INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Reengenharia de software em MEAN para lidar com elasticidade da carga de utilização

BEATRIZ OLIVEIRA VAZ
Setembro de 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Reengenharia de software em MEAN para lidar com elasticidade da carga de utilização

Beatriz Oliveira Vaz

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Nuno Silva

Porto, setembro 2024

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 15 de setembro de 2024

Resumo

Este documento aborda os desafios associados à aplicação SWiPE, inicialmente desenvolvida para o processamento de ficheiros CSV e envio de emails. A evolução da aplicação promoveu o crescimento da complexidade da mesma, impactando a elasticidade e adaptabilidade às variações de carga de utilização.

Com a existência de períodos de utilização muito variáveis, as infraestruturas usadas tornam-se subaproveitadas. Este estudo propõe diferentes estratégias de reengenharia, com um principal foco na otimização de desempenho, adaptabilidade dinâmica e eficiência operacional.

Com base nas estratégias expostas, este estudo proporciona quatro cenários distintos propostos para enfrentar problemas com cargas de trabalho variáveis. Seguidamente, este documento analisa os resultados obtidos nos diferentes cenários e são tomadas as devidas conclusões.

Palavras-chave: elasticidade de software, reengenharia de software, adaptabilidade dinâmica, eficiência operacional

Abstract

This document addresses the challenges associated with the SWiPE application, initially developed for processing CSV files and sending emails. The evolution of the application has led to an increase in its complexity, impacting on its elasticity and adaptability to variations in usage load.

With highly variable periods of use, the infrastructures used become underutilized. This study proposes different reengineering strategies, with a main focus on performance optimization, dynamic adaptability and operational efficiency.

Based on the strategies presented, this study provides four different scenarios proposed to tackle problems with varying workloads. This document then analyzes the results obtained in the different scenarios and draws the appropriate conclusions.

Keywords: software elasticity, software reengineering, dynamic adaptability, operational efficiency

Agradecimentos

Em primeiro lugar, gostaria de agradecer a orientação e disponibilidade do professor Nuno Silva que se revelou fundamental para a elaboração do presente relatório. O seu apoio, os seus conselhos, os momentos de discussão que proporcionaram a melhoria do trabalho desenvolvido e a prontidão prestada relativamente às minhas questões foram, sem dúvida, extremamente importantes para a conclusão deste projeto.

Aos meus pais, agradeço por todo o apoio incondicional e pelos valores de persistência e dedicação que sempre me transmitiram. Agradeço pela vossa confiança em mim e por todos os sacrifícios que fizeram para que eu alcançasse os meus objetivos. Desde cedo ensinaram-me a importância do trabalho árduo e da resiliência e, por isso, sou eternamente grata.

À minha irmã, um agradecimento muito especial. Ao longo desta jornada foste mais que uma irmã, foste uma amiga e confidente, a que mais me incentivou a seguir em frente. Agradeço pelas palavras de apoio, pela paciência, pelo carinho e pela capacidade de estar sempre disponível, mesmo nos momentos mais difíceis. Este trabalho é também um reflexo do teu apoio incansável.

Gostava de agradecer às minhas amigas de infância, nomeadamente à Luísa, Mafalda e Mariana, que, mesmo com o passar do tempo, continuam a fazer parte da minha vida. A vossa amizade ao longo dos anos tem sido uma constante fonte de alegria e equilíbrio. O vosso apoio, mesmo que à distância, foi essencial para que me mantivesse motivada. É raro ter amizades que resistam ao tempo e estou muito agradecida por continuar a alcançar novas etapas ao vosso lado.

Aos meus colegas de curso, em especial ao Tiago Pinto, Pedro Lemos, Petra Pisco, Rodrigo Tavares e José Oliveira, obrigada pela amizade, pelo apoio mútuo e pelas muitas horas de trabalho em conjunto. Crescemos juntos durante este percurso e a vossa companhia tornou este desafio mais fácil e, certamente, mais divertido.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos

