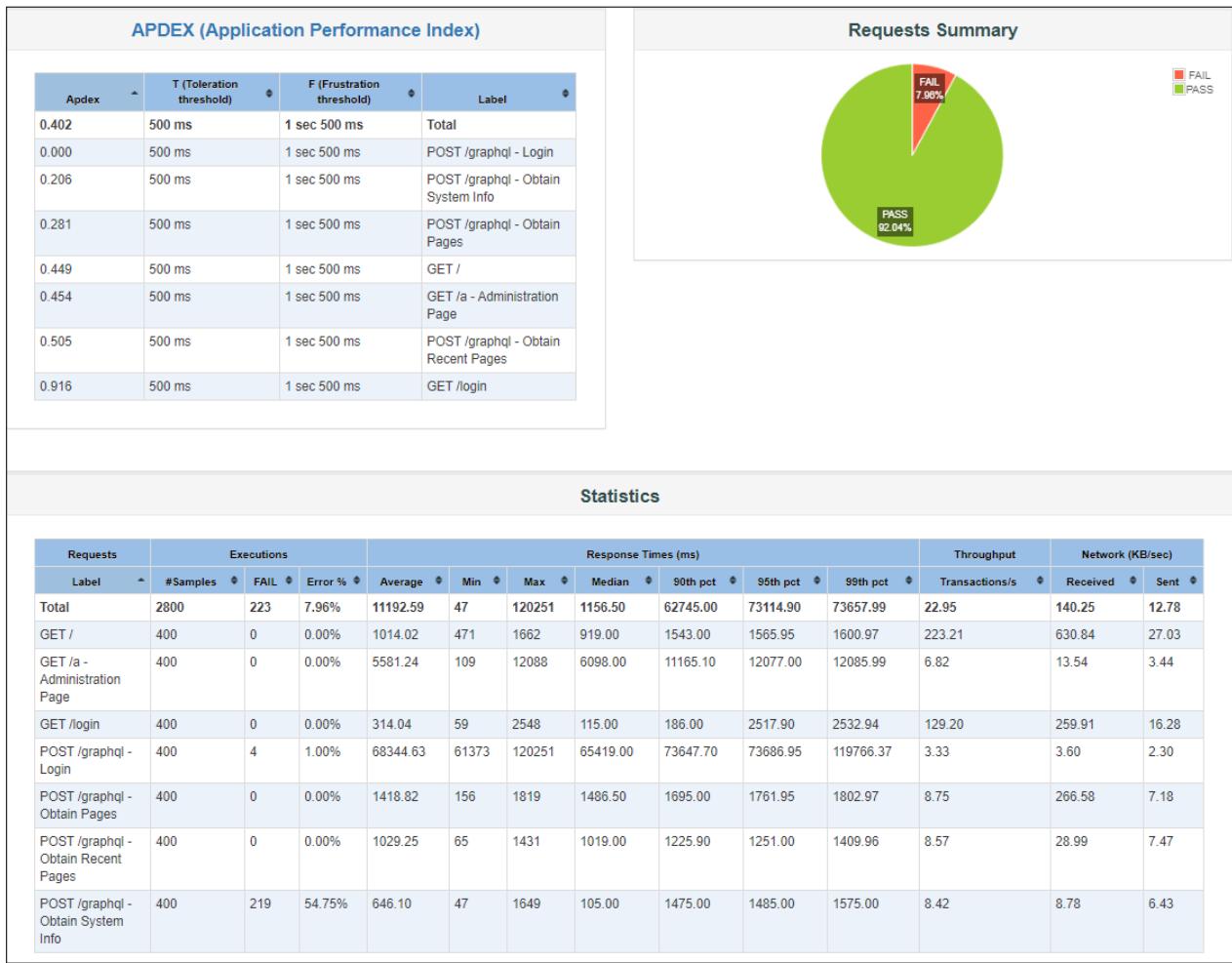


Figura 41: Estatísticas do Teste 5 para 400 threads

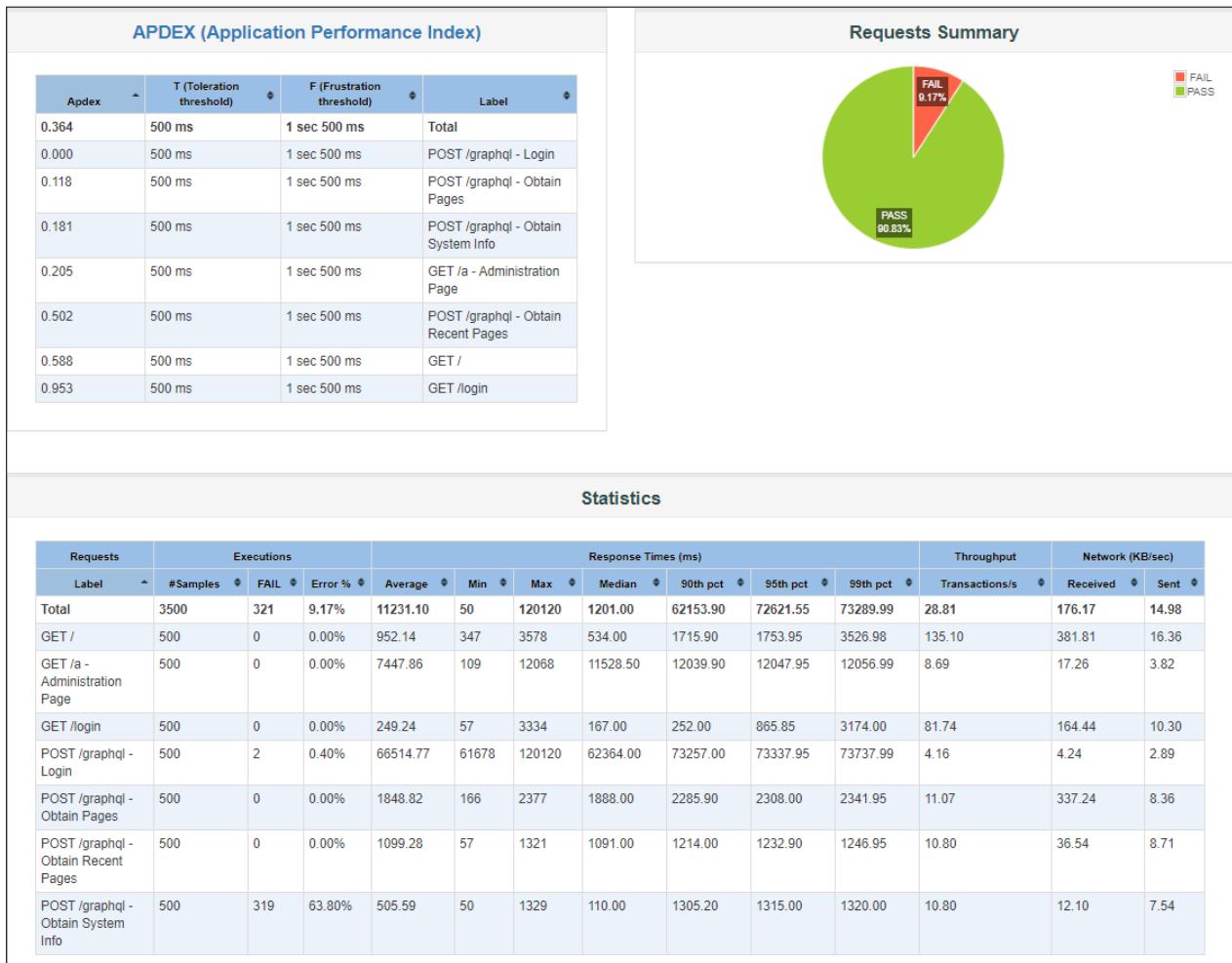


Figura 42: Estatísticas do Teste 5 para 500 threads

#### 4.3.7 Teste 6: Login + Pesquisa + Escolha de Página + Save

Por fim, através da ferramenta *JMeter* foi feito o último teste que consiste em entrar na página inicial do *Wikijs* e, em seguida, efectuar o *login* na aplicação, pesquisar por uma página e no fim editar a mesma. Os pedidos feitos à aplicação podem ser vistos na seguinte imagem:

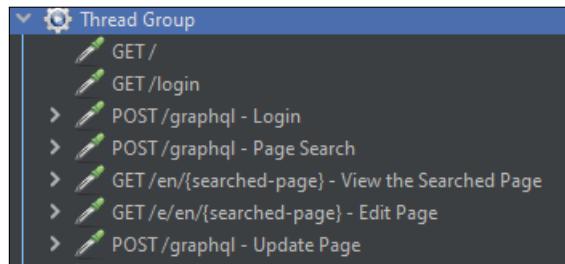


Figura 43: Pedidos feitos no teste 6.

O teste foi executado para 100, 200, 300, 400 e 500 *threads* aos quais os resultados estão apresentados nas figuras 44, 45, 46, 47 e 48 respectivamente.

Neste teste consegue-se verificar que para 100 threads não existe qualquer problema na pesquisa e na edição da página, sendo que a percentagem de erro neste caso é de 0%.

Com 200 threads, já se consegue observar uma percentagem de erro de 2% na parte de dar update à página editada (POST). Este comportamento é considerado normal devido à concorrência que está a existir nas escritas na base de dados.

Para 300 threads observamos mais uma vez um aumento da percentagem de erro, desta vez de 5.71% e toda esta percentagem de erro é observada sobre o *save* da página editada.

Com 400 e 500 threads, observa-se exatamente o mesmo comportamento, tendo-se obtido uma percentagem de erro de 7.9% e 9.2% respetivamente. Espera-se que com a implementação da solução de alta disponibilidade estas percentagens possam ser reduzidas, uma vez que a operação de edição de páginas é uma operação crucial na aplicação do Wiki.js.