Lista de Acrónimos

CSV	Arquivos Comma-separed values, arquivos de texto em que os valores estão separados por vírgulas
UC	Unidade Curricular
REST	Representational State Transfer, estilo de arquitetura de software
DEI	Departamento de Engenharia Informática
ISEP	Instituto Politécnico de Engenharia do Porto
SOA	Arquitetura orientada para serviços
UVT	Unidade Virtual de Tributação
ERP	Enterprise Resource Planning, Sistema integrado de gestão empresarial
SOAP	Simple Object Access Protocol, protocol para troca de informações estruturadas numa plataforma descentralizada e distribuída
FTP	File Transfer Protocol, protocolo de transferência de arquivos entre computadores em redes locais ou na internet
M2K	Mappping to Know ou Methodology to Know
UI	Interface do utilizador
ISO	Organização Internacional de Normalização
OSI	Open System Interconnection, modelo de rede de computador referência da ISO dividido em camadas
DOM	Document Object Model, interface de programação para documentos HTML e XML. Representa os dados dos objetos que compõem a estrutura e o conteúdo de um documento na web
HTML	HyperText Markup Language, linguagem de marcação utilizada na construção de páginas web
XML	Extensible Markup Language, linguagem de marcação que fornece regras para definir quaisquer dados

CPU	Central Processing Unit, responsável pela realização das operações logicas, cálculos e processamento de dados
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
CDF	Métrica de medição da função de distribuição acumulada
DP	Métrica de medição da probabilidade de descarte
DR	Métrica de medição da taxa de rejeição
MRT	Métrica de medição do tempo médio de resposta
TP	Métrica de medição da taxa de transferência
TCP	Transmission Control Protocol, protocolo de comunicação
UDP	User Datagram Protocol, protocolo simples da camada de transporte
IP	Endereço IP, etiqueta numérica atribuída pelo seu fornecedor se serviços de internet
SSL	Transport Layer Security, protocol de segurança projetado para fornecer segurança nas comunicações sobre uma rede de computadores
AWS	Amazon Web Services, plataforma de serviços de computação em nuvem
DDD	Domain Driven Design, abordagem no design de software
MEAN	Conjunto de soluções JavaScript de software
NPM	Gerenciador de pacotes Node.js
API	Application Programming Interface, interface que permite a comunicação entre dois componentes de software
HTTP	Hypertext Transfer Protocol, protocolo de comunicação
AHP	Analytic Hierarchy Process, método utilizado na tomada de decisões complexas
RC	Razão de consistência
DNS	Domain Name System, sistema hierárquico e distribuído de gestão de nomes para computadores, serviços ou qualquer máquina conectada à internet

1. Introdução

Neste capítulo é descrito o enquadramento do projeto realizado em torno da aplicação SWiPE. Os principais objetivos deste capítulo são a descrição do problema em análise e as consequentes interpretações analítica, crítica e ética da resolução do mesmo. Além disso, será ainda abordada a metodologia utilizada nesta investigação, bem como uma breve descrição relativa à estrutura da dissertação.

1.1 Enquadramento/Contexto

O projeto descrito ao longo do relatório foi desenvolvido no âmbito da unidade curricular Preparação para Dissertação do Mestrado em Engenharia Informática - Engenharia de Software do Instituto Superior de Engenharia do Porto e, por isso, executado no Departamento de Engenharia Informática. Este trabalho visa realizar a reengenharia do software de forma a lidar com as variações nas cargas de utilização sem usar hardware caro e ineficiente por períodos de uso significativos.

1.2 Descrição do Problema

A aplicação SWiPE foi concebida tendo em visa a leitura de um ficheiro em CSV e o envio de emails correspondentes a cada linha do ficheiro.

Com o decorrer do tempo, a SWiPE foi incrementada com funcionalidades anteriormente executadas em Excel sobre os ficheiros CSV para uma única UC e respetiva edição. Atualmente, trata-se de uma aplicação com forte interação com o ser humano, uma vez que procura apoiar as funções de gestão de propostas e projetos curriculares.

As várias edições de UC em funcionamento implicam o registo de centenas de estudantes, docentes e colaboradores das organizações, bem como de centenas de propostas e posteriormente projetos, suas revisões e arguições.

1.2 Descrição do Problema

A carga de utilização do sistema varia muito ao longo dos períodos de funcionamento das várias edições. Se, por um lado, pode haver períodos de meses em que não existe praticamente qualquer utilização, por outro, podem existir períodos de poucos dias em que a carga de utilização é da ordem de milhares de pedidos REST por minuto. Este último cenário é caracterizado pela diminuição do desempenho e usabilidade do sistema.

Existem sistemas físicos (no DEI e ISEP) capazes de lidar com a carga máxima de utilização e manter um desempenho e usabilidade adequada. No entanto, e no atual contexto de negócio no qual a carga de utilização é muito variável, esses dispositivos físicos seriam subaproveitados e pouco justificados em muitos períodos de baixa carga de utilização.