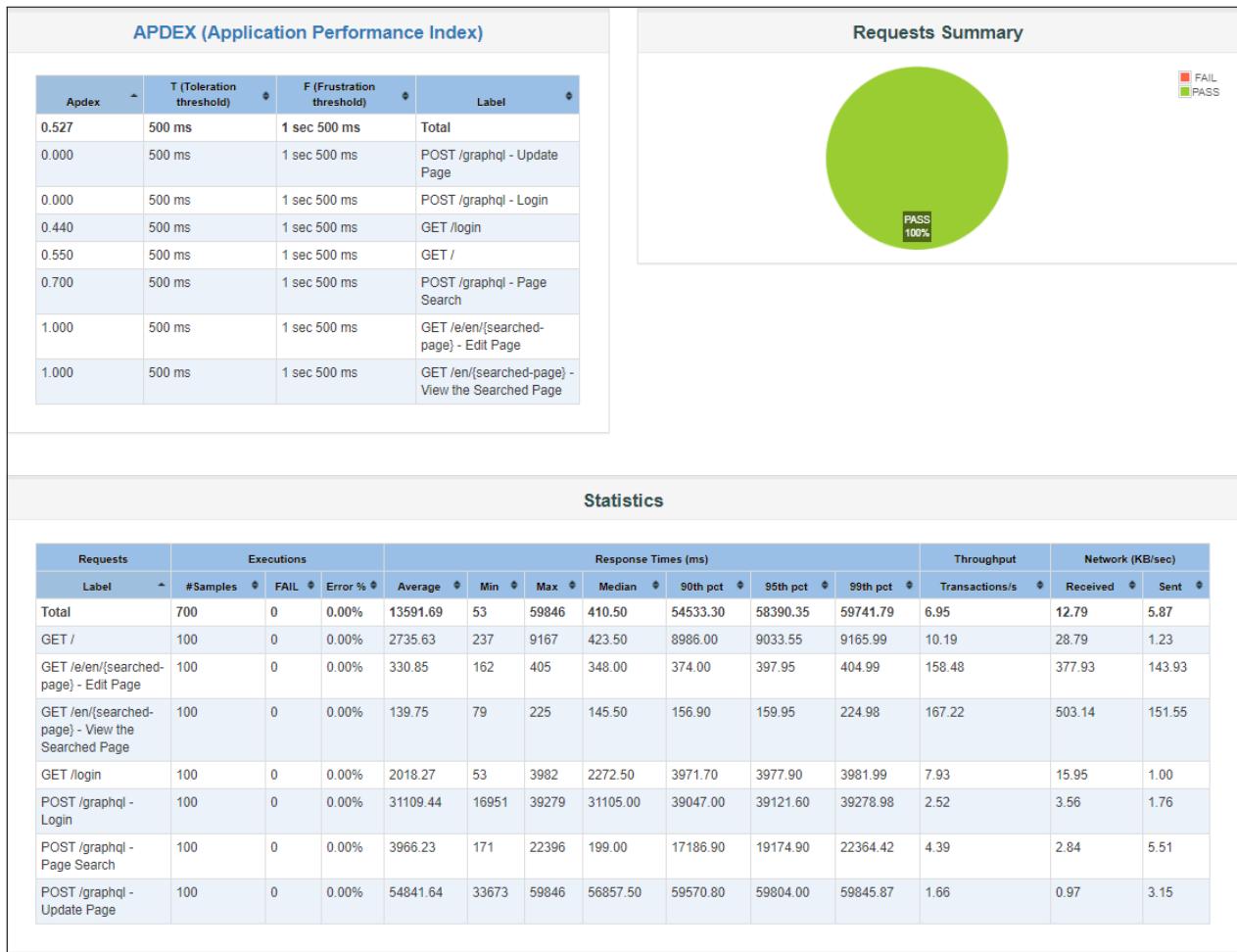


Figura 44: Estatísticas do Teste 6 para 100 threads

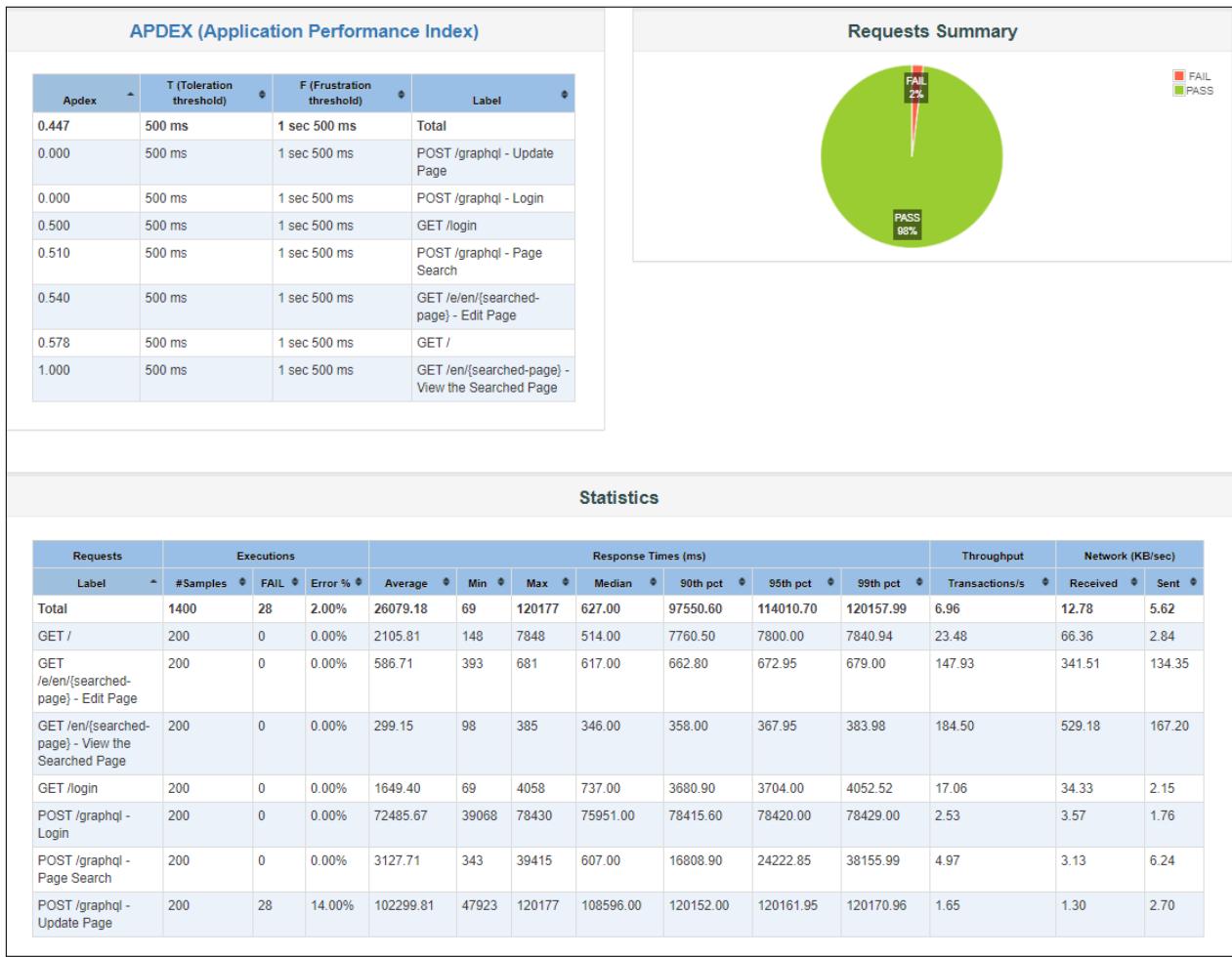


Figura 45: Estatísticas do Teste 6 para 200 threads

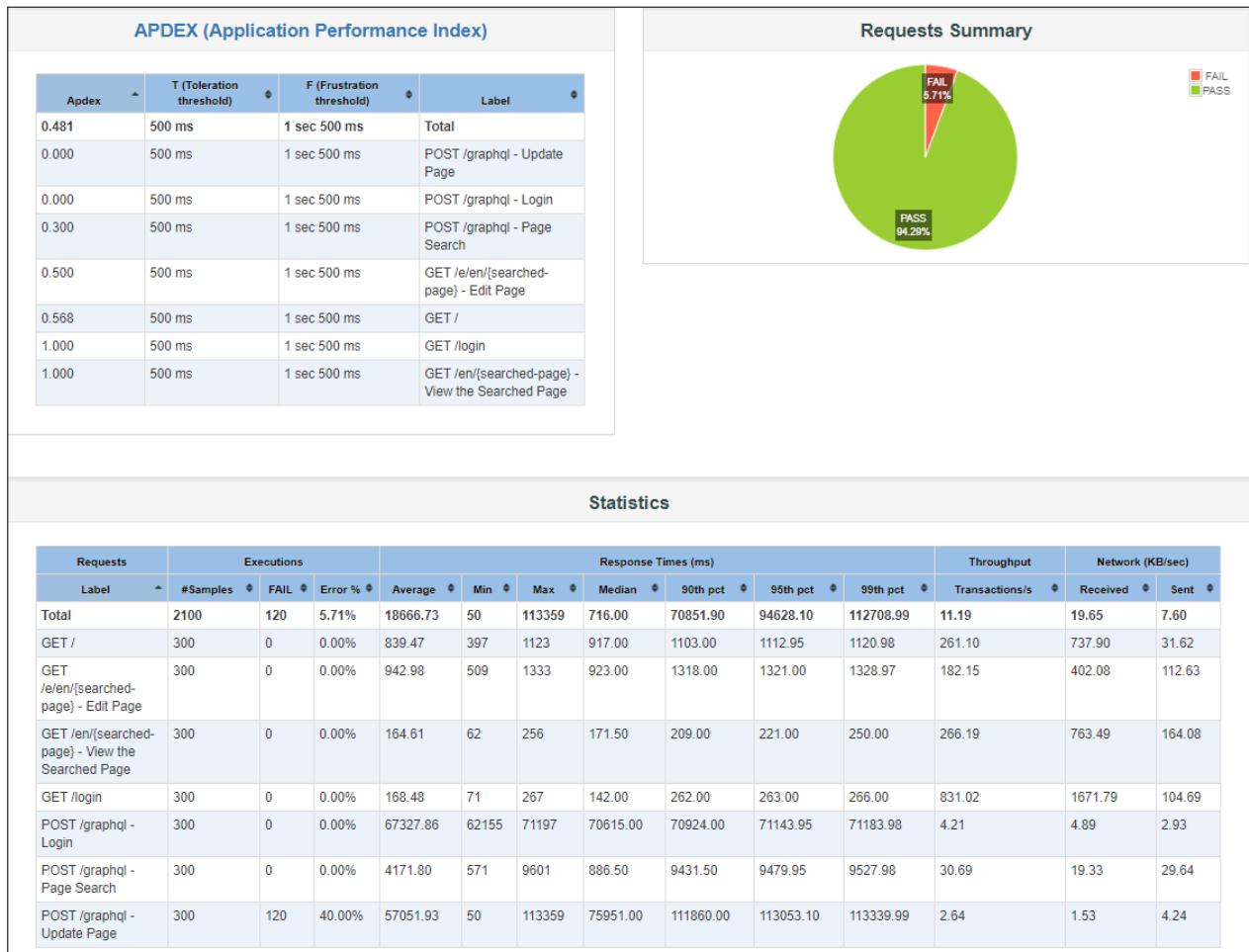


Figura 46: Estatísticas do Teste 6 para 300 threads

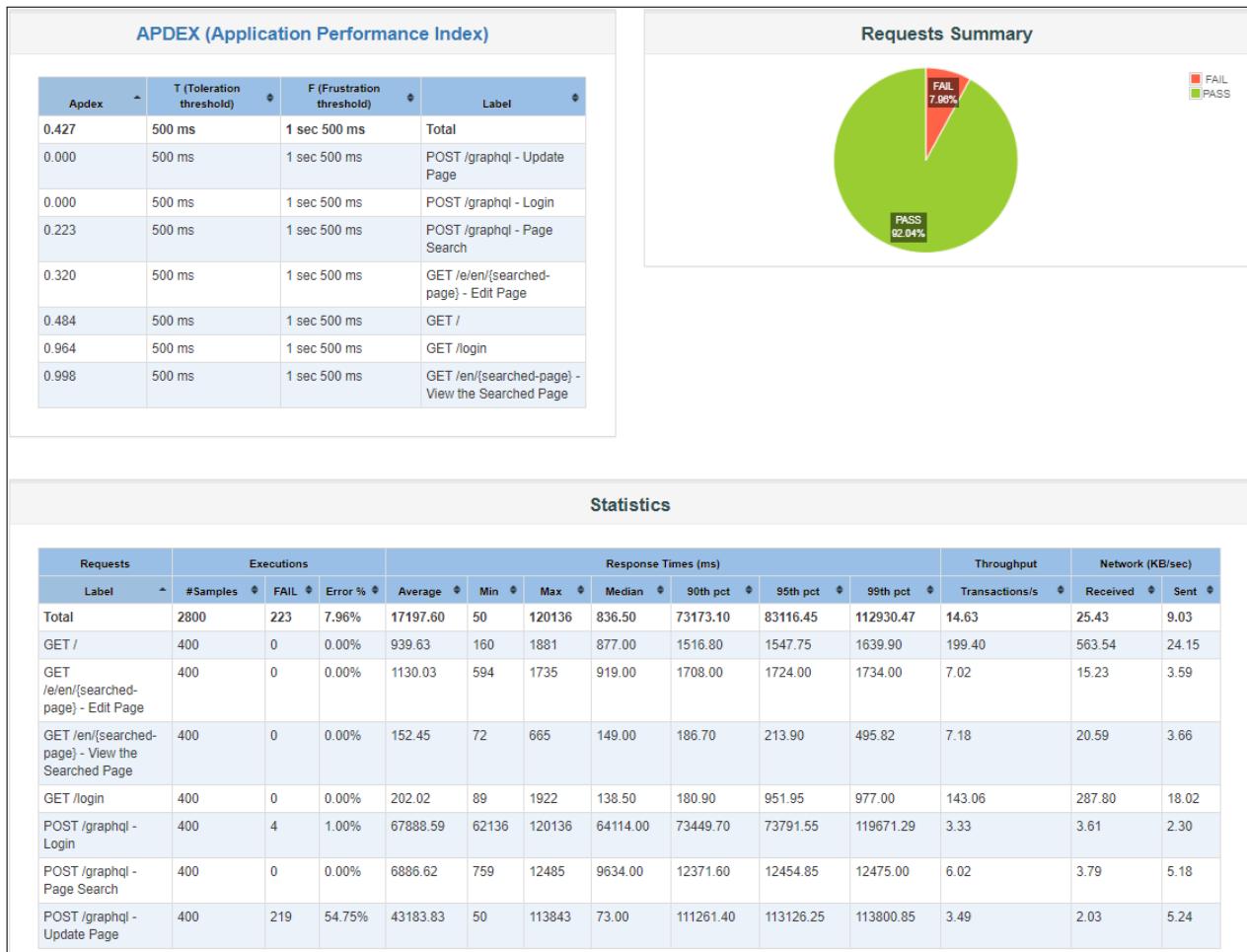


Figura 47: Estatísticas do Teste 6 para 400 threads

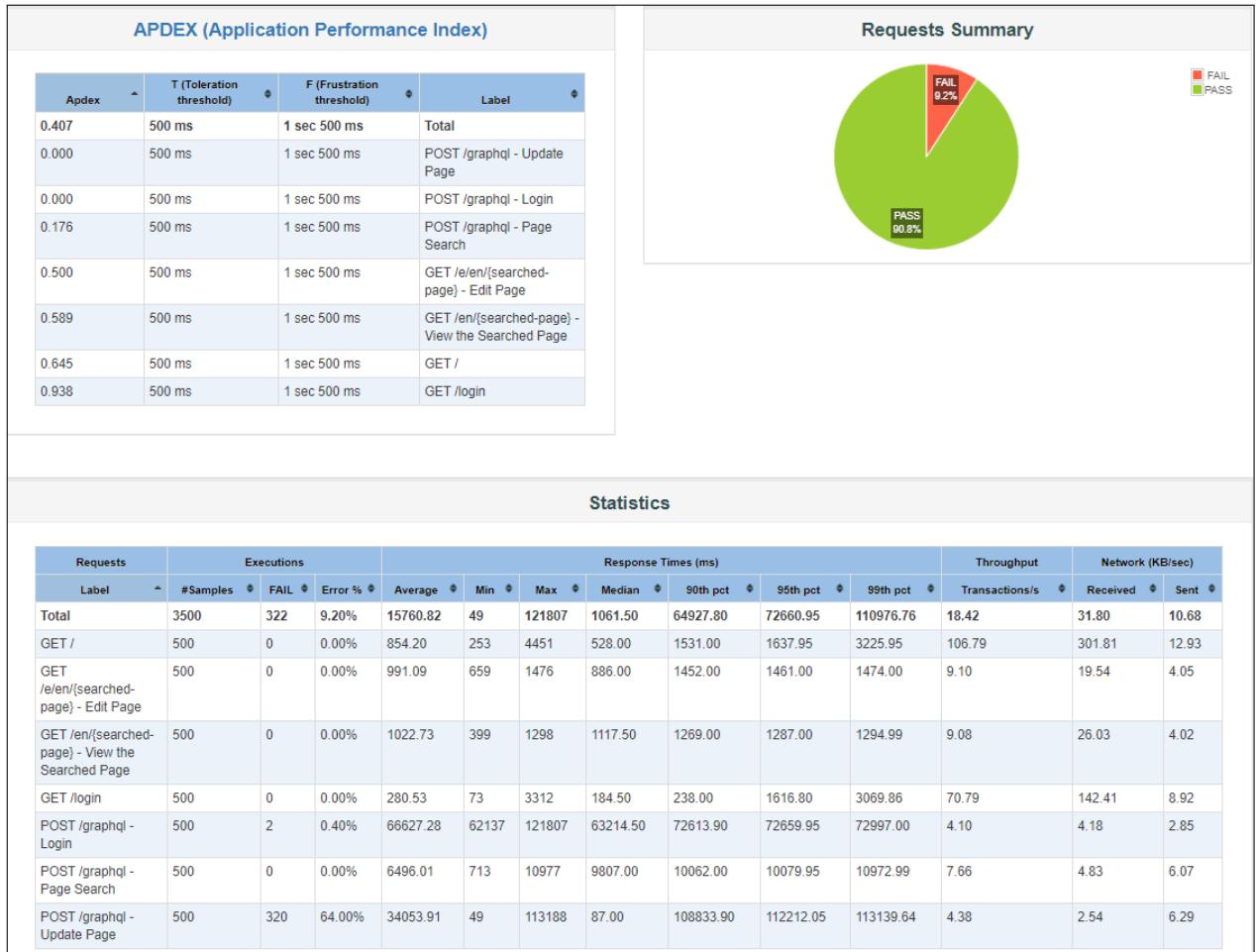


Figura 48: Estatísticas do Teste 6 para 500 threads

## 4.4 Análise dos resultados

Uma vez feitos os testes pode-se concluir que a plataforma, com esta configuração inicial, não está a suportar um número desejado de débito e, para além disso, os tempos de resposta médios acabam por ser bastante altos. É de notar que mesmo para um pedido com baixa carga computacional, como é o exemplo do *Teste 1 (Login)*, o tempo de espera médio para 100 clientes é de 11s, algo inaceitável numa infraestrutura. O teste que apresenta maior tempo de resposta é o *Teste 2 (Login + Listagem + Criação de Página)*, cujo valor médio é de cerca de 40s. Apesar de ser um dos testes com maior carga computacional, 40s de espera torna-se não viável para qualquer infraestrutura. Em relação aos erros obtidas com os testes, o teste que apresenta mais percentagem de erros é o *Teste 2 (Login + Listagem + Criação de Página)* que, como já foi referido anteriormente, é um dos testes que mais sobrecarrega a infraestrutura,

segundo o *Teste 3 (Login + Criação de Users)*, pelos mesmos motivos.

Nos próximos capítulos, irá-se apresentar uma de arquitectura que apresenta alta disponibilidade e tolerância a faltas que visa colmatar os aspectos negativos da configuração inicial. Os testes realizados anteriormente serão executados novamente sobre a nova arquitectura com a expectativa de obter menores valores em relação aos tempos de resposta e percentagens de erros, e maiores valores em relação ao débito.

## 5 Arquitetura Implementada

A escolha da arquitetura de alta disponibilidade e desempenho a implementar baseou-se nos pontos únicos de falha identificados inicialmente. Como tal decidimos ter redundância ao nível do armazenamento através de *armazenamento partilhado* e com replicação tirando partido do DRBD, garantindo que em caso de falha de um dos discos ou máquinas, poderemos continuar a escrever num outro nodo. Além disso, uma vez que existe replicação, podemos aceder a qualquer uma das máquinas para ter acesso aos dados guardados em disco.

Depois na camada onde temos os serviços de base de dados e o sistema de ficheiros, como queríamos garantir a tolerância a faltas nesta camada, optou-se por utilizar um *High Availability Cluster* através do *pacemaker* e *corosync*, garantindo que em caso de falha num dos nodos todos os serviços são migrados para o nodo de *backup*, garantindo assim alta disponibilidade da nossa aplicação.

Para a os *web servers* optou-se por utilizar replicação dos mesmos, sendo que foi introduzido um平衡ador de carga para balancear os pedidos entre as diferentes réplicas, garantindo assim alto desempenho.

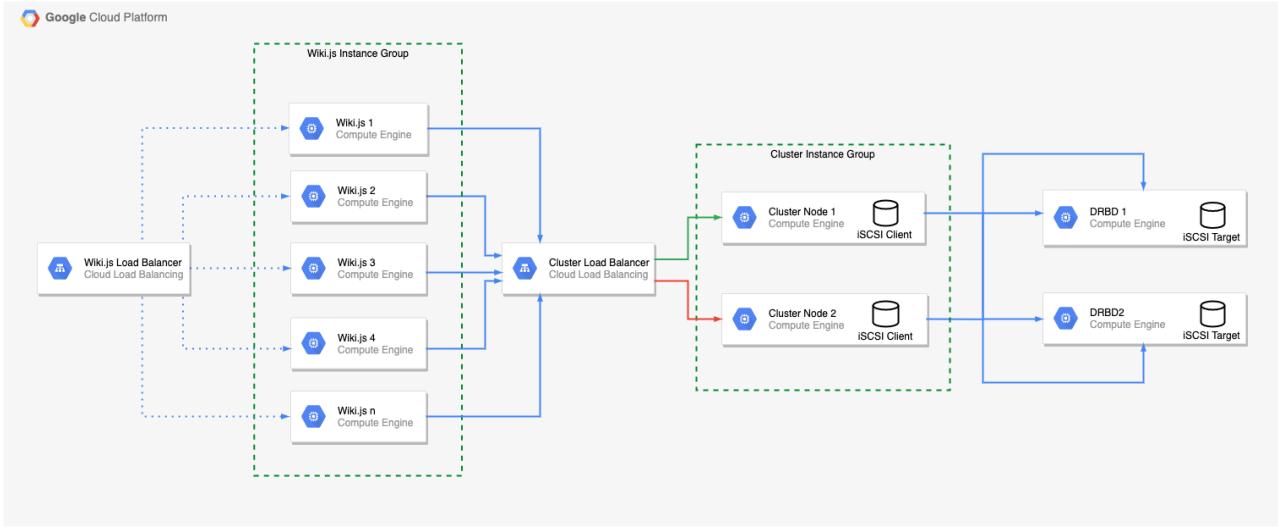


Figura 49: Arquitetura implementada na Google Cloud Platform.

## 5.1 Problemas Encontrados

Durante a implementação da arquitetura definida anteriormente na Google Cloud Platform deparamo-nos com alguns problemas. O primeiro prende-se com o facto de a Google Cloud ter limitações face à utilização de *IP's* flutuantes (virtuais), sendo este um grande entrave para a implementação de algumas soluções abordadas nas aulas práticas desta unidade curricular.

Após bastante pesquisa percebemos que podíamos solucionar este problema utilizando uma camada extra com um平衡ador de carga da Google, garantindo que teríamos então apenas um único endereço IP que estava conectado às nossas máquinas virtuais.

Para o caso do nosso *High Availability Cluster* introduziu-se então essa camada extra com o balanceador de carga e foram criados *health checks* na porta 5432 (porta do PostgreSQL) para garantir que o nosso balanceador iria detetar um nodo sempre com falha e o outro a funcionar, uma vez que com a utilização do cluster os serviços apenas correm num único nodo.

O segundo problema surgiu depois da arquitetura implementada, onde reparámos que ao aceder à Wiki.js através do endereço IP do balanceador externo e efetuar a configuração inicial da Wiki (definir email, password, etc) se verificava que se efetuássemos *refreshes* à página em algumas vezes aparecia novamente a página para proceder à configuração inicial. Após investigarmos um pouco sobre o assunto, reparámos que na documentação do Wiki.js é referido que para se utilizar o Wiki.js em *High Availability* é necessário efetuar a configuração inicial numa máquina e só depois disso criar as réplicas, sendo necessário também alterar um parâmetro no ficheiro de configuração (*ha: true*). Posto isto para solucionar o problema bastou mudar no ficheiro de configuração o parâmetro *ha* para *true* e efetuar a configuração inicial

da Wiki.js uma vez, desligar as máquinas virtuais e voltar a ligar, ficando assim já todas sincronizadas.

## High-Availability



This feature is available from version **2.3 and up**.



PostgreSQL is **required** to enable this option.

**You must deploy a single instance in order to setup the application.** Once setup is completed, you can increase the number of replicas to any amount.

Set to `true` if you have multiple concurrent instances running off the same DB (e.g. Kubernetes pods / load balanced instances). Leave `false` otherwise.

```
1 | ha: true
```

Figura 50: High Availability do Wiki.js.

## 5.2 Storage

O armazenamento de dados é feito em duas máquinas diferentes usando a ferramenta *DRBD* (*Distributed Replication Block Device*). Ambas as máquinas estão definidas como primárias significando que os recursos criados estão disponíveis, simultaneamente, nas duas instâncias, ao contrário do que aconteceria se se estivesse a usar um modo Primário-Secundário. O modo de replicação de dados usa o Protocolo C que garante que uma escrita só é efectivamente concluída quando é feita no disco local e no disco remoto (escrita síncrona). Como resultado, este protocolo garante que, na falha de uma das máquinas, os dados não são perdidos. Cada uma das máquinas nesta camada implementam um *ISCSI Target* que é usado pelo *ISCSI Client* definido nos nós do *cluster*, na camada superior.

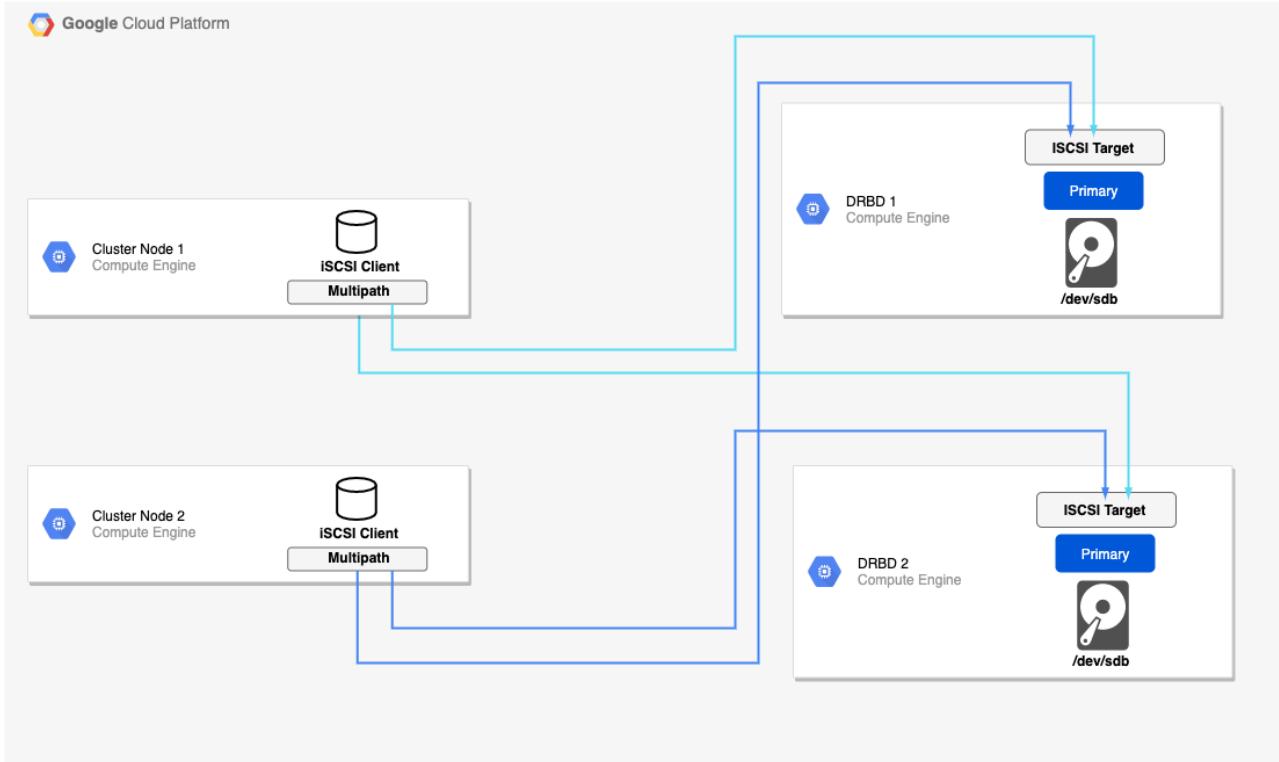


Figura 51: Arquitetura da Storage Partilhada.

### 5.3 High Availability Cluster

Um dos pontos que achamos necessários resolver foi a falha na camada de base de dados, uma vez que toda a nossa aplicação depende da mesma. Para garantir isto decidimos utilizar um *high availability cluster* nessa camada, garantindo que em caso de falha todos os serviços são migrados de um nodo para outro.

#### 5.3.1 Arquitetura do High Availability Cluster

A arquitetura escolhida para o cluster teve de ser adaptada devido ao problema dos IP's virtuais, posto isto introduziu-se uma nova camada que é o balanceador interno. Como se pode observar pela figura abaixo, através deste balanceador a cada instante o tráfego é redirecionado para a instância que tem os serviços a correr, em caso de falha nesta instância será então utilizada a outra rota, garantindo assim que a instância acedida é de facto a que tem os serviços a correr.

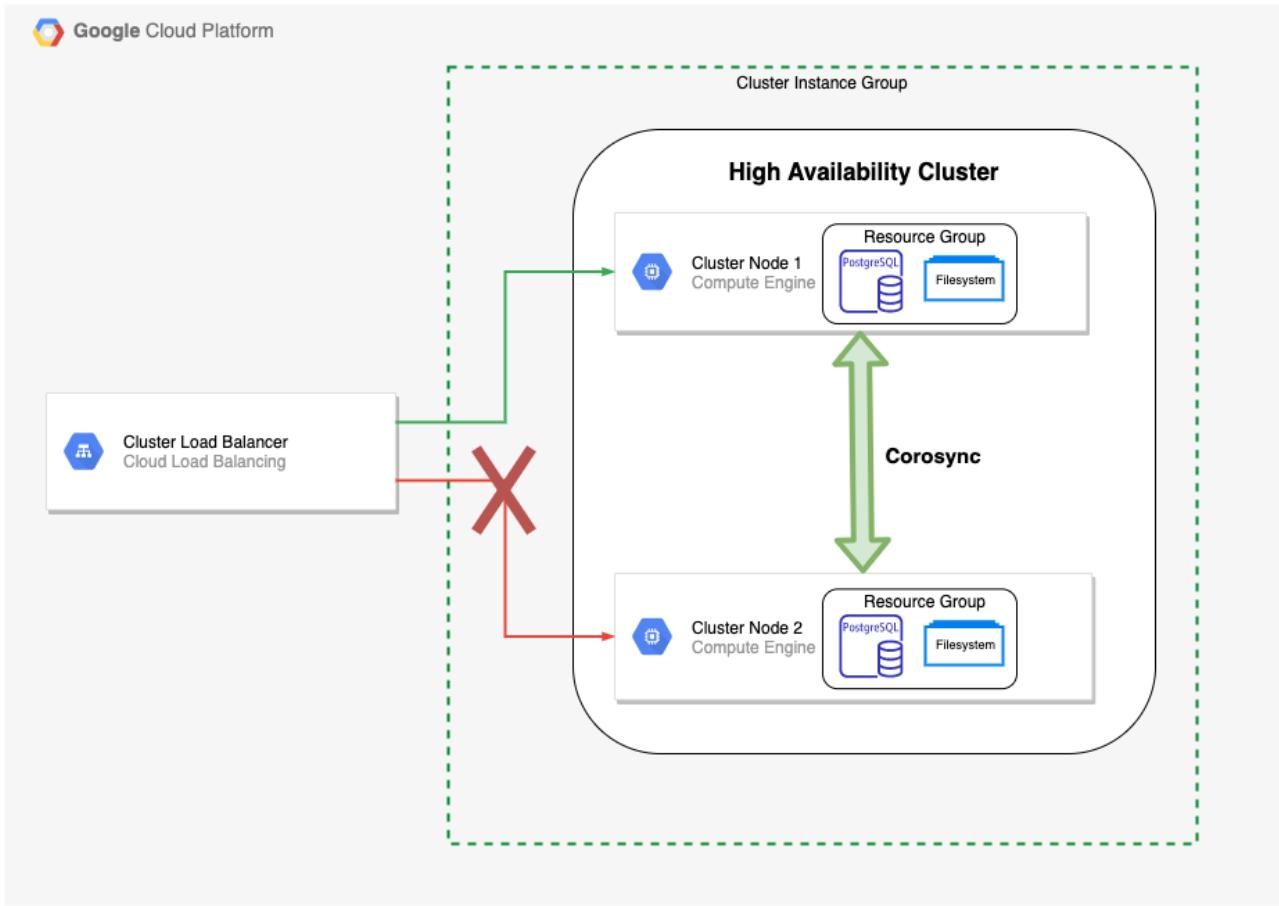


Figura 52: Arquitetura do High Availability Cluster.

### 5.3.2 Balanceador de Carga

Tal como referido na secção dos Problemas Encontrados, existiu um grande problema com a utilização de endereços IP's virtuais para acesso ao nosso cluster. Para combater este problema decidimos utilizar um balanceador de carga TCP interno para que assim conseguissemos ter um único endereço IP entre os dois nodos do nosso cluster.

Para se conseguir obter isto, foi criado um grupo de instâncias na Google Cloud composto pelos dois nodos do cluster e uma vez que o serviço crucial nesses nodos para o nosso frontend era a base de dados em PostgreSQL decidimos criar uma verificação de integridade no balanceador sobre a porta 5432. Com esta verificação de integridade vai ser possível que o nosso balanceador redirecione todo o tráfego relativo à base de dados para o nodo que efetivamente está a correr esse serviço. Em caso de falha dos serviços de base de dados nesse nodo, estes irão ser migrados para o nodo de backup e através da verificação de integridade irá verificar-se que o nodo anterior deixou de estar disponível e portanto todo o tráfego irá ser redirecionado

para o nodo de backup.

Com esta alternativa conseguimos camuflar o problema dos endereços IP's virtuais e obter tolerância a faltas na camada de base de dados.

The screenshot shows the configuration details for a load balancer named 'wikis-internal-balancer-db'. It includes sections for Front-end and Back-end configurations, along with a summary of endpoint settings and advanced configuration options.

**Front-end**

Protocolo	Escopo	Sub-rede	IP:Portas	Nome do DNS
TCP	Regional (europe-west1)	default (10.132.0.0/20)	10.132.0.50:5432	

**Back-end**

Região: europe-west1 | Rede: default | Protocolo do endpoint: TCP | Afinidade da sessão: Nenhuma | Verificação de integridade: wikis-db-health-check

Configurações avançadas

Grupo de instâncias	Zona	Integra	Escalonamento automático	Usar como grupo de failover
wikis-db-group	europe-west1-b	1 / 2	Nenhuma configuração	Não

Figura 53: Balanceador Interno sobre o Cluster.

### 5.3.3 Cluster

Para criação do cluster recorremos ao cluster utilizado nas aulas práticas desta unidade curricular, tirando partido do *pacemaker* e do *corosync*. No cluster existem então dois nodos, o nodo 1 que terá maior prioridade para correr os serviços e o nodo 2 que servirá como backup em caso de falha do nodo 1.

Como recursos decidimos ter um recurso para o filesystem, que irá tirar partido do ISCSI sobre multipath. Este recurso irá montar os ficheiros da nossa base de dados na diretoria /mnt/db/postgresql, sendo possível que em caso de falha esta diretoria seja montada automaticamente no outro nodo. Temos também um outro recurso que é o PostgreSQL que irá correr a nossa base de dados, sendo possível por isso conectarmo-nos remotamente à base de dados através do IP interno fornecido pelo balanceador de carga. Estes recursos estão agrupados num grupo com o nome: db. Nas figuras abaixo consegue verificar-se a configuração obtida.

**Cluster: wikijs\_cluster**

**NODES**

- Remove + Add
- wikijs\_1
- wikijs\_2

Edit Node **wikijs\_1**

Node ID: 1 Uptime: 0 days, 00:17:38

Cluster Daemons

NAME	STATUS
pacemaker	✓ Running (Enabled)
corosync	✓ Running (Enabled)
pcsd	✓ Running (Enabled)

Running Resources

NAME
db_pgsql (ocf:heartbeat:pgsql)
db_fs (ocf:heartbeat:Filesystem)

Resource Location Preferences

NAME	Score
wikis_db	80

Figura 54: Página do Cluster com os nodos existentes.

**Cluster: wikijs\_cluster**

**NODES**

- Remove + Add
- wikijs\_1
- wikijs\_2

Edit Node **wikijs\_2**

Node ID: 2 Uptime: 0 days, 00:17:18

Cluster Daemons

NAME	STATUS
pacemaker	✓ Running (Enabled)
corosync	✓ Running (Enabled)
pcsd	✓ Running (Enabled)

Running Resources

NAME
NONE

Resource Location Preferences

NAME	Score
wikis_db	20

Figura 55: Nodo Wikijs 2.

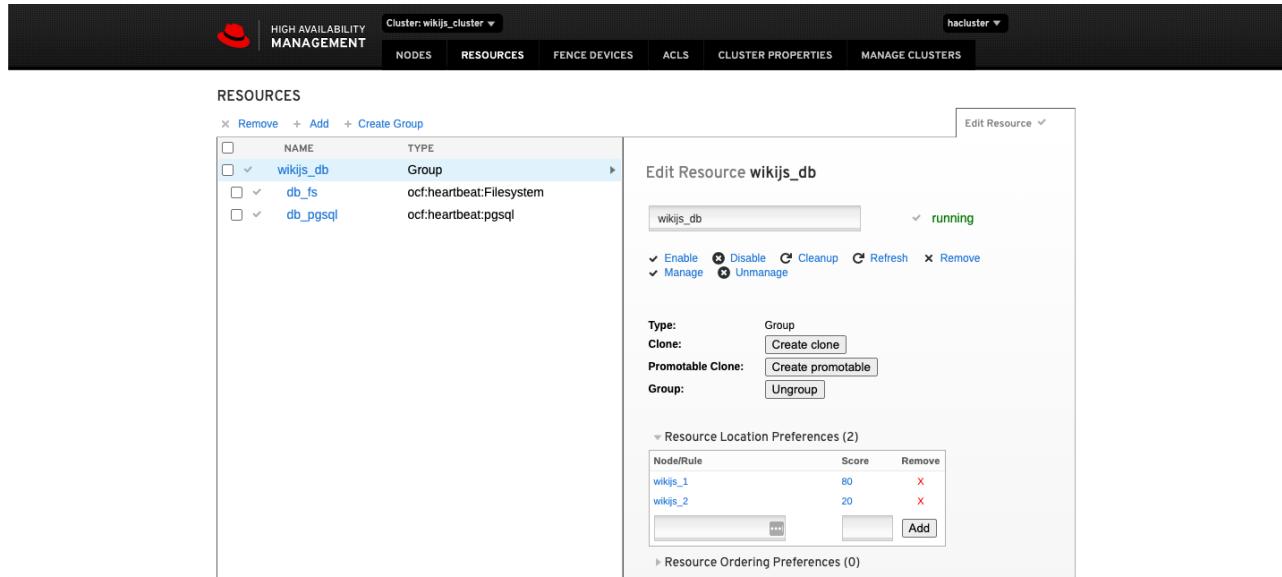


Figura 56: Grupo de Recursos no Cluster.

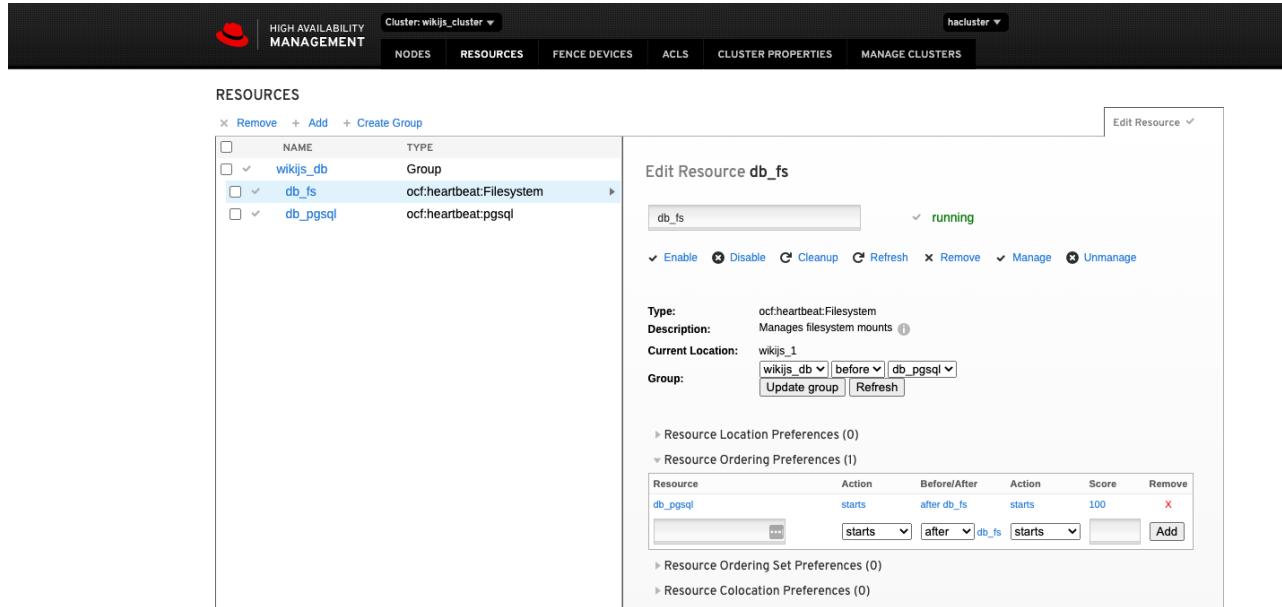


Figura 57: Recurso do Filesystem.

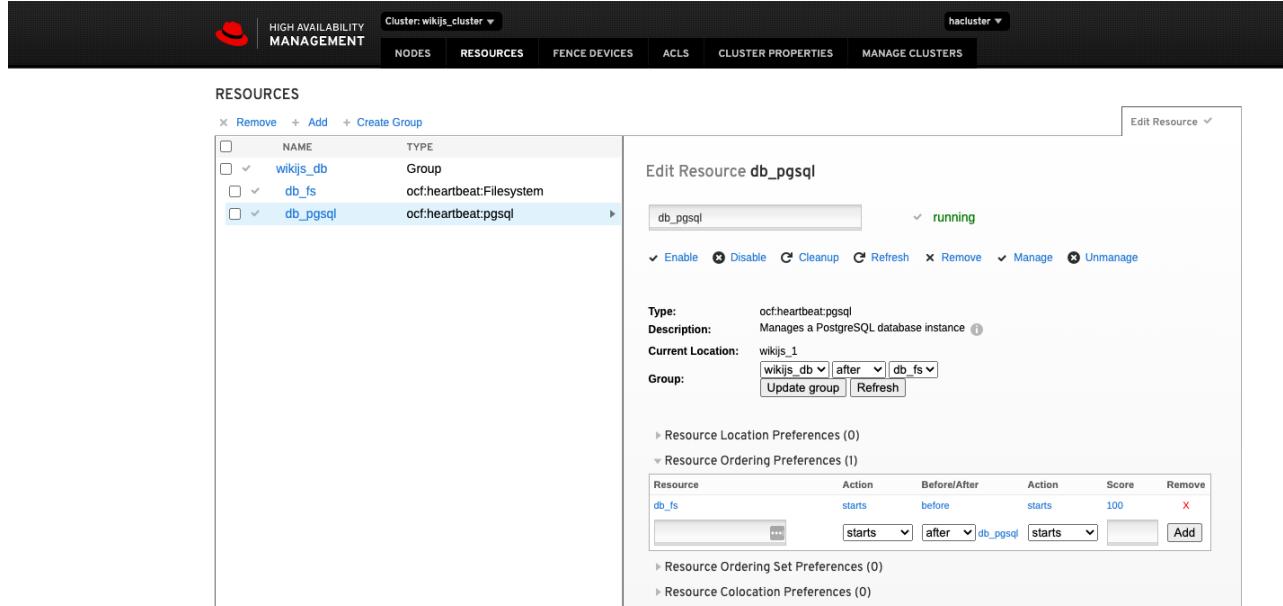


Figura 58: Recurso do PostgreSQL.

## 5.4 Web Servers e Política de Balanceamento

Após analisar formas de balancear a carga através das ferramentas aprendidas nas aulas práticas como o *Linux Virtual Server (LVS)* com base no serviço *KeepAlive* para a monitorização dos *Web Servers*, como referido anteriormente, o grupo chegou à conclusão que não iria ser possível fazer o balanceamento desta forma na *Google Cloud*.

### 5.4.1 Balanceador de Carga

Depois de várias pesquisas na *internet* sobre como cumprir este objectivo na *GCP*, foi descoberto que a própria *Google Cloud* fornece serviços de balanceamento *HTTP*. Este balanceador não representa um ponto de falha devido ao seu *design* porque este é um serviço totalmente distribuído, definido por *software* e gerenciado para todo o tráfego. Não é uma solução baseada em instância ou dispositivo, então não irá ficar preso à infraestrutura de balanceamento de carga física.

O *Cloud Load Balancing* distribui a carga sobre um grupo de instâncias que estão sobre um serviço de *Back-end* ao qual podemos definir o protocolo de balanceamento como tempos de *timeout*, verificações de integridade e outras configurações mais avançadas às quais não foram analisadas com profundidade.

The screenshot shows the Google Cloud Balancer interface for the service 'wikijs-balancer'. The 'Front-end' section displays a single endpoint: Protocolo: HTTP, IP:Porta: 35.190.33.180:80, and Nível da rede: Premium. The 'Regras de host e caminho' section shows a default rule for 'Todos sem correspondência (padrão)'. The 'Back-end' section lists one service: '1. wikijs-backendservice'. This service has a protocol of HTTP, a named port of 'http', a timeout of 650 seconds, and is configured with Cloud CDN desativada, Política de tráfego desativada, and Verificação de integridade set to 'wikijs-healthcheck'. Below this, there's a 'Configurações avançadas' section and a table for the 'wikijs-group' back-end instance. The table columns are Nome, Tipo, Zona, Íntegro, Escalonamento automático, Modo de balanceamento, Capacidade, and Portas selecionadas. The row shows 'wikijs-group' as a Grupo de instâncias in the europe-west1 zone, with 4/4 healthy instances, no automatic scaling, round-robin balancing, 100% capacity, and port 10000 selected.

Figura 59: *Google Cloud Balancer* sobre 4 instâncias da aplicação *Wikijs*

#### 5.4.2 Web Servers

O grupo de instâncias representa um número dinâmico ou estático de máquinas virtuais com as mesmas configurações. As *vms* são criadas automaticamente através de um modelo que contém as mesmas configurações de *cpu* e memória que as referidas no início da secção 4 e um *script* que faz a instalação da aplicação.

Devido à estrutura da aplicação *Wikijs*, não foi separada em *front-end* e em *back-end* ao qual foi decidido fazer a instalação da aplicação completa em cada instância do grupo que por sua vez estão conectadas ao *cluster* para comunicar com a base de dados. Optamos por lançar 4 instâncias destas para o grupo devido ao limite de número de instâncias que podemos correr por zona.

Esta instâncias estão hospedadas na zona oeste da Europa e para obter mais disponibilidade na nossa infraestrutura, foi configurado para distribuir as várias *vms* pelas sub-zonas desta para caso uma delas falhe ainda existirem máquinas virtuais em funcionamento. Para além disso, foi utilizada outra ferramenta do *Google Cloud* que é a *Health Check* que nos permite verificar a integridade de cada instância do grupo e que em caso de falha, permite-nos tomar medidas de maneira a manter a aplicação no ar e da melhor forma possível no momento.

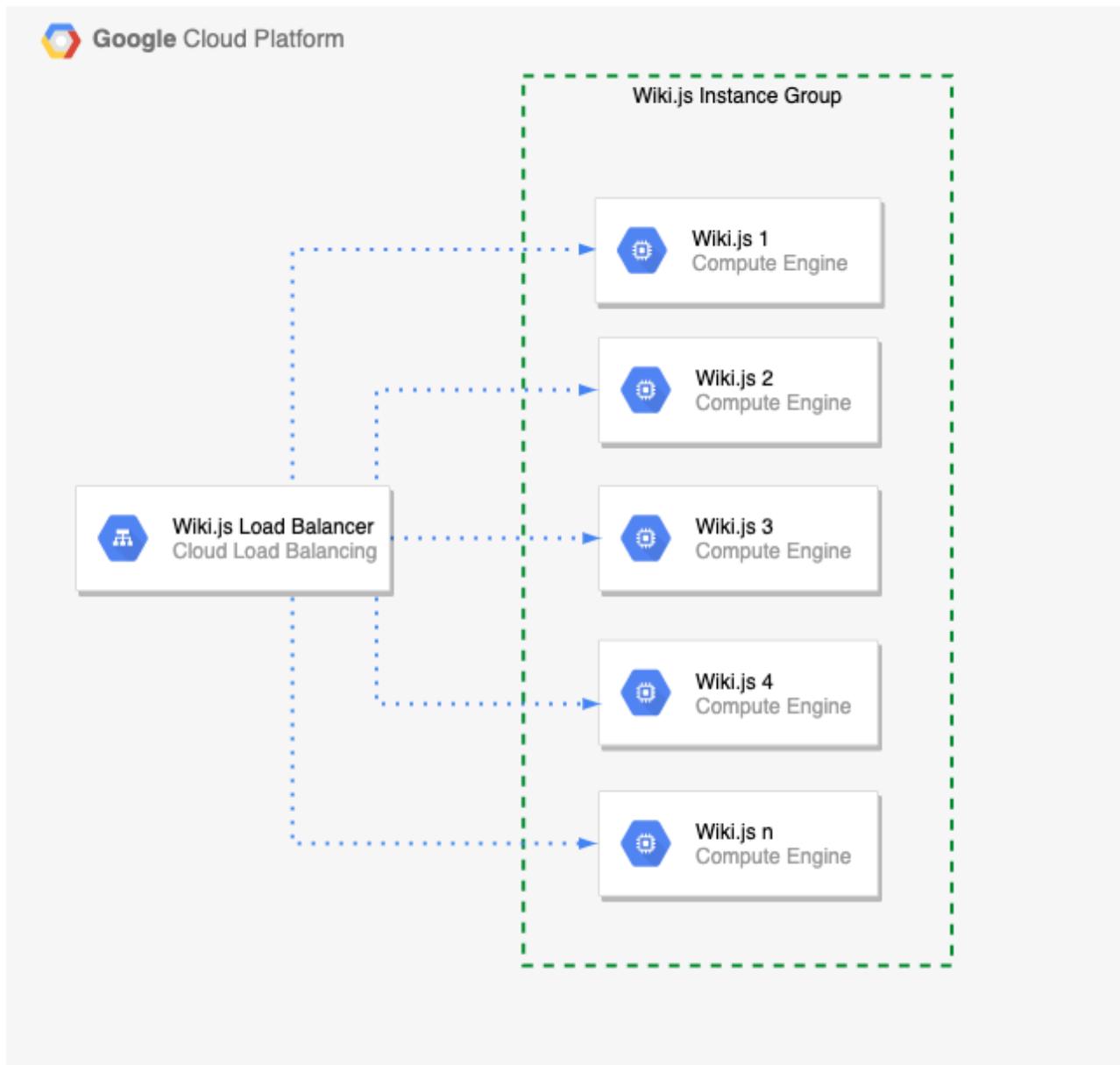


Figura 60: Arquitetura do *front-end*

## 5.5 Testes de Carga

Os testes apresentados nas seguintes seções foram realizados sobre a nova arquitetura e são exatamente os mesmos realizados anteriormente para que haja coerência na comparação dos resultados obtidos.

### 5.5.1 Teste 1 : Login

Comparando os resultados obtidos com os anteriores, tanto os valores de tempo de resposta e débito melhoraram bastante, sendo os do tempo de resposta os que sofreram maior impacto. O valor mais alto registado nos testes anteriores correspondia ao teste com 200 *threads* e tinha o valor de 25s. Dada esta arquitetura, pode-se observar que esse valor baixou para 6s, o que é uma melhoria bastante notável. A nível de erros, houve um aumento ligeiro neste valor, mas pode-se considerar insignificativo.

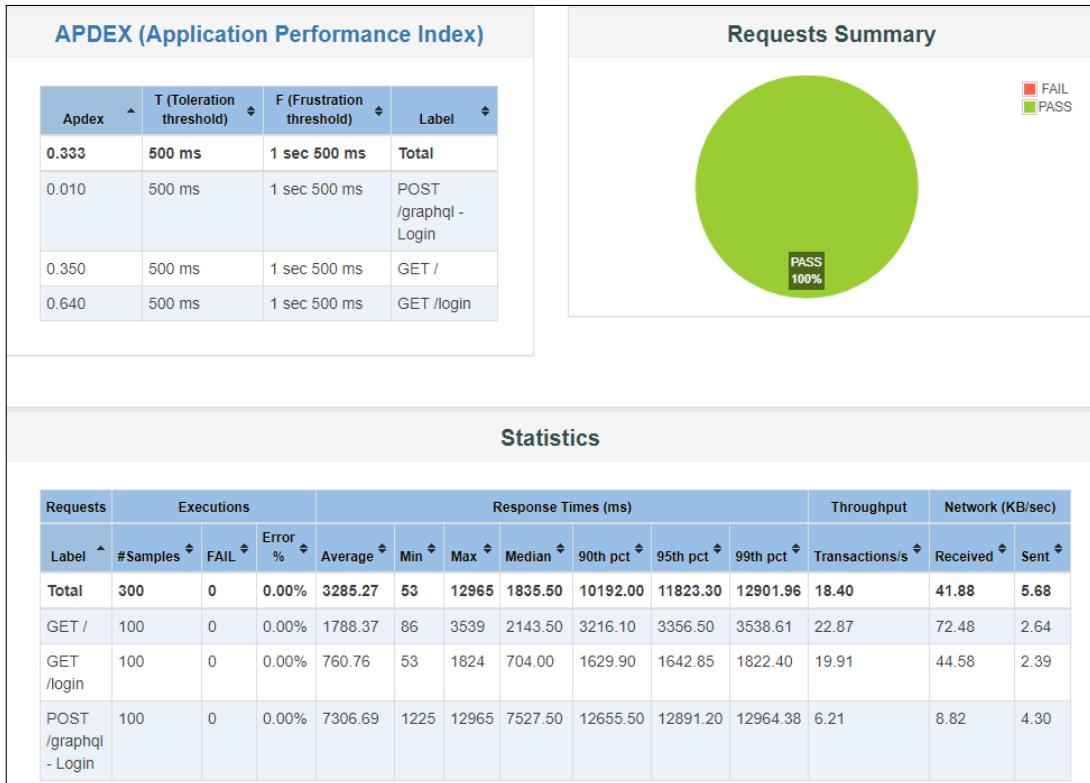


Figura 61: Estatísticas da arquitectura implementada com o Teste 1 para 100 *threads*

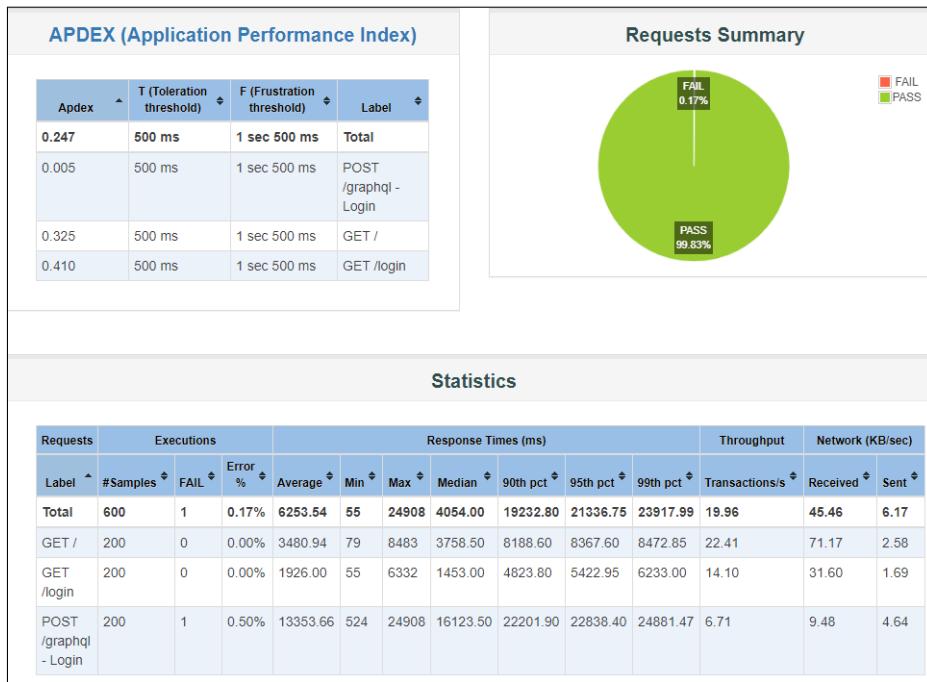


Figura 62: Estatísticas da arquitectura implementada com o Teste 1 para 200 threads

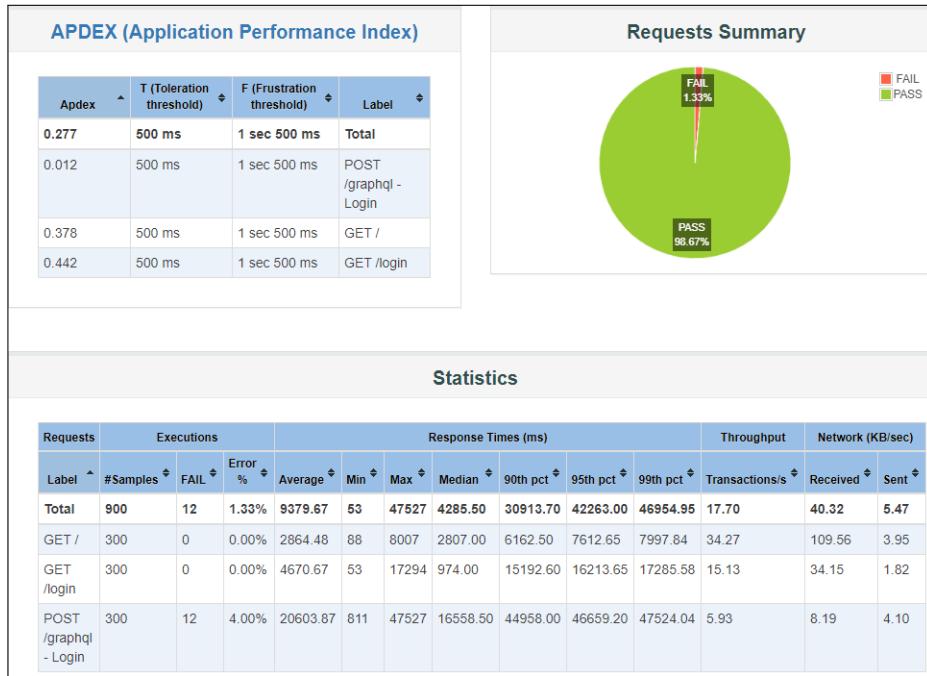


Figura 63: Estatísticas da arquitectura implementada com o Teste 1 para 300 threads

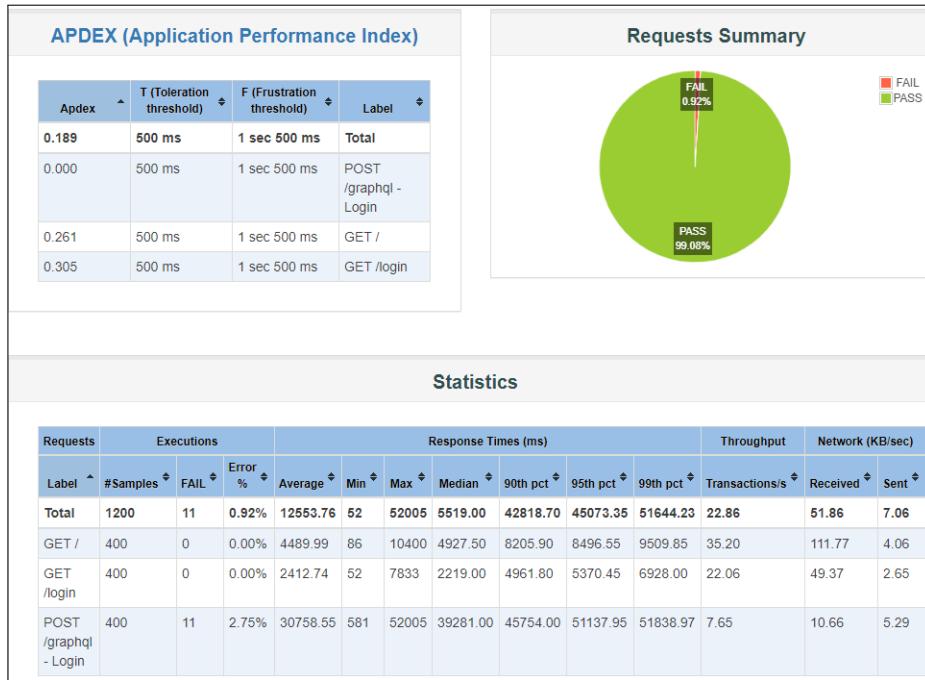


Figura 64: Estatísticas da arquitectura implementada com o Teste 1 para 400 threads

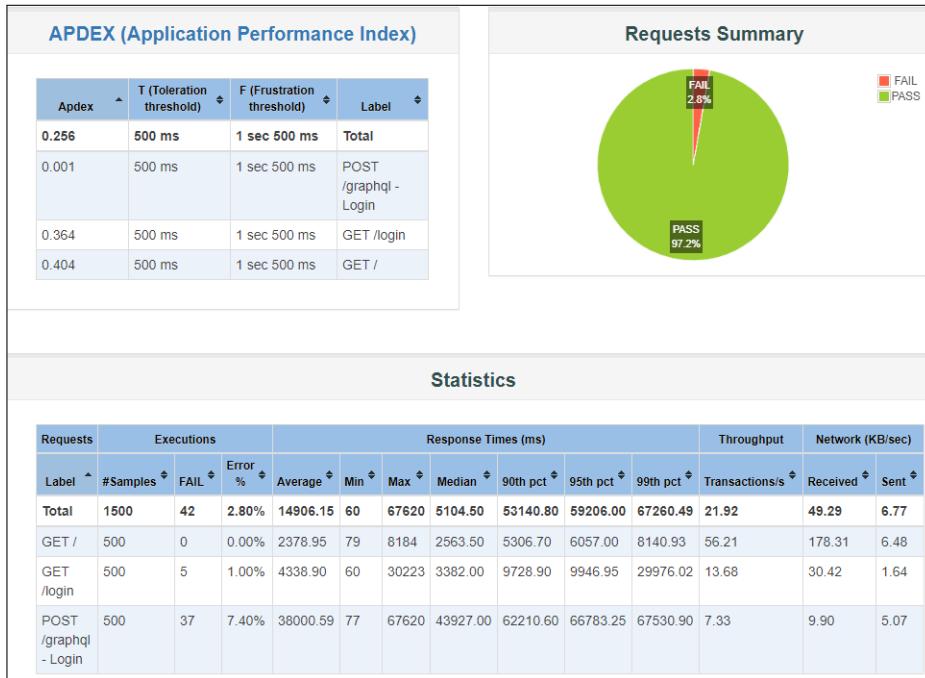


Figura 65: Estatísticas da arquitectura implementada com o Teste 1 para 500 threads

### 5.5.2 Teste 2: Login + Listagem de Páginas + Criação de páginas

Como se pode verificar nos resultados obtidos anteriormente, este teste apresenta uma carga relativamente alta para o servidor, sendo que foi dos testes que apresentou piores resultados na primeira fase de análise. Dada a alta disponibilidade da infraestrutura e como se pode observar nos resultados abaixo apresentados, consegui-se uma melhoria bastante alta. O pior valor obtido anteriormente em relação ao tempo de resposta foi o teste com 200 clientes, assumindo o valor de 38s e, com esta nova configuração, esse valor baixou para 11s. Os tempos de resposta para os número de clientes igual a 400 e 500 manteve-se praticamente igual, havendo apenas melhoria no débito. O número de erros baixou em todos os casos, há exceção do teste com 100 clientes que teve um ligeiro aumento.

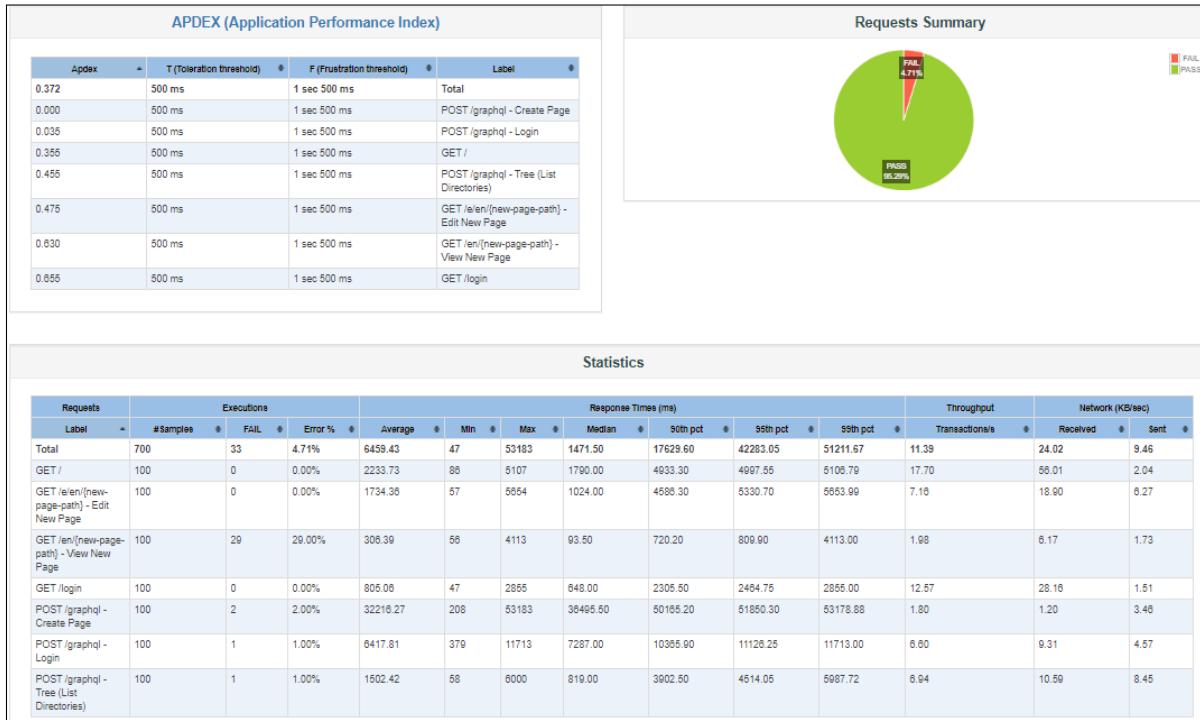


Figura 66: Estatísticas da arquitectura implementada com o Teste 2 para 100 *threads*

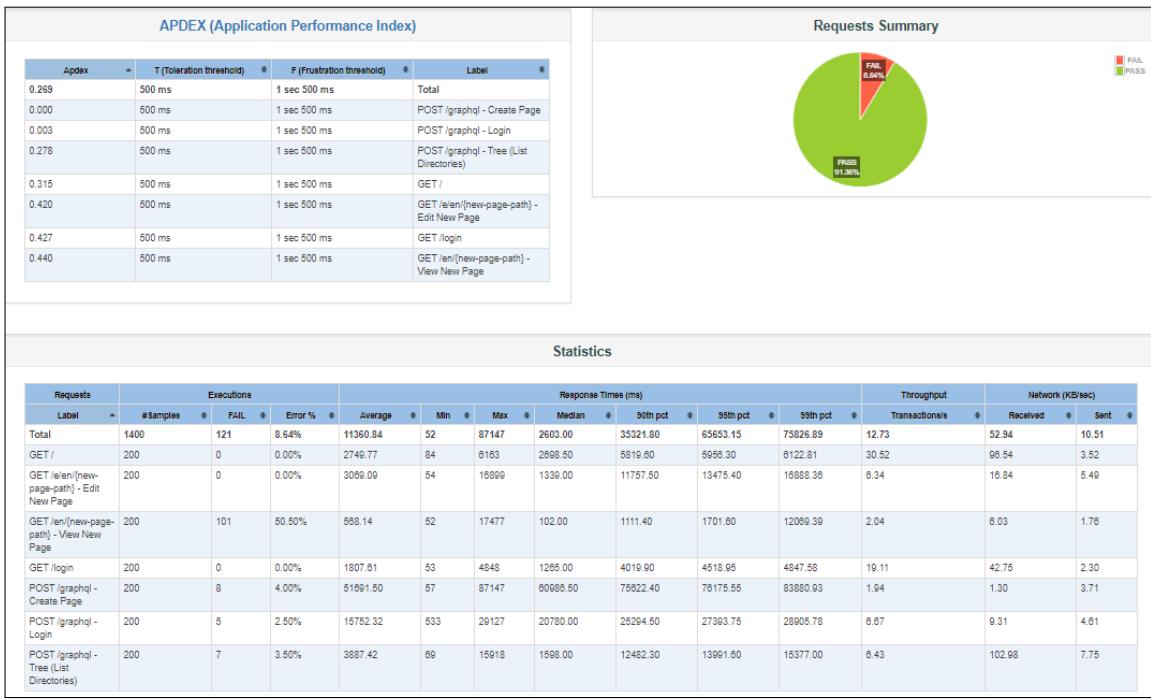


Figura 67: Estatísticas da arquitectura implementada com o Teste 2 para 200 threads

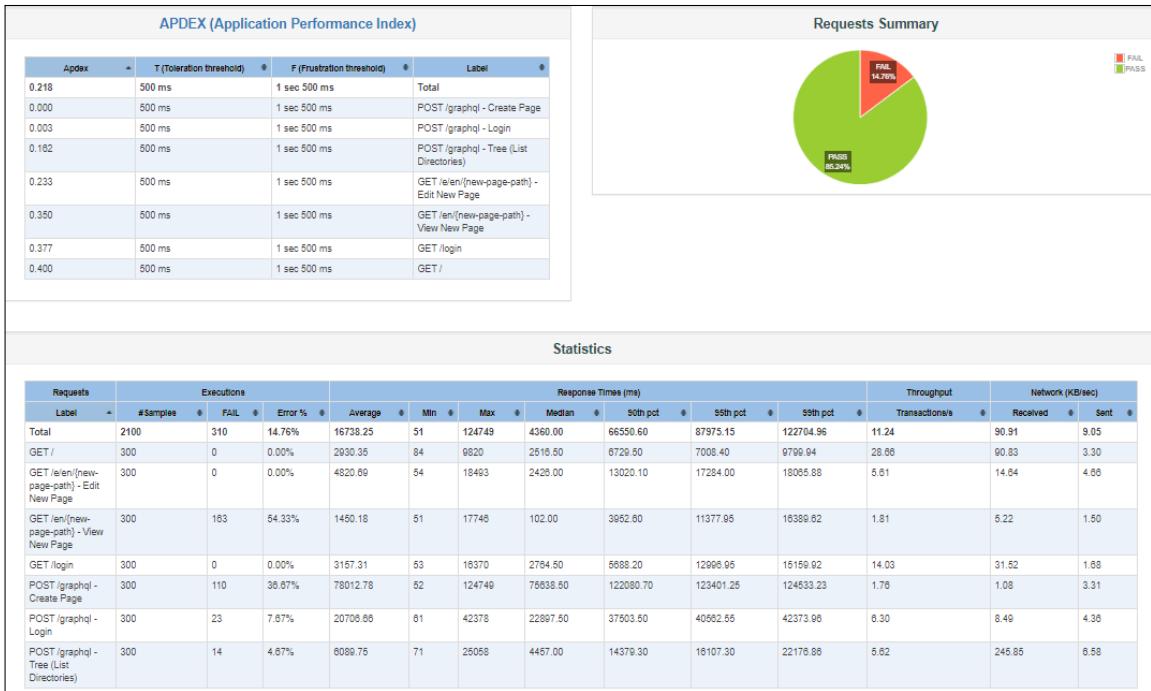


Figura 68: Estatísticas da arquitectura implementada com o Teste 2 para 300 threads

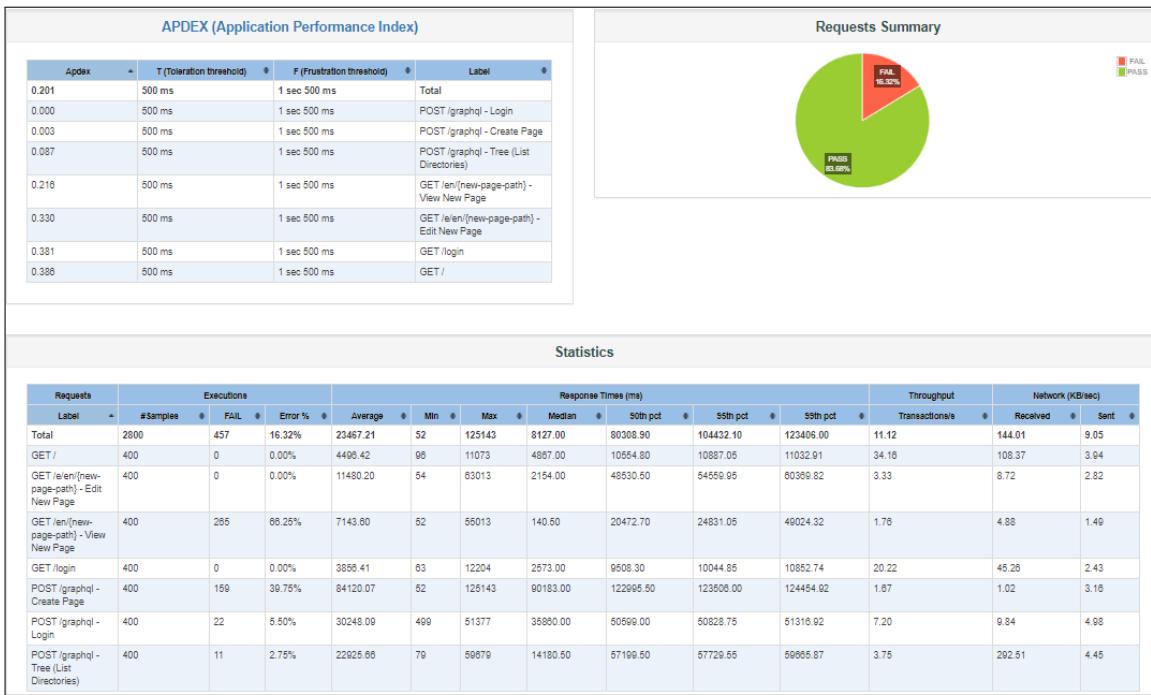


Figura 69: Estatísticas da arquitectura implementada com o Teste 2 para 400 threads

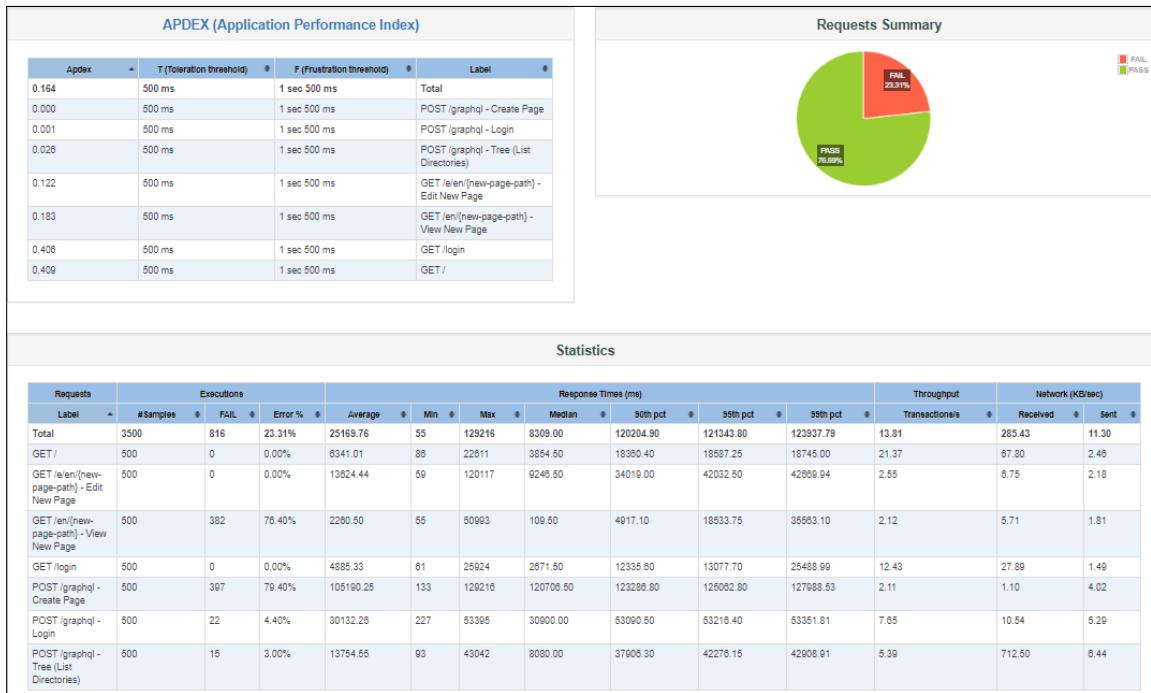


Figura 70: Estatísticas da arquitectura implementada com o Teste 2 para 500 threads

### 5.5.3 Teste 3: Login + Pesquisa + Obtenção da Página Pesquisada

Foi aplicado novamente o teste 3 para 100, 200, 300, 400 e 500 *threads* mas desta vez sobre a nova arquitetura. Os resultados obtidos estão nas figuras 71, 72, 73, 74 e 75 para cada número de *threads* respectivamente.

Comparativamente aos resultados da subseção 4.3.4 apesar de haver uma ligeira subida na percentagem de erros de 100 a 400 *threads* e uma subida mais acentuada com 500 *threads*, o tempo médio de resposta é 2 vezes menor do que a arquitetura inicial, como por exemplo com 400 utilizadores em simultâneo passou de 15 s para apenas 8 s em média. Houve uma diminuição ainda maior principalmente nos pedidos *POST LogIn* em que com o exemplo de 400 *threads* desceu de 70 s para 28 s.

Para além disto, o número de transações por segundo aumentou consideravelmente em que nos novos testes esteve na casa dos 30 tx/s enquanto que nos testes na primeira arquitetura manteve-se entre os 10 e 20 tx/s dependendo do número de *threads*.

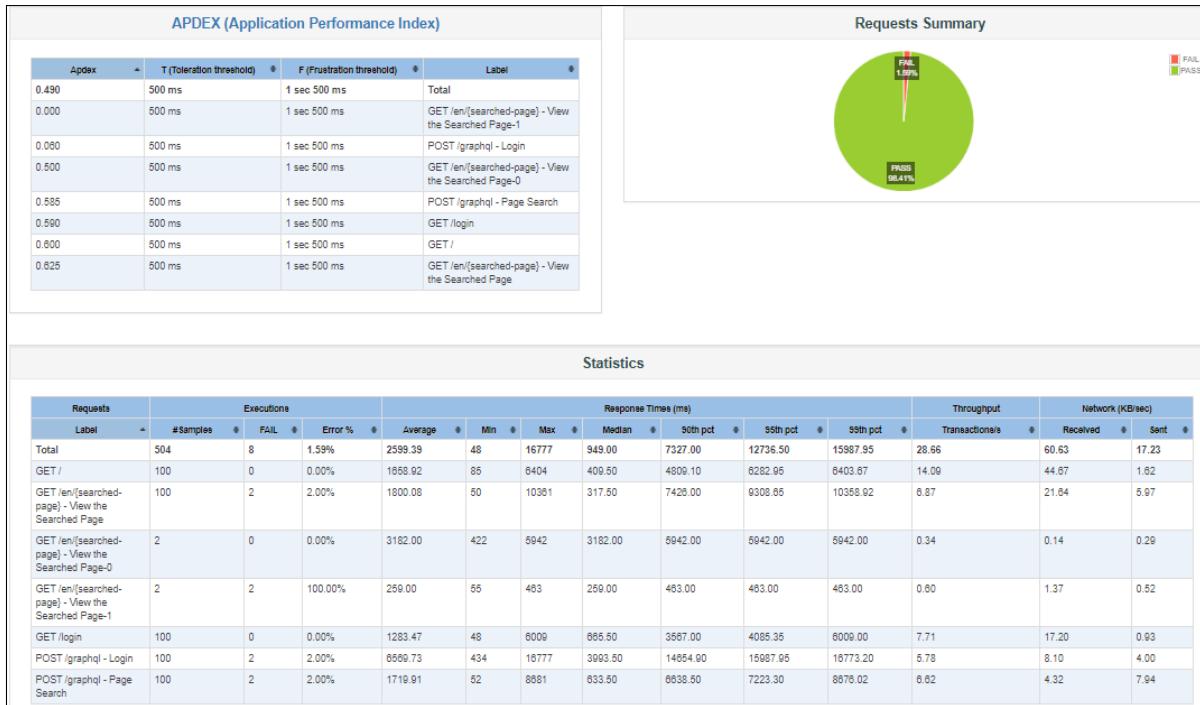


Figura 71: Estatísticas da arquitetura implementada com o Teste 3 para 100 *threads*

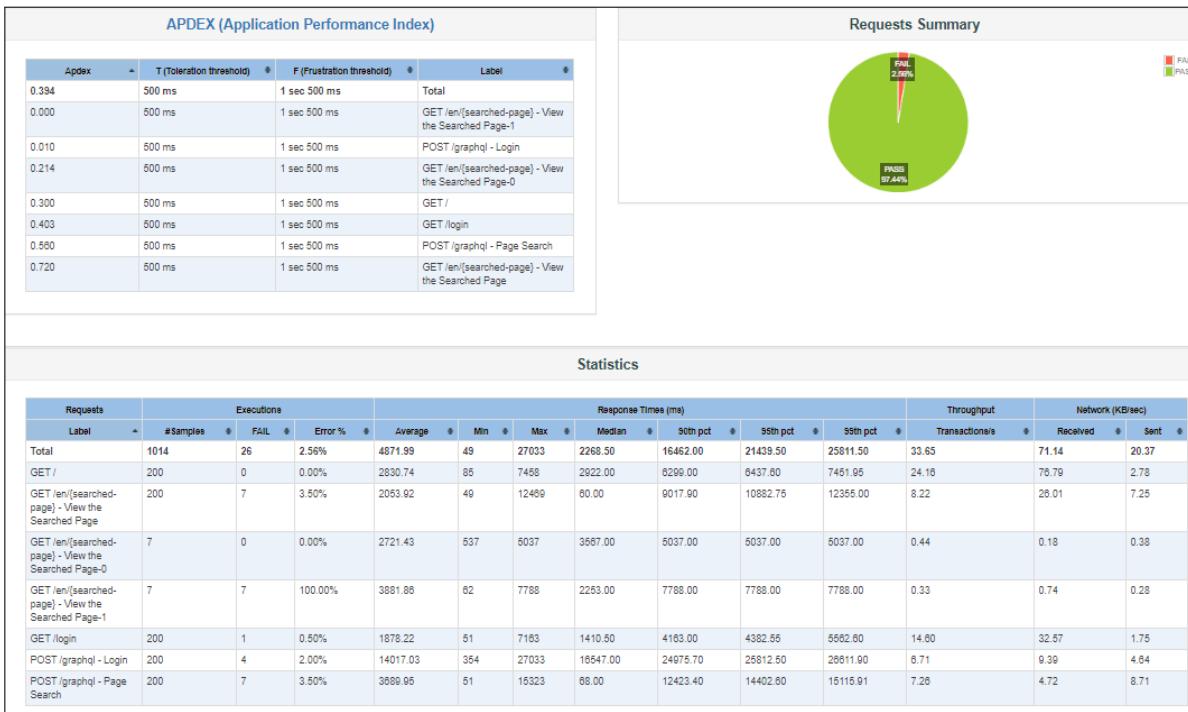


Figura 72: Estatísticas da arquitectura implementada com o Teste 3 para 200 threads

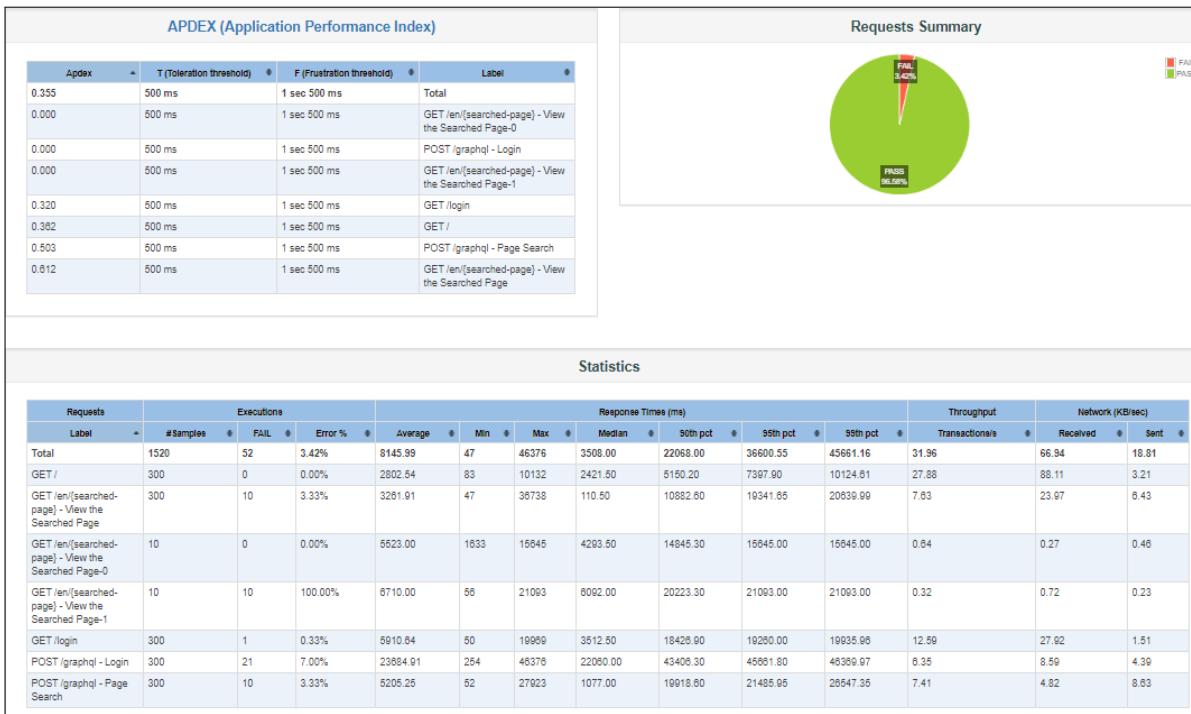


Figura 73: Estatísticas da arquitetura implementada com o Teste 3 para 300 threads

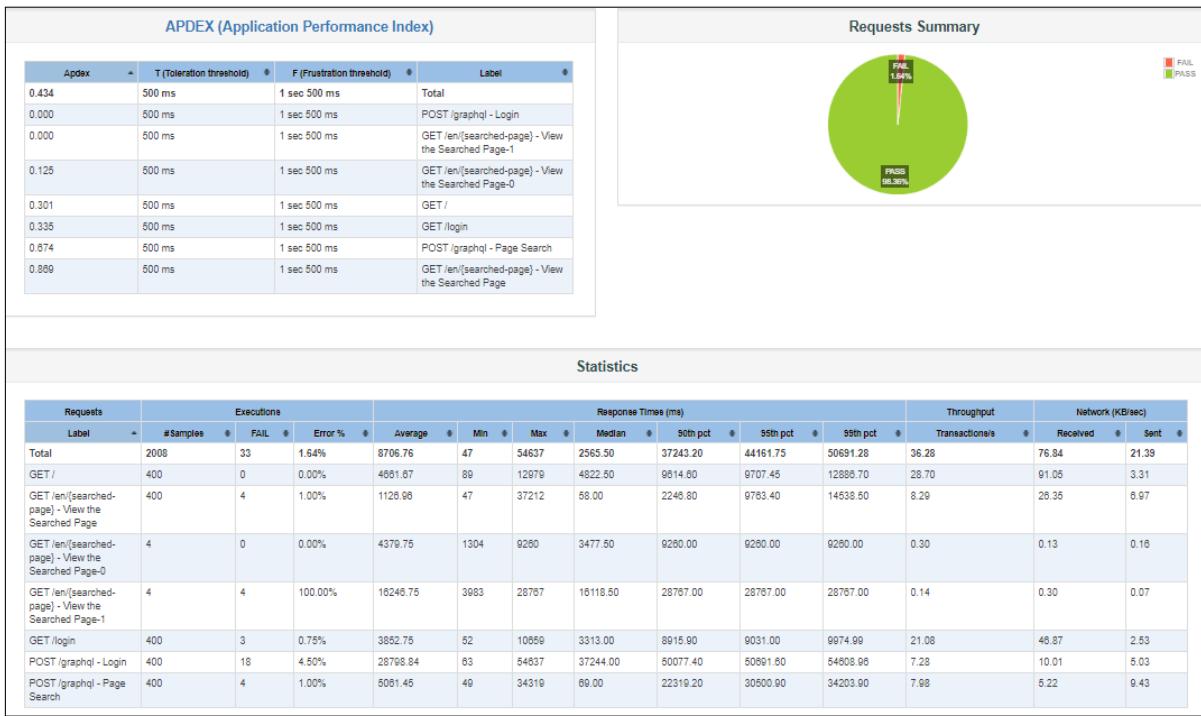


Figura 74: Estatísticas da arquitectura implementada com o Teste 3 para 400 threads

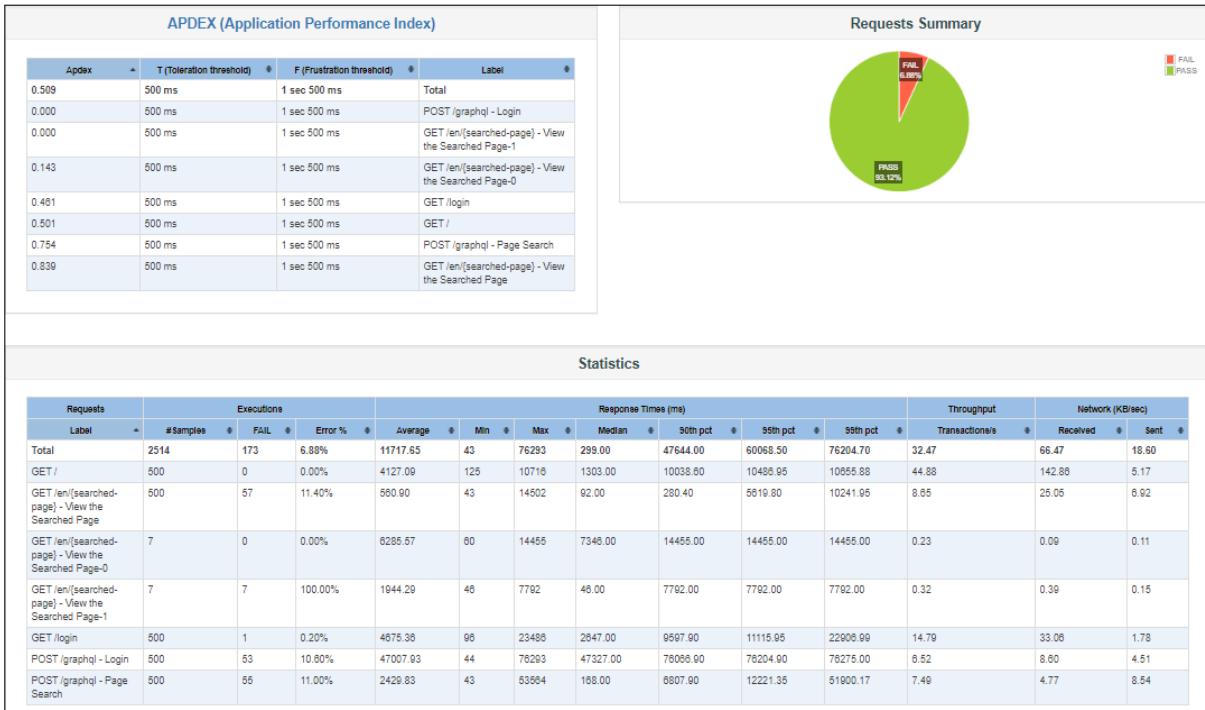


Figura 75: Estatísticas da arquitectura implementada com o Teste 3 para 500 *threads*

#### 5.5.4 Teste 4: Login + Criação de Utilizador

Foi aplicado o quarto teste para 100, 200 , 300 , 400 e 500 *threads* sobre a nossa nova arquitetura. Os resultados obtidos estão nas figuras 76, 77, 78, 79 e 80 para cada número de *threads* respectivamente.

Comparativamente aos resultados da subsecção 4.3.5 houve novamente uma melhoria considerável nos tempos de resposta e no número de transações por segundo para todos os testes.

Apesar de haver um ligeiro aumento de erros com 100 e 200 *threads*, quando sobrecarregamos ainda mais o sistema com 300 a 500 *threads*, a nova arquitetura apresenta muitos melhores resultados. Com 300 *threads* desceu desde 19.73% para 2.8%, com 400 de 25% para 3.77% e com 500 de 31.78% para 7.06% havendo assim mais utilizadores a serem inseridos na nossa base de dados.

Devido ao facto de que os resultados foram superiores em geral, pode-se assim concluir que a nova arquitetura fornece um melhor desempenho à plataforma.

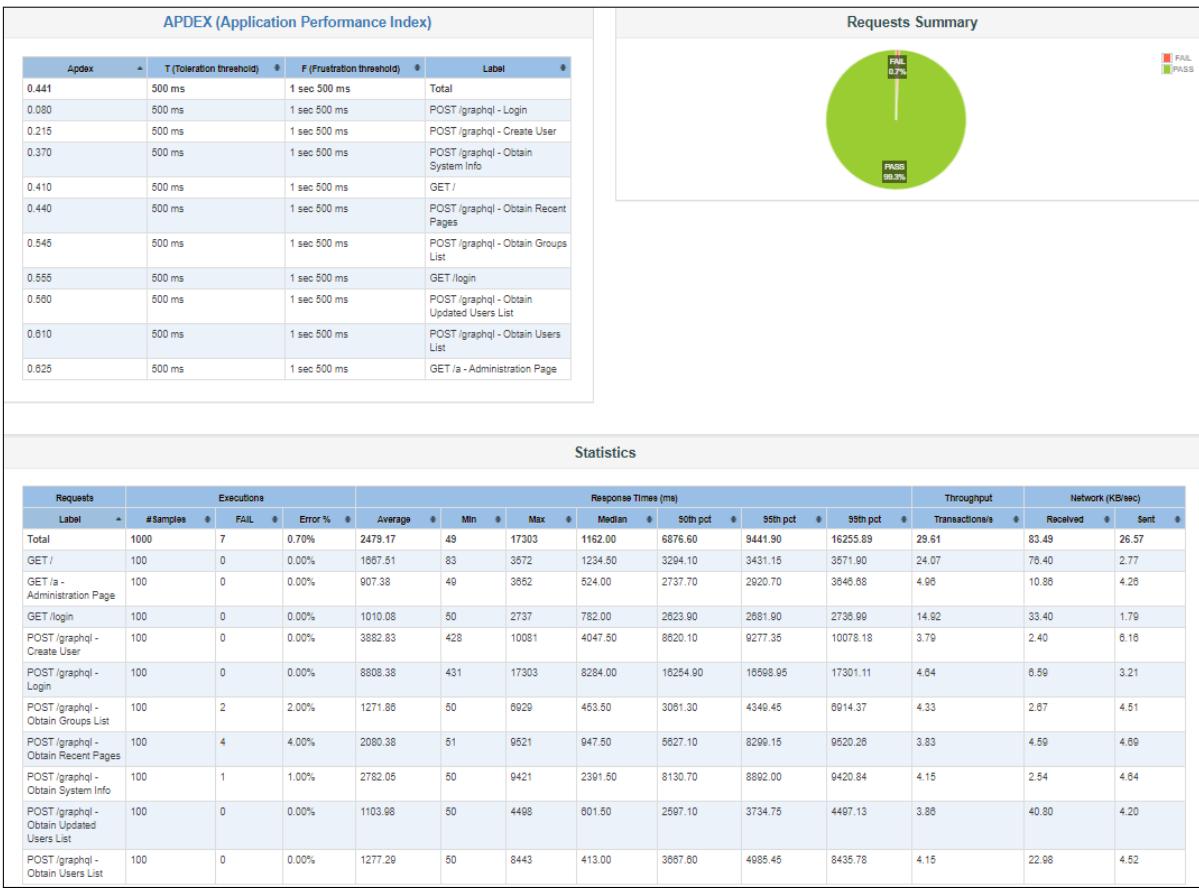


Figura 76: Estatísticas da arquitetura implementada com o Teste 4 para 100 threads

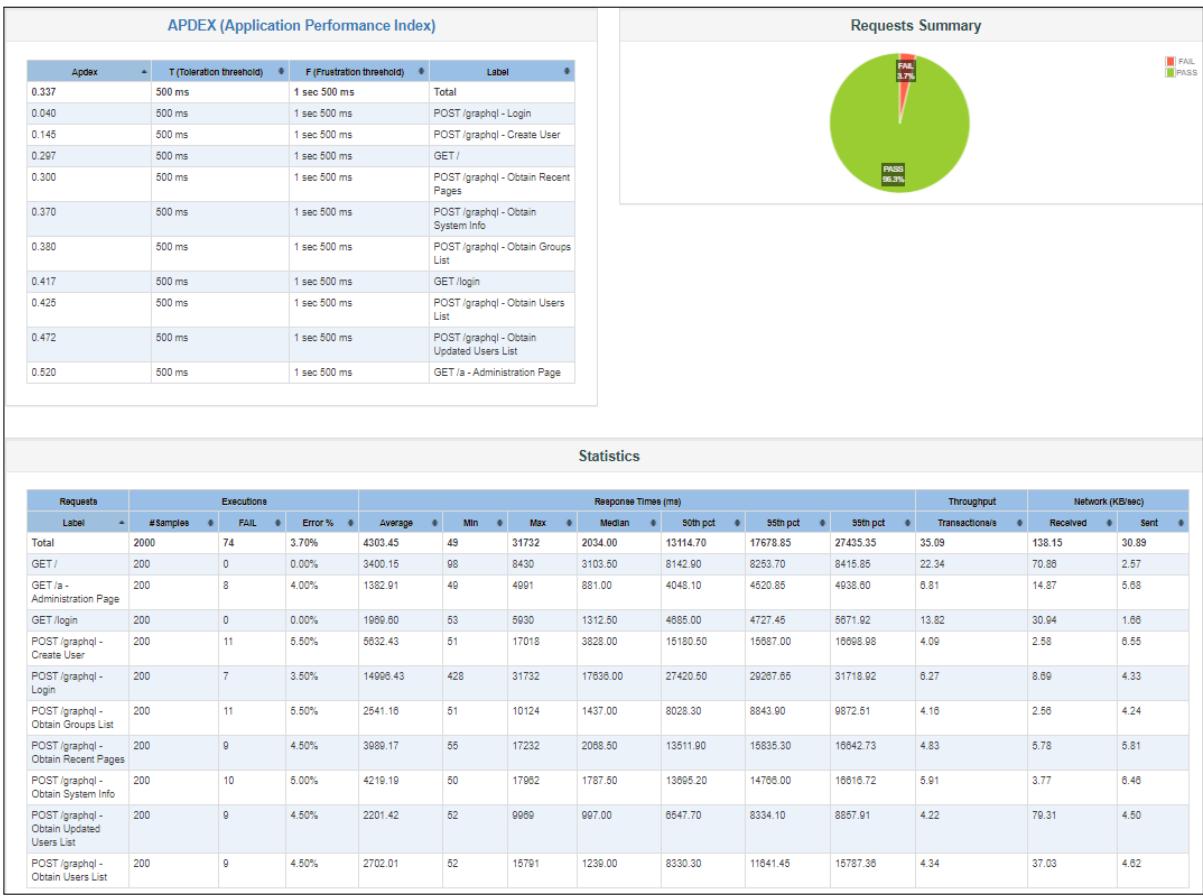


Figura 77: Estatísticas da arquitectura implementada com o Teste 4 para 200 threads

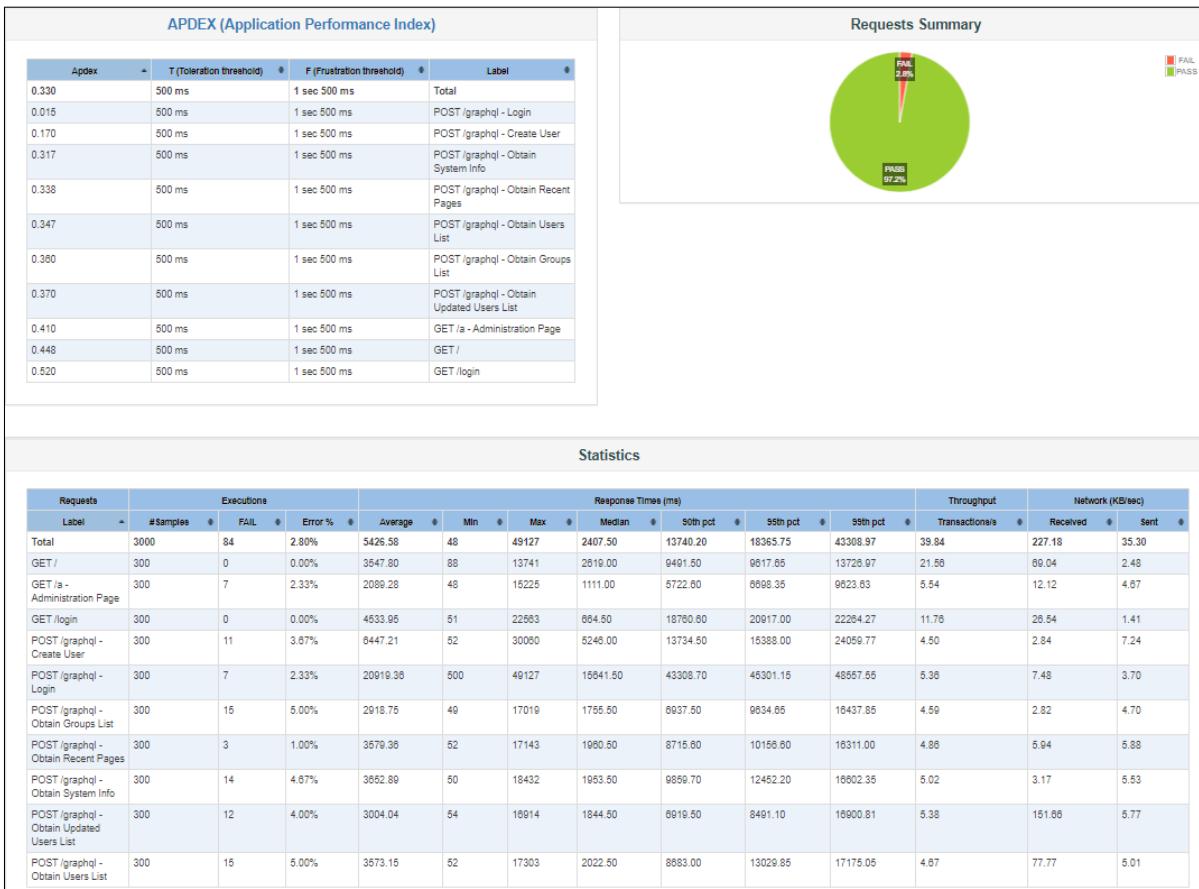


Figura 78: Estatísticas da arquitectura implementada com o Teste 4 para 300 threads

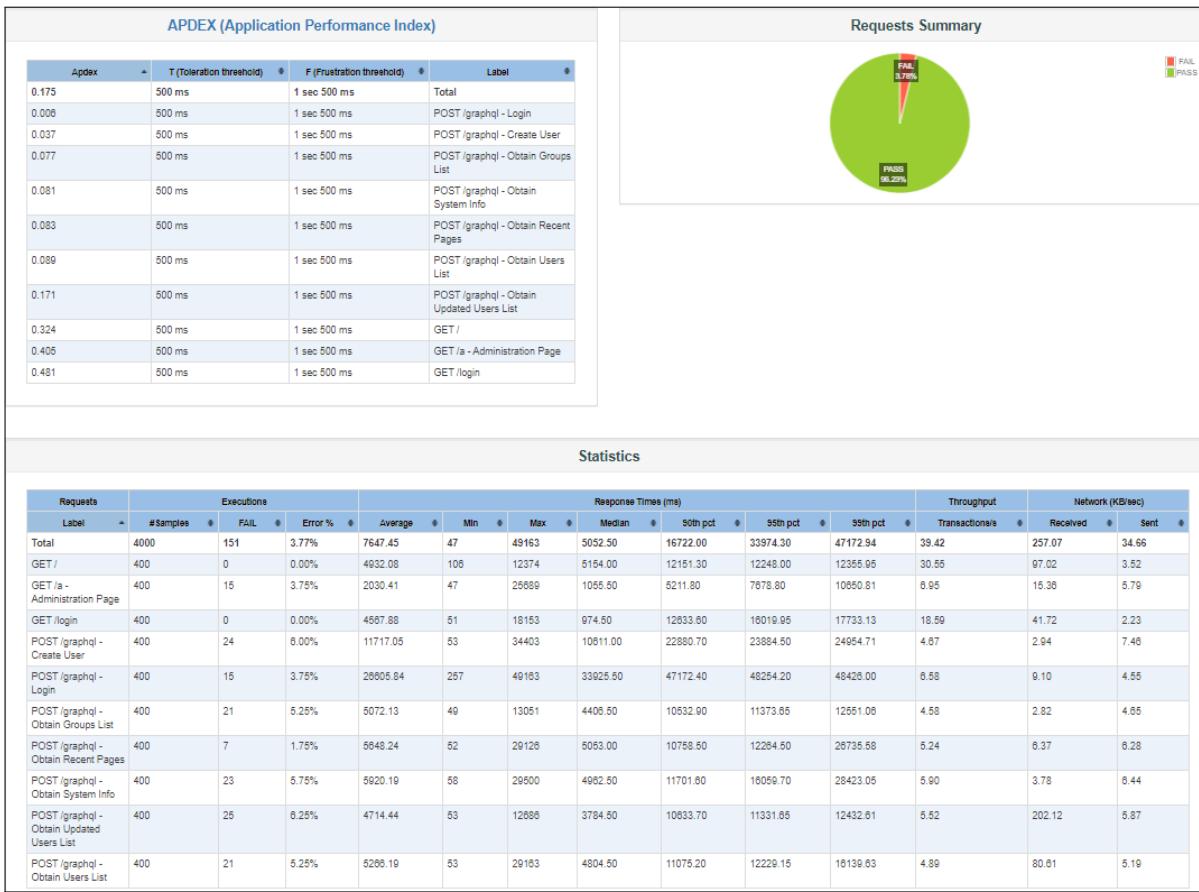


Figura 79: Estatísticas da arquitectura implementada com o Teste 4 para 400 threads

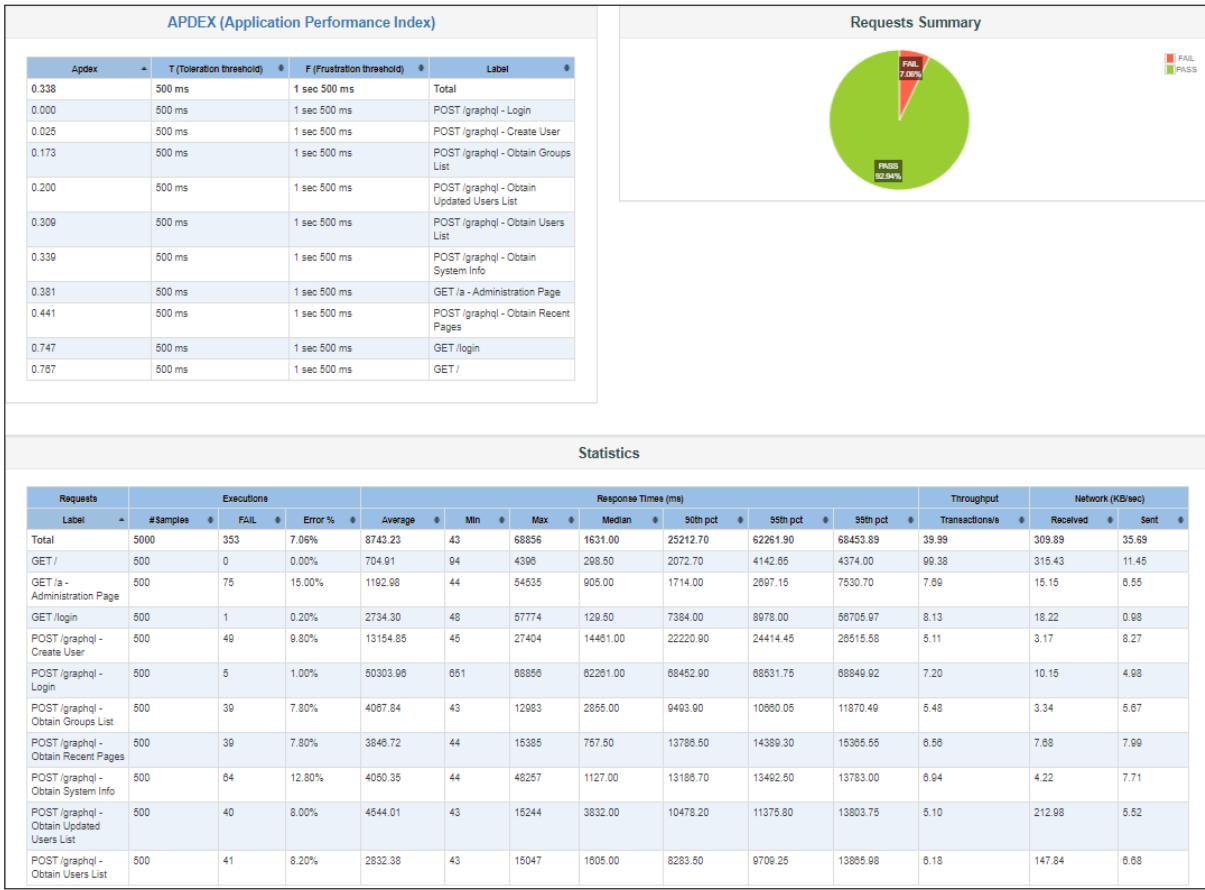


Figura 80: Estatísticas da arquitectura implementada com o Teste 4 para 500 threads

### 5.5.5 Teste 5: Login + Consulta de todas as Páginas do Sistema

Como se pode observar nos resultados obtidos com a nova configuração, todos os valores de tempo de resposta diminuíram substancialmente, sendo que a maior parte dos testes anteriores rondavam os 11s de tempo de resposta e passaram para valores entre os 4 e 7 segundos. Os valores de débito aumentaram em cerca de 30 tx/s e as percentagens de erro diminuíram em todos os testes feitos à excepção do teste com 500 threads.

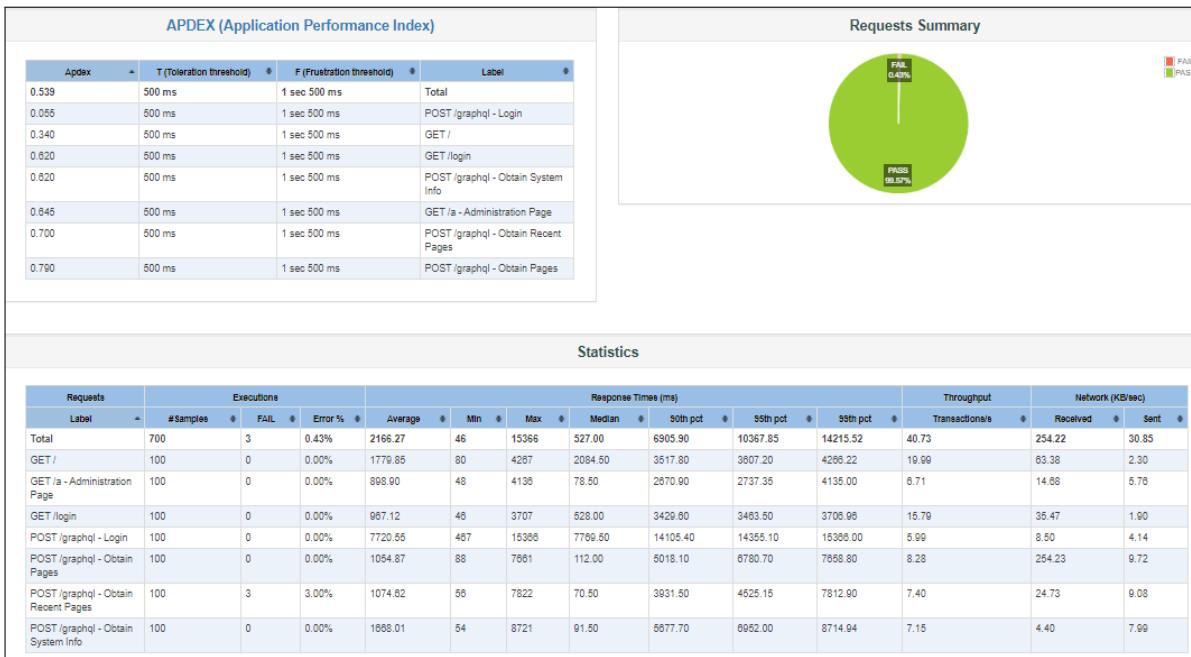


Figura 81: Estatísticas da arquitectura implementada com o Teste 5 para 100 threads

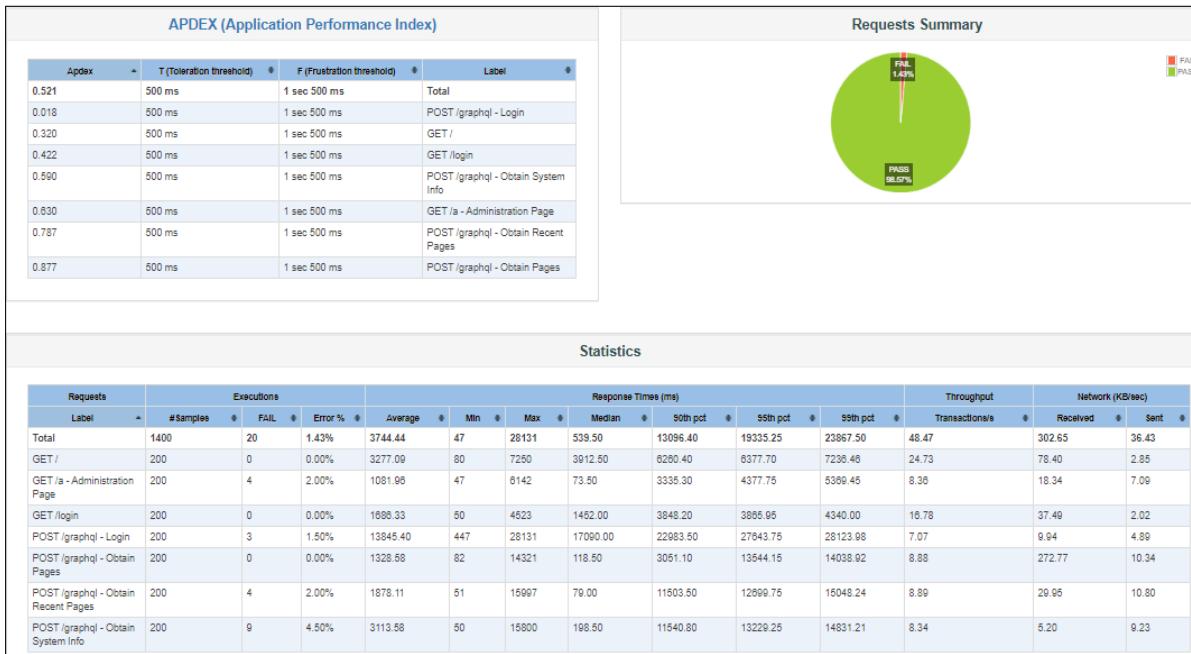


Figura 82: Estatísticas da arquitectura implementada com o Teste 5 para 200 threads

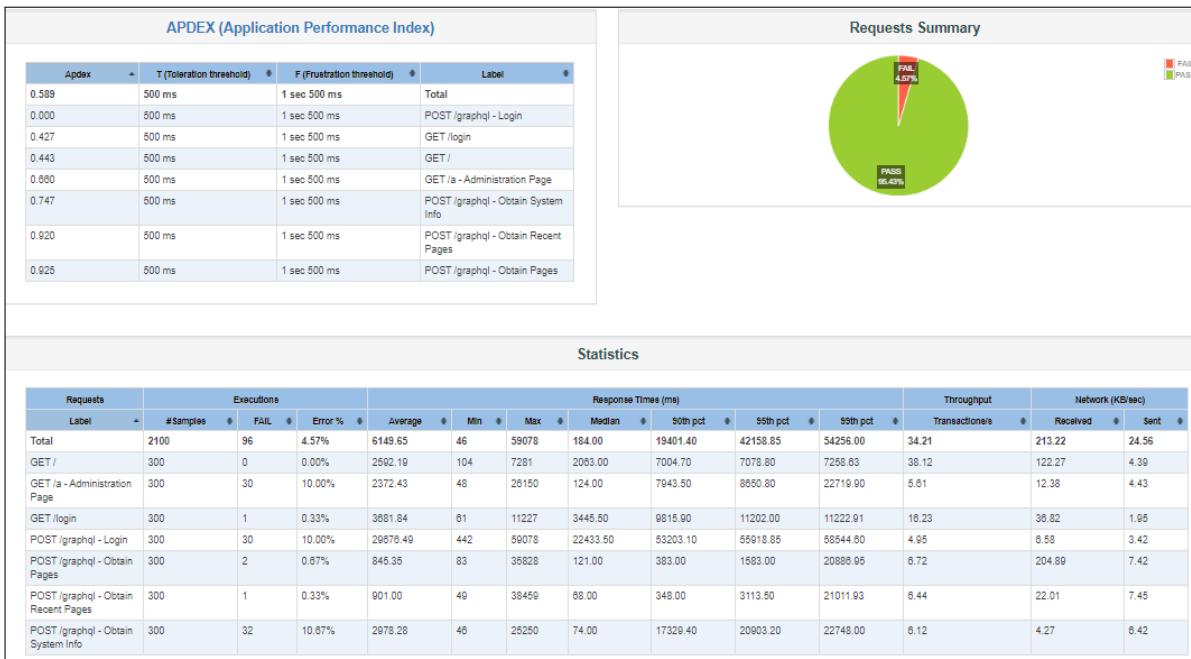


Figura 83: Estatísticas da arquitectura implementada com o Teste 5 para 300 threads

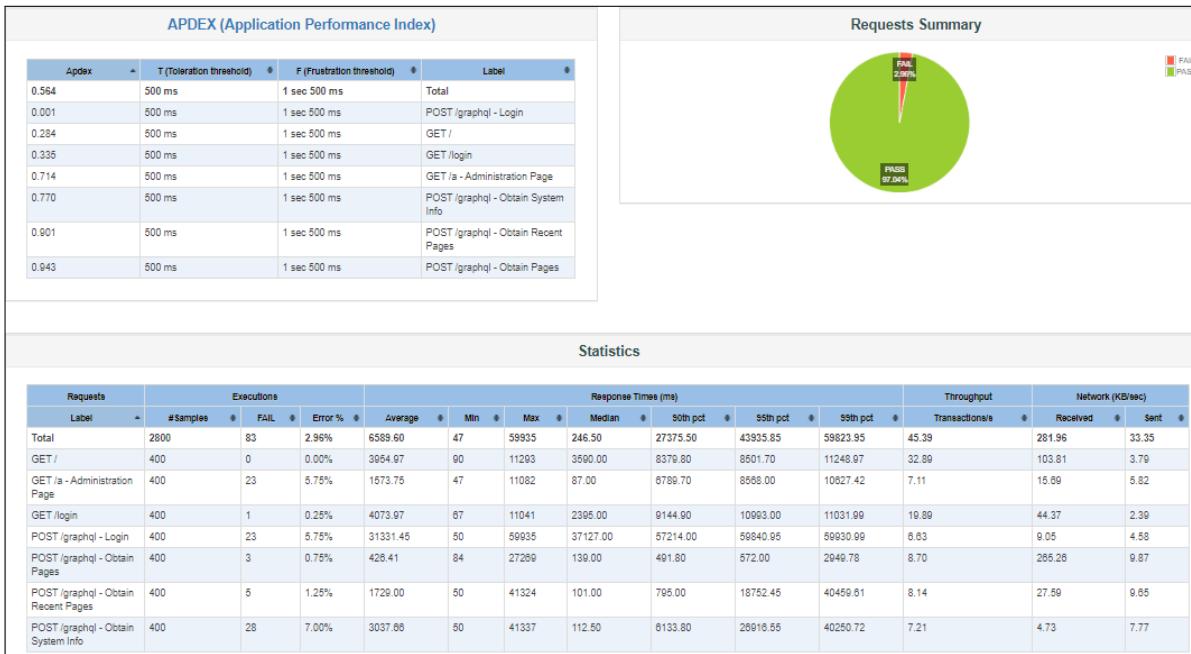


Figura 84: Estatísticas da arquitectura implementada com o Teste 5 para 400 threads

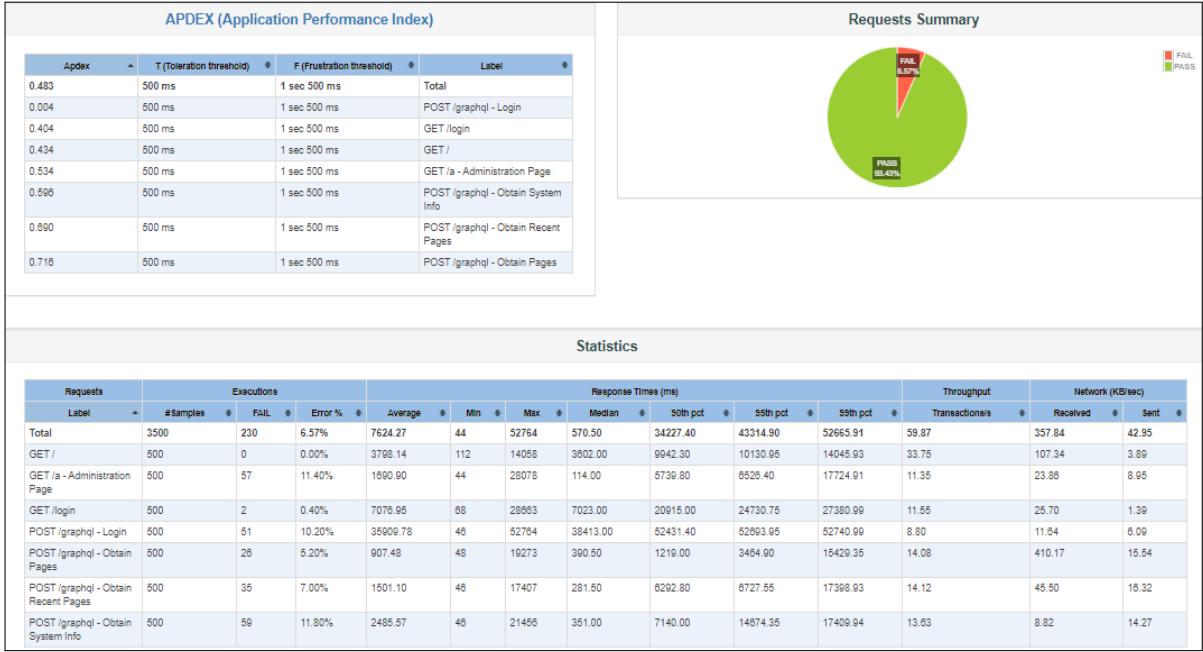


Figura 85: Estatísticas da arquitectura implementada com o Teste 5 para 500 threads

### 5.5.6 Teste 6: Login + Pesquisa + Escolha de Página + Save

Como se pode observar pelos resultados obtidos, embora no teste com 100 threads se tenha obtido 1% de percentagem de erro a mais face ao teste inicial, consegue observar-se a uma redução drástica do *response time*.

Esta redução do *response time* é visível em todas as execuções (100, 200, 300, 400 e 500 threads) e deve-se sem dúvida à utilização da replicação dos servidores aplicacionais.

A percentagem de erros nos diferentes testes diminuiu ligeiramente, pelo que assumimos que esta percentagem de erro possa ser por causa das verificações de integridade feitas pelo balanceador de carga externo, uma vez que caso uma máquina não esteja a responder aos pedidos de verificação de integridade, esta é considerada indisponível e só passados 5 segundos é que se faz uma nova verificação. Ou então outra razão é o facto de existir grande concorrência nas escritas na base de dados.

No geral neste teste 6, observou-se uma ligeira redução da percentagem de erro face aos resultados obtidos inicialmente e observou-se uma extrema melhoria no response time, considerando por isso que a nossa arquitetura ajudou bastante na otimização do *response time*.

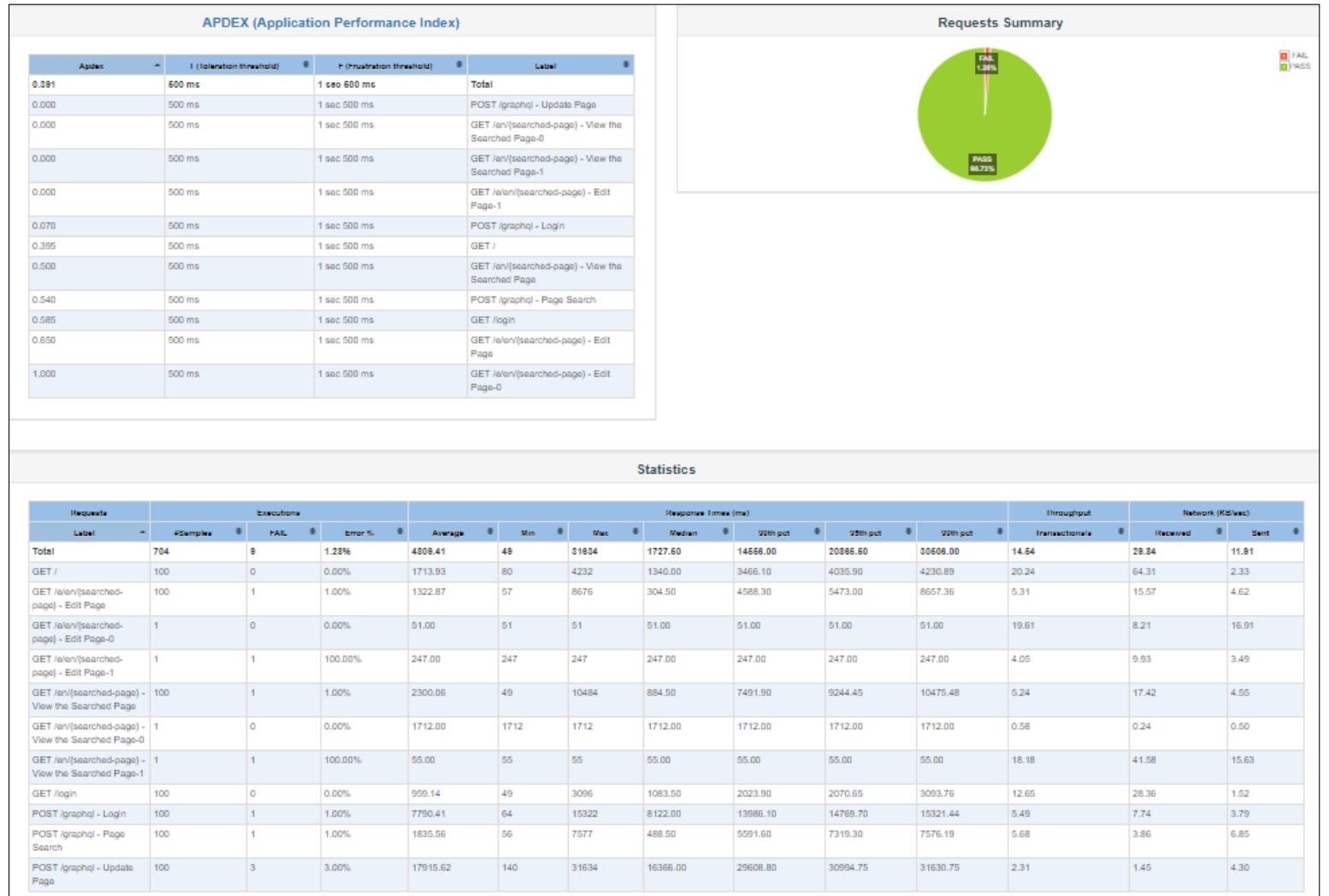


Figura 86: Estatísticas da arquitectura implementada com o Teste 6 para 100 threads

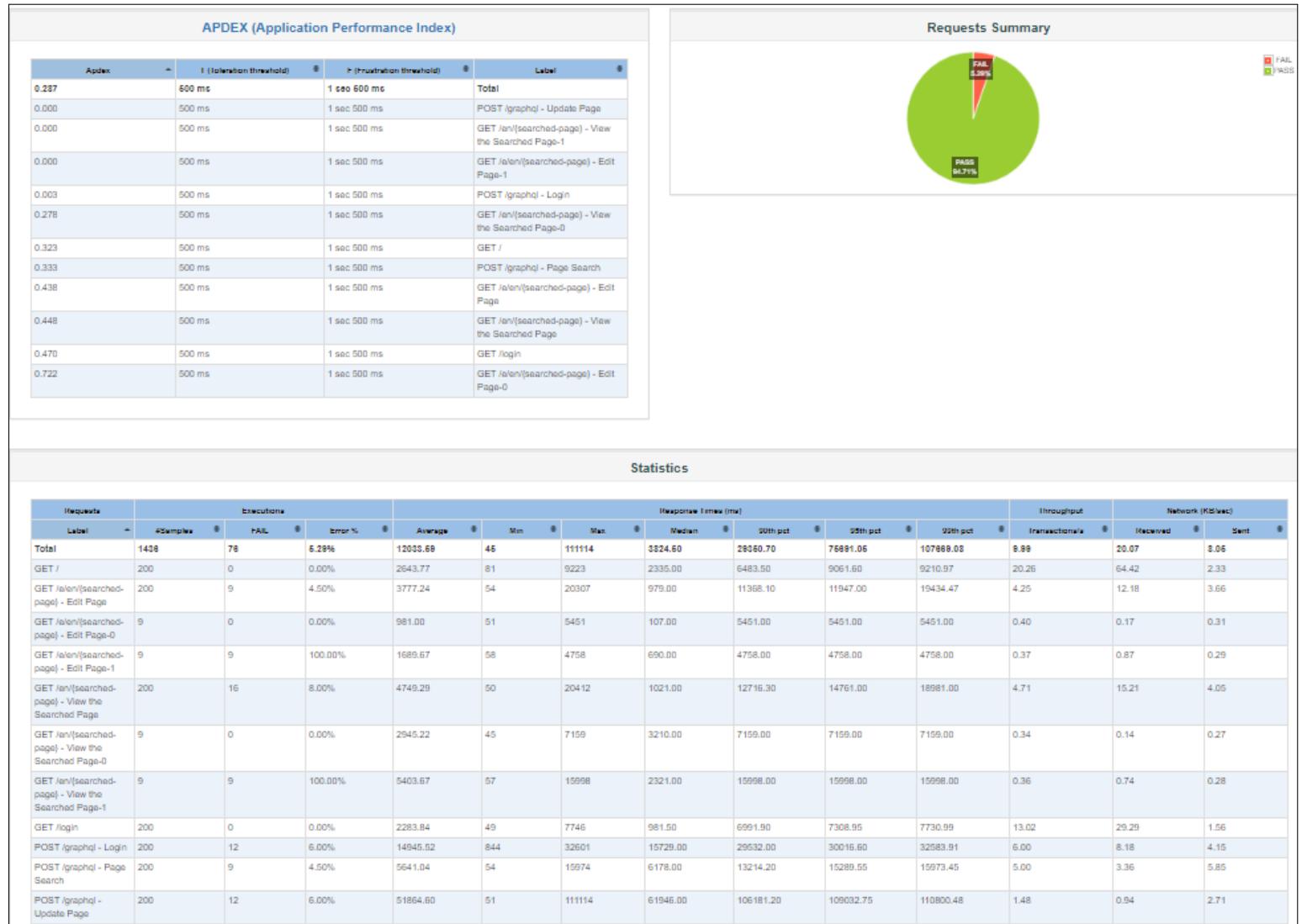


Figura 87: Estatísticas da arquitectura implementada com o Teste 6 para 200 threads

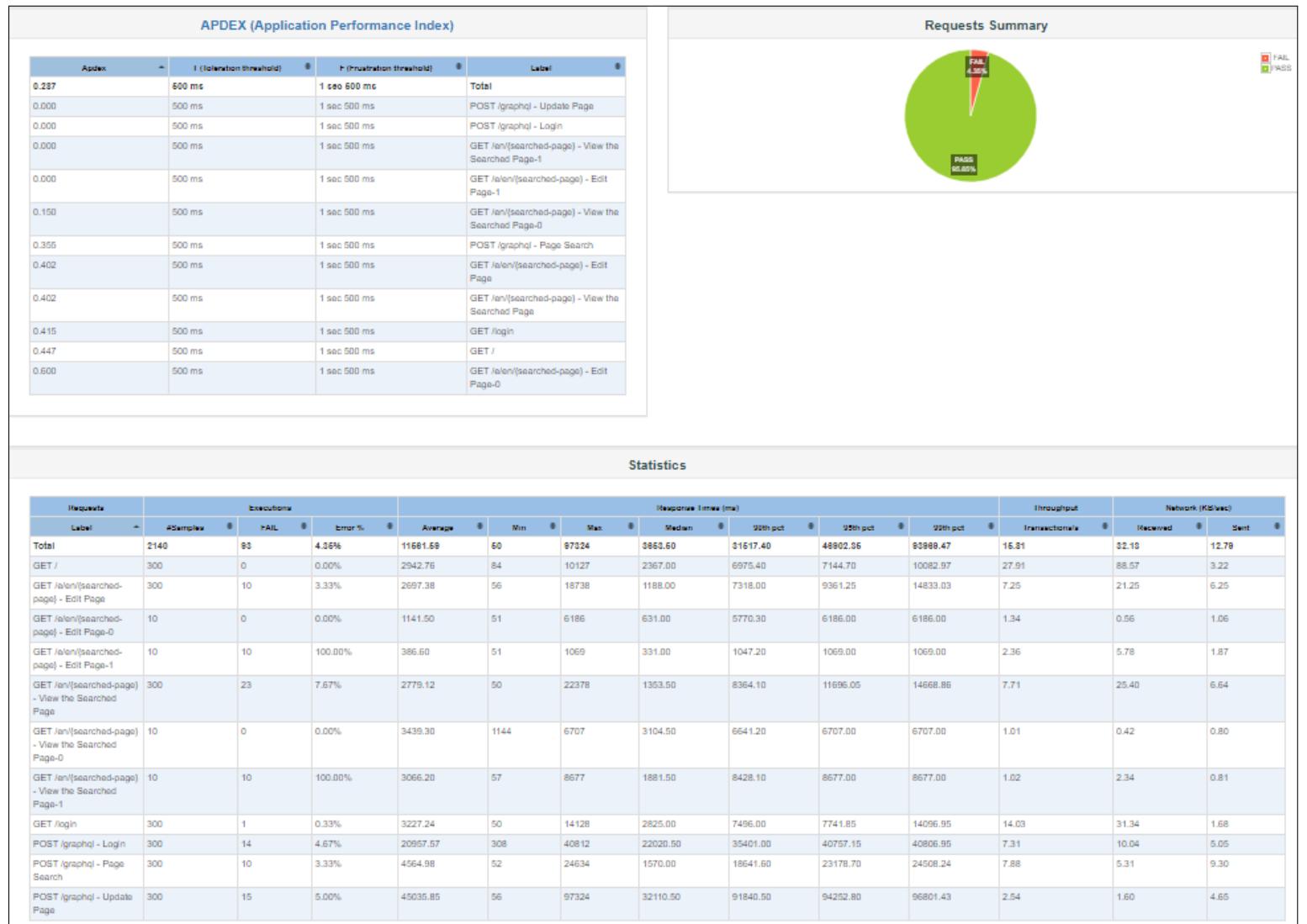


Figura 88: Estatísticas da arquitectura implementada com o Teste 6 para 300 threads

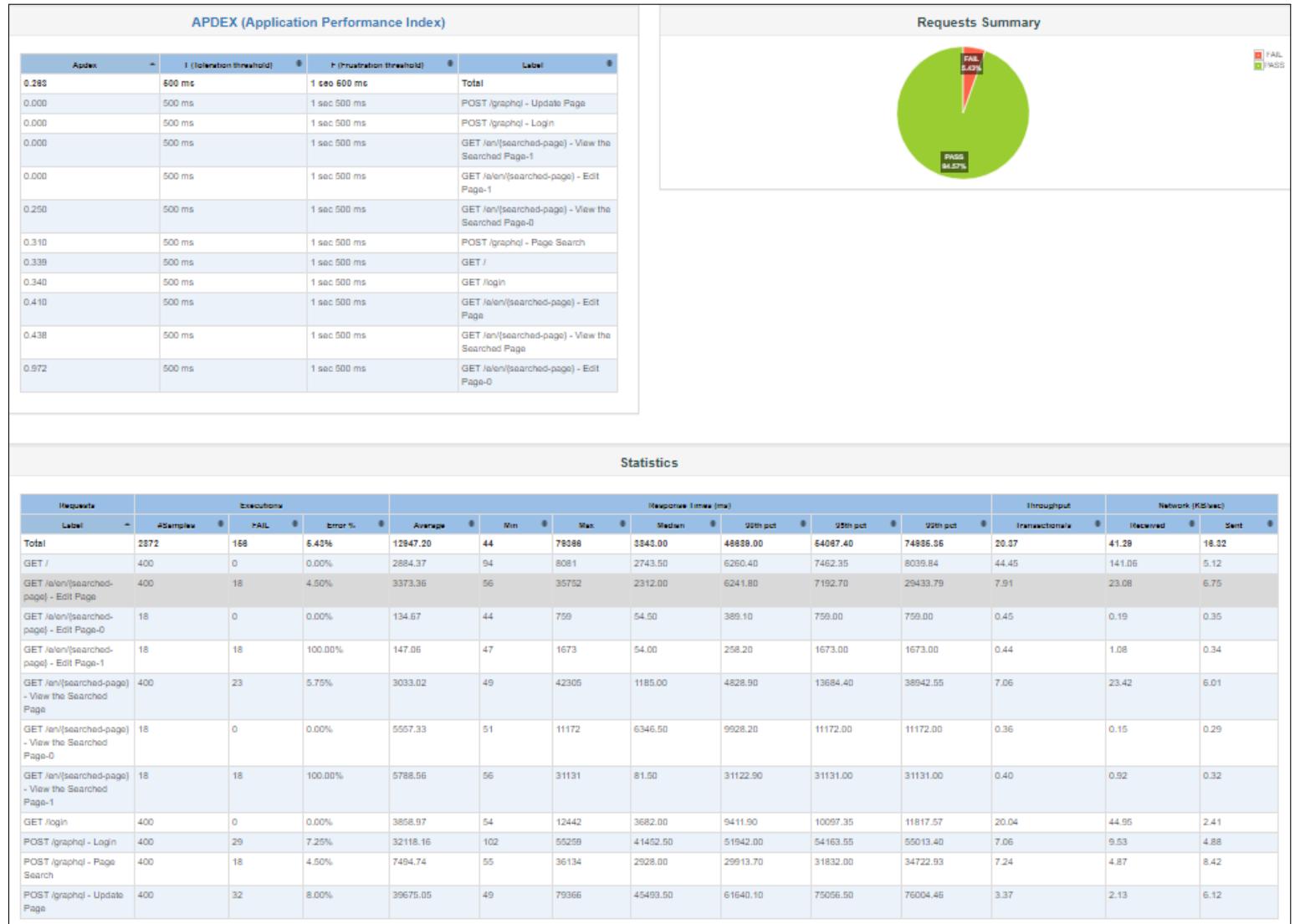


Figura 89: Estatísticas da arquitectura implementada com o Teste 6 para 400 threads

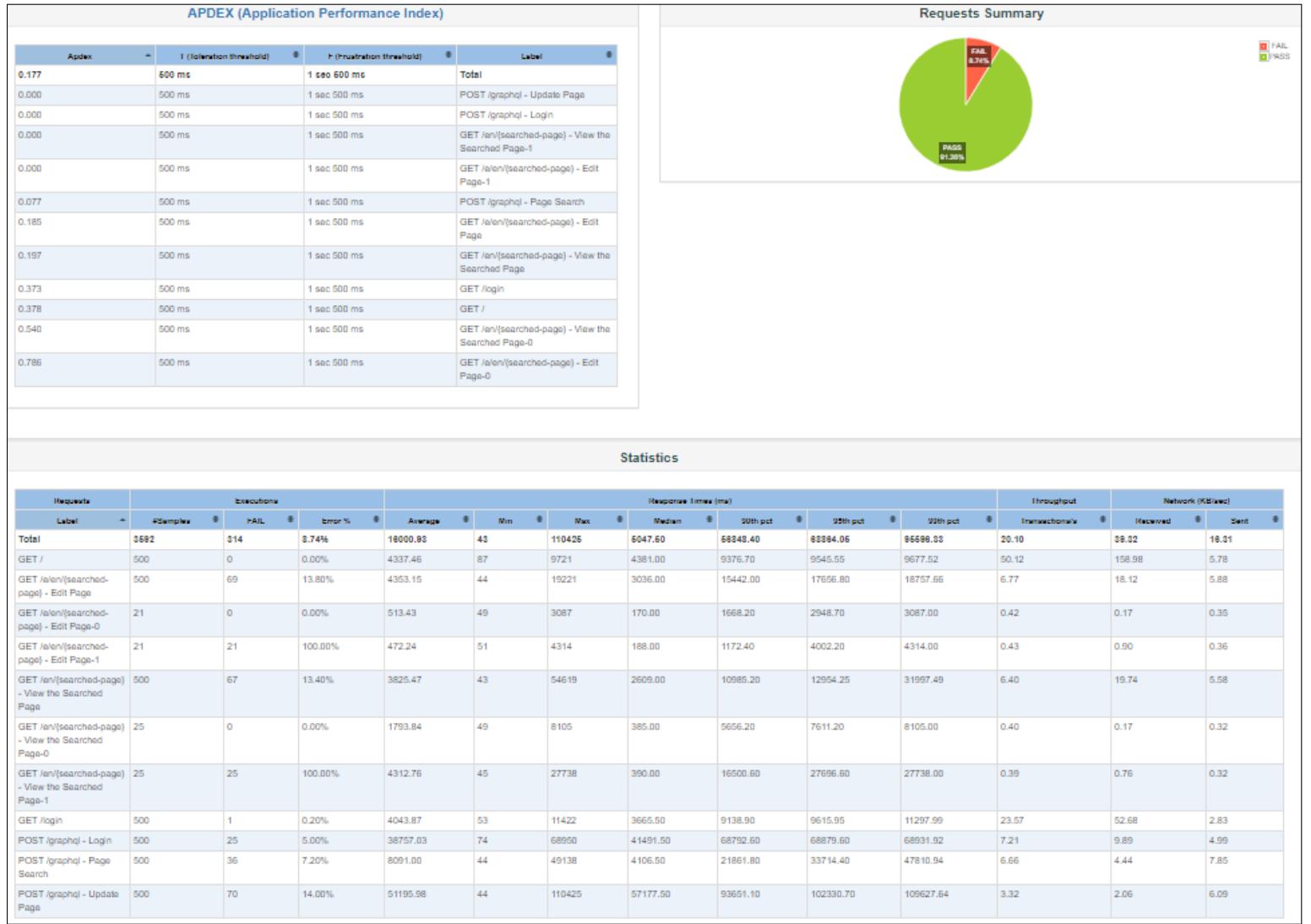


Figura 90: Estatísticas da arquitectura implementada com o Teste 6 para 500 *threads*

## 5.6 Análise dos resultados

Uma vez implementada a nova arquitectura, verifica-se que praticamente todos os valores de tempos de resposta diminuíram e os valores do débito aumentaram, há excepção dos tempos de resposta do Teste 2 para 400 e 500 *threads* e do Teste 6 para 500 *threads* que são praticamente iguais aos testes feitos inicialmente. Como exemplo, no teste 1 com 100 *threads* obteve-se um tempo de resposta de 11 s e débito de 7 tx/s na configuração inicial e, na configuração final, obteve-se 3s e 18 tx/s, respetivamente. Um outro exemplo é o do teste 2 com 200 *threads*, onde se tinha obtido um tempo de resposta de 38 s e débito de 5 tx/s e, posteriormente com a nova

arquitectura, obteve-se, respetivamente, 11 s e 13 tx/s. Quanto aos erros obtidos, observou-se que os testes que tinham uma percentagem de erros quase nula aumentaram ligeiramente a mesma mas, em contrapartida, os testes que apresentavam uma percentagem de erros elevada, como era o caso do Teste 2 e Teste 3, diminuíram consideravelmente esses valores.

## 6 Conclusão

Numa primeira fase do trabalho prático foi apresentada uma configuração inicial de uma infraestrutura capaz de hospedar a aplicação *Wiki.js*. O objectivo de ter uma configuração inicial foi ter um ponto de referência de comparação com uma infraestrutura que apresenta alta disponibilidade e tolerância a faltas. Dado isto, procedeu-se à criação e execução de uma bateria de testes usando a ferramenta *JMeter*. Com isto foi possível fazer testes de carga à aplicação e verificar qual o comportamento da mesma dado um número variável e elevado de clientes. Verificou-se, como era esperado, que a infraestrutura não suporta um número de clientes elevado eficientemente dado os tempos de respostas altos e os débitos muito baixos.

De seguida, foi apresentada uma arquitectura que apresenta alta disponibilidade e tolerância a faltas. Este objectivo foi atingido usando diversos mecanismos de replicação e de recuperação de *failovers*. Através dos testes feitos a esta nova arquitetura, constatou-se que houveram melhorias em relação aos tempos de resposta, débito e percentagem de erros. Estas melhorias devem-se ao facto da carga estar a ser balanceada entre diferentes máquinas, algo que não era possível com a configuração inicial do sistema.

As melhorias obtidas face à configuração inicial são significativas, no entanto, seria possível obter melhores resultados caso se considerasse uma configuração melhor nas máquinas virtuais escolhidas (mais RAM, mais CPU e por exemplo utilização de discos SSD).

Apesar das melhorias obtidas, reparamos que o tempo de migração dos serviços de um nodo do nosso *cluster* em caso de falha é bastante elevado, e tal como foi várias vezes mencionado nas aulas desta unidade curricular: "downtime tem custos". Uma solução futura para este problema seria por exemplo utilizar diferentes zonas com o nosso *cluster* e introduzir replicação ao nível da base de dados, garantindo que em caso de falha numa zona, poderíamos utilizar a base de dados replicada noutra.

Por fim, pode-se concluir que o objectivo do trabalho foi cumprido na sua totalidade e os resultados obtidos foram bastante satisfatórios.

## 7 Anexos

### Script das templates das instâncias Wiki.js

```
1 #!/bin/bash
2 sudo apt-get update
3 echo y | sudo apt install nodejs
4 echo wikijs | sudo adduser wikijs
5 cd /home/wikijs/
6 sudo mkdir wikijs
7 cd wikijs/
8 sudo wget https://github.com/Requarks/wiki/releases/download/2.5.170/wiki-js
     .tar.gz
9 sudo tar -xf wiki-js.tar.gz
10 sudo mv config.sample.yml config.yml
11 sudo sed -i 's/port: 3000/port: 10000/' config.yml
12 sudo sed -i 's/type:.*/type: postgres/' config.yml
13 sudo sed -i 's/host:.*/host: 10.132.0.50/' config.yml
14 sudo sed -i 's/user:.*/user: wikijsuser/' config.yml
15 sudo sed -i 's/pass:.*/pass: wikijspassword/' config.yml
16 sudo sed -i 's/db: wiki/db: wikijsdb/' config.yml
17 sudo sed -i 's/ha: false/ha: true/' config.yml
18 sudo sed -i '0,/limit: 5/s//limit: 500/' server/graph/schemas/authentication
     .graphql
19 sudo chown -R wikijs:wikijs /home/wikijs/wikijs/
20 echo $'[Unit]\nDescription=Wiki.js\nAfter=network.target\n\n[Service]\nType=
     simple\nExecStart=/usr/bin/node server\nRestart=always\nUser=wikijs\
     \nEnvironment=NODE_ENV=production\nWorkingDirectory=/home/wikijs/wikijs\n\n
     [Install]\nWantedBy=multi-user.target' | sudo tee -a /etc/systemd/system
     /wiki.service
21 sudo systemctl daemon-reload
22 sudo systemctl start wiki
23 sudo systemctl enable wiki
```

Listing 1: Script das templates das instâncias Wiki.js

# Criação do serviço de inicialização do *DRBD*

## *Script* com os comandos de inicialização

```
1 #!/bin/bash
2
3 sudo drbdadm up d1
4 sudo drbdadm --force primary d1
5 sudo systemctl restart target
```

Listing 2: init-drbd.sh

## Criação do serviço

```
1 [Unit]
2 Description=drbd start service
3 After=network.target
4
5 [Service]
6 Type=simple
7 ExecStart=/usr/bin/initdrbd.sh
8 Restart=always
9
10 [Install]
11 WantedBy=multi-user.target
```

Listing 3: updrbd.service

# Ansible para *deployment* do Wiki.js

## *Playbook*

```
1 ---
2 - name: DB install and start
3   hosts: database_server
4   become: yes
5   roles:
6     - db
7   tags:
8     - dbinstall
9
10 - name: WikiJS configure, install and deploy
11   hosts: wikijs_webserver
```

```
12   become: yes
13   roles:
14     - wikijs
15   tags:
16     - appinstall
```

Listing 4: Playbook (playbook.yml)

## Inventário de Hosts

```
1 [database_server]
2 35.187.35.22
3
4 [wikijs_webserver]
5 34.77.156.245
```

Listing 5: Inventário de Hosts (hosts.inv)

## *Group Vars*

```
1 ---
2 # Wiki JS database settings
3 postgresql_version: 12
4 wikijs_db_type: postgres
5 wikijs_db_ip: 10.132.0.19
6 wikijs_db_port: 5432
7 wikijs_db_name: wiki
8 wikijs_db_user: wikijs
9 wikijs_db_password: wikijsrocks
```

Listing 6: Group Vars (all.yml)

## *Roles*

### 1 - database

#### 1.1 - tasks

```
1 ---
2 - name: Install and configure PostgreSQL
3   apt:
4     name: "{{ item }}"
5     update_cache: yes
```

```

6   state: present
7   loop: "{{ db_packages }}"
8
9 - name: Configure PostgreSQL. Set listen_addresses.
10  lineinfile: dest=/etc/postgresql/{{ postgresql_version }}/main/postgresql.
11    conf
12    regexp="listen_addresses = " line="listen_addresses = '*' " state=present
13    notify: restart postgresql
14
15 - name: Configure allowed hosts.
16  lineinfile:
17    path: /etc/postgresql/{{ postgresql_version }}/main/pg_hba.conf
18    line: host all all 0.0.0.0/0 trust
19    create: yes
20
21 - name: Install psycopg2
22  pip: name=psycopg2
23
24 - name: Ensure Wiki.js database is created
25  become: true
26  become_user: postgres
27  postgresql_db:
28    name: '{{ wikijs_db_name }}'
29    encoding: 'UTF-8'
30    lc_collate: 'en_US.UTF-8'
31    lc_ctype: 'en_US.UTF-8'
32    template: 'template0'
33    state: present
34
35 - name: Ensure user has access to the database
36  become: true
37  become_user: postgres
38  postgresql_user:
39    db: '{{ wikijs_db_name }}'
40    name: '{{ wikijs_db_user }}'
41    password: '{{ wikijs_db_password }}'
42    priv: ALL
43    state: present

```

Listing 7: Tasks do *role database*

## 1.2 - vars

```

1 ---
2 db_packages:
3   - postgresql-{{ postgresql_version }}
4   - postgresql-client-{{ postgresql_version }}
5   - postgresql-contrib-{{ postgresql_version }}

```

```
6   - libpq-dev
7   - python3-pip
8   - acl
```

Listing 8: Vars do *role database*

### 1.3 - handlers

```
1 ---
2 - name: restart postgresql
3   service:
4     name: postgresql
5     state: restarted
```

Listing 9: Handlers do *role database*

## 2 - wikijs

### 2.1 - tasks

```
1 ---
2 - name: Install NodeJS
3   apt:
4     name: nodejs
5     update_cache: yes
6     state: present
7
8 - name: Create Wiki.js user
9   user:
10    name: "{{ wikijs_unix_user }}"
11    create_home: yes
12
13 - name: Create Wiki.js directory
14   file:
15    path: "{{ wikijs_dir }}"
16    state: directory
17    owner: "{{ wikijs_unix_user }}"
18
19 - name: Download Wiki.js distribution
20   get_url:
21    url: "https://github.com/Requarks/wiki/releases/download/2.5.170/wiki-js
22      .tar.gz"
23    dest: "{{ wikijs_dir }}/"
24    owner: "{{ wikijs_unix_user }}"
25
26 - name: Extract Wiki.js
```

```

26 unarchive:
27   src: "{{ wikijs_dir }}/wiki-js.tar.gz"
28   dest: "{{ wikijs_dir }}"
29   remote_src: true
30
31 - name: Create config file
32   copy:
33     src: "{{ wikijs_dir }}/config.sample.yml"
34     dest: "{{ wikijs_dir }}/config.yml"
35     remote_src: yes
36     owner: "{{ wikijs_unix_user }}"
37
38 - name: Wiki.js Configuration - bind port
39   lineinfile:
40     regexp: "^\s+port: \d*"
41     line: "  port: {{ wikijs_port }}"
42     path: "{{ wikijs_dir }}/config.yml"
43
44 - name: Wiki.js Configuration - database type
45   lineinfile:
46     regexp: "^\s+type: [a-z]*"
47     line: "    type: {{ wikijs_db_type }}"
48     path: "{{ wikijs_dir }}/config.yml"
49
50 - name: Wiki.js Configuration - database ip
51   lineinfile:
52     regexp: "^\s+host: [a-z0-9.]*"
53     line: "      host: {{ wikijs_db_ip }}"
54     path: "{{ wikijs_dir }}/config.yml"
55
56 - name: Wiki.js Configuration - database port
57   lineinfile:
58     regexp: "^\s+port: \d*"
59     line: "      port: {{ wikijs_db_port }}"
60     path: "{{ wikijs_dir }}/config.yml"
61
62 - name: Wiki.js Configuration - database user
63   lineinfile:
64     regexp: "^\s+user: \w*"
65     line: "        user: {{ wikijs_db_user }}"
66     path: "{{ wikijs_dir }}/config.yml"
67
68 - name: Wiki.js Configuration - database password
69   lineinfile:
70     regexp: "^\s+pass: \S*"
71     line: "        pass: {{ wikijs_db_password }}"
72     path: "{{ wikijs_dir }}/config.yml"
73
74 - name: Wiki.js Configuration - database name

```

```

75 lineinfile:
76   regexp: "^\s{2}db: \w*"
77   line: "  db: {{ wikijs_db_name }}"
78   path: "{{ wikijs_dir }}/config.yml"
79
80 - name: Give user permissions to the directory
81   shell: chown -R {{ wikijs_unix_user }}:{{ wikijs_unix_user }} {{ wikijs_dir }}
82
83 - name: Create Wiki.js Service
84   template:
85     src: "wikijs.service.j2"
86     dest: "/etc/systemd/system/wikijs.service"
87     owner: root
88     mode: '644'
89   notify:
90     - reload daemon
91     - enable wikijs
92     - start wikijs

```

Listing 10: Tasks do *role wikijs*

## 2.2 - vars

```

1 ---
2 wikijs_unix_user: "wikijs"
3 wikijs_dir: "/home/{{ wikijs_unix_user }}/wikijs"
4 wikijs_port: 10000

```

Listing 11: Vars do *role wikijs*

## 2.3 - handlers

```

1 ---
2 - name: reload daemon
3   command: systemctl daemon-reload
4
5 - name: enable wikijs
6   service:
7     name: wikijs
8     enabled: yes
9
10 - name: start wikijs
11   service:
12     name: wikijs

```

```
13 state: started
```

Listing 12: Handlers do *role wikijs*

## 2.4 - templates

```
1 [Unit]
2 Description=Wiki.js
3 After=network.target
4
5 [Service]
6 Type=simple
7 ExecStart=/usr/bin/node server
8 Restart=always
9 User={{ wikijs_unix_user }}
10 Environment=NODE_ENV=production
11 WorkingDirectory={{ wikijs_dir }}
12
13 [Install]
14 WantedBy=multi-user.target
```

Listing 13: Templates do *role wikijs*

# Selenium

```
1 import org.openqa.selenium.*;
2 import org.openqa.selenium.chrome.ChromeDriver;
3 import org.openqa.selenium.chrome.ChromeOptions;
4 import org.openqa.selenium.support.ui.ExpectedCondition;
5 import org.openqa.selenium.support.ui.ExpectedConditions;
6 import org.openqa.selenium.support.ui.WebDriverWait;
7
8 import java.time.LocalDateTime;
9 import java.time.format.DateTimeFormatter;
10 import java.util.ArrayList;
11 import java.util.Arrays;
12 import java.util.List;
13 import java.util.Random;
14
15 public class ChromeTester extends Thread {
16     /* --- Class variables --- */
17     public WebDriver driver;
18     public String url;
19     public double[] time;
20     public int index;
21
22     /**
23      * Parametrized Construtor
24      *
25      * @param driver Driver for browser
26      * @param time Execution time for operations
27      * @param index Number of operation
28      */
29     public ChromeTester(WebDriver driver, String url, double[] time, int
index) {
30         this.driver = driver;
31         this.url = url;
32         this.time = time;
33         this.index = index;
34     }
35
36     // --- Waits for loading a page
37     public void waitForPageLoaded() {
38         ExpectedCondition<Boolean> pageLoadCondition = driver1 -> {
39             assert driver1 != null;
40             return ( (JavascriptExecutor) driver1 ).executeScript( "
41             return document.readyState" ).equals( "complete" );
42         };
43         WebDriverWait wait = new WebDriverWait( driver, 30 );
44         wait.until( pageLoadCondition );
45         try {
```

```

45         Thread.sleep( 2000 );
46     } catch(InterruptedException ie) {
47         ie.printStackTrace();
48     }
49 }
50
51 // --- Login Behaviour
52 private void login(boolean close) throws Exception {
53     WebElement element;
54     driver.get( url + "login" );
55
56     waitForPageLoaded();
57
58     // -- Login
59     element = driver.findElement( By.id( "input-10" ) );
60     element.sendKeys( "jf5138@gmail.com" );
61     element = driver.findElement( By.id( "input-12" ) );
62     element.sendKeys( "sdb2020" );
63     element.sendKeys( Keys.ENTER );
64
65     if( close ) { // if the operation is only login, close the
66         browser after the operation
67         WebDriverWait wait = new WebDriverWait( driver, 30 );
68         wait.until( ExpectedConditions.urlMatches( "http
69 ://34.77.156.245:10000" ) );
70         Thread.sleep( 2000 );
71         driver.quit();
72     }
73 }
74
75 // --- ( Login + Create page ) behaviour
76 private void login_createPage() throws Exception {
77     WebElement element;
78     WebDriverWait wait = new WebDriverWait( driver, 30 );
79
80     // --- Login
81     login( false );
82
83     waitForPageLoaded();
84     wait.until( ExpectedConditions.urlMatches( "http
85 ://34.77.156.245:10000" ) );
86
87     Thread.sleep( 2000 );
88
89     // --- Click the page to remove left side bar
90     driver.findElement( By.xpath( "//html" ) ).click();
91
92     Thread.sleep( 1000 );

```

```

91         // --- Find create page button and click it
92         List<WebElement> buttons = driver.findElements( By.tagName( "button" ) );
93         buttons.get( 1 ).click();
94
95         Thread.sleep( 3000 );
96
97         // --- Enters name of new page based on current time
98         Random r = new Random();
99         DateTimeFormatter dtf = DateTimeFormatter.ofPattern( "yyyyMMddHHmmss" );
100        LocalDateTime now = LocalDateTime.now();
101        String format = dtf.format( now ) + r.nextInt();
102        element = driver.findElement( By.id( "input-103" ) );
103        element.sendKeys( format );
104
105        // --- Press select button
106        element = driver.findElement( By.xpath( "/html/body/div/div/div[3]/div/div/div[4]/div/button[2]" ) );
107        element.click();
108
109        wait.until( ExpectedConditions.urlMatches( "http://34.77.156.245:10000/e/en/new-page" + format ) );
110
111        Thread.sleep( 1000 );
112
113        // --- Choose html editor
114        element = driver.findElement( By.xpath( "/html/body/div/div/div[3]/div/div[1]/div/div[2]/div/div[3]/div/div/img" ) );
115        element.click();
116
117        Thread.sleep( 2000 );
118
119        // --- Define page name
120        element = driver.findElement( By.id( "input-161" ) );
121        element.sendKeys( format );
122
123        // --- Confirm
124        buttons = driver.findElements( By.tagName( "button" ) );
125        buttons.get( 17 ).click();
126
127        Thread.sleep( 1000 );
128
129        // --- Save page
130
131        buttons = driver.findElements( By.tagName( "i" ) );
132        buttons.get( 0 ).click();
133        waitForPageLoaded();
134        wait.until( ExpectedConditions.urlMatches( "http

```

```

: //34.77.156.245:10000/en/new-page" + format ) );
135
136     // --- Close browser
137     driver.quit();
138 }
139
140 // --- Thread work
141 @Override
142 public void run() {
143     long start = System.currentTimeMillis();
144
145     try{
146         //login( true );
147         login_createPage();
148     }catch(Exception e){
149         e.printStackTrace();
150         System.out.println("Thread " + Thread.currentThread().getName
());
151     }
152
153     long end = System.currentTimeMillis();
154
155     time[ index ] = ( end - start ) / 1000.0;
156 }
157
158 // --- Creates threads to simulate browser navigation and outputs time
report
159 public static void main(String[] args) throws Exception {
160     String url = "http://34.77.156.245:10000/";
161     List<double[]> report = new ArrayList<>();
162     int threadsExp = 4;
163     int runs = 1;
164     ChromeOptions options = new ChromeOptions();
165     options.addArguments( "headless" );
166
167     int i, j;
168     for(int r = 0 ; r < runs ; r++) {
169         for(i = 0; i < threadsExp ; i++) {
170             int numThreads = (int) Math.pow( 2, i );
171             report.add( new double[ numThreads ] );
172             ChromeTester[] chrome = new ChromeTester[ numThreads ];
173
174             for(j = 0; j < numThreads ; j++) {
175                 WebDriver driver = new ChromeDriver( options );
176                 chrome[ j ] = new ChromeTester( driver, url, report
.get( i ), j );
177             }
178             for(j = 0; j < numThreads ; j++) {
179                 chrome[ j ].start();

```

```

180
181         }
182         for(j = 0; j < numThreads ; j++) {
183             chrome[ j ].join();
184             System.out.println( "Ended " + j );
185         }
186     double m;
187     i = 0;
188     int threads;
189     for(double[] time : report) {
190         threads = (int) Math.pow( 2, i );
191         m = Arrays.stream( time ).sum() / threads;
192         System.out.println( "Threads " + threads + " : " + m + "
193 s" );
194         i++;
195     }
196     report.clear();
197 }
198 }
```

Listing 14: Templates do *role wikijs*