O desempenho e a usabilidade do SWiPE são diretamente afetados por estas diferenças extremas no uso do sistema. A resposta do utilizador da aplicação em estudo pode ser comprometida durante os períodos de alta demanda, o que pode levar a uma experiência insatisfatória. Já durante os períodos prolongados de inatividade, os sistemas físicos do DEI e do ISEP ficam subaproveitados e economicamente ineficientes.

Os problemas identificados e, posteriormente, analisados, surgiram do desenvolvimento da aplicação em causa. A evolução do SWiPE contribuiu para a crescente complexidade da aplicação e afetou negativamente a elasticidade e adaptabilidade a cargas de uso variáveis do sistema. A expansão para suportar diferentes unidades curriculares com variados requisitos operacionais contribuiu para o aumento desta complexidade. Finalmente, e como a aplicação é utilizada num ambiente educacional, as variações de carga de utilização são justificadas pela existência de períodos de uso mínimo, nomeadamente o período de férias escolares, e períodos de atividade intensa, como, por exemplo, os períodos de revisões de propostas e discussões de projetos. Assim, surge a necessidade de um sistema mais flexível e adaptável.

Não existem estatísticas disponíveis referentes ao uso do SWiPE por parte dos utilizadores nem relatórios de impacto correspondentes. Todavia, a partir da utilização do sistema desenvolvido, é possível inferir que o carregamento da informação para o cliente web se trata de um processo demorado.

Relativamente às restrições do sistema, há algumas considerações a ter em conta associadas à disponibilidade de recursos financeiros para investir em soluções de infraestrutura, escalabilidade e otimização.

Considerando o problema descrito, foram identificados diferentes desafios a serem superados. Entre eles, a otimização da aplicação de modo a garantir um desempenho eficiente mesmo durante os picos de carga de utilização, com tempos de resposta aceitáveis e uma experiência de utilizador satisfatória.

Além disto, a aplicação deve ser capaz de se adaptar de forma dinâmica às variações na carga de utilização. Para que tal aconteça, a SWiPE deve ter condições para escalar os recursos disponíveis quando necessário e reduzir eficientemente os mesmos durante os períodos de baixa utilização.

Perante as necessidades expostas, é necessário identificar e implementar estratégias de elasticidade que permitam um desempenho consistente e uma adaptabilidade dinâmica na aplicação SWiPE, sem incorrer em custos desnecessários durante períodos de baixa demanda.

1.3 Interpretação Analítica, Crítica e Ética do Problema a Resolver

Sob uma perspetiva analítica, torna-se essencial especificar os fatores previamente mencionados com o objetivo de compreender as suas implicações individuais no problema em questão. A evolução da aplicação, aliada aos distintos requisitos operacionais das unidades curriculares, evidencia a importância de um sistema flexível e escalável. Os padrões variáveis da utilização do sistema sublinham a importância de soluções adaptativas capazes de alocar e desalocar recursos de forma eficiente e conforme a utilização.

Quanto à análise crítica de um problema, esta envolve a avaliação das práticas atuais, a identificação dos problemas e a contestação dos pressupostos subjacentes. A aplicação em estudo foi originalmente implementada com o objetivo de realizar tarefas específicas, como a leitura de ficheiros CSV e o envio de emails. Contudo, o sistema evoluiu de forma natural, sem antecipar as dificuldades que poderiam surgir posteriormente. Esta análise revela potenciais falhas no design inicial da aplicação, particularmente no que diz respeito ao planeamento de escalabilidade e à adaptabilidade face às constantes mudanças no setor educacional. Recomenda-se, portanto, uma reavaliação da arquitetura da aplicação, bem como a adoção de um modelo mais adaptável e eficiente no uso de recursos utilizados.

Do ponto de vista ético, a satisfação do utilizador e a gestão de recursos são os dois fatores mais importantes para esta perspetiva. A constante evolução do sistema e das operações educacionais proporcionadas contribui para o aumento da responsabilidade moral de fornecer um serviço eficiente e justo a todos os stakeholders.

Por outro lado, também é possível observar uma preocupação sobre o acesso equitativo aos recursos e o tratamento justo dos utilizadores. Isto deve-se à possibilidade de degradação do desempenho em períodos de alta carga de utilização.

A integração das perspetivas mencionadas oferece uma compreensão mais clara do problema visado. A partir destas, é percetível os interesses dos stakeholders, isto é, todos aqueles que utilizam a aplicação SWiPE, desde professores e alunos a instituições de ensino. A eficiência do sistema é crucial na reengenharia proposta, de modo a atender às necessidades de todos os grupos referidos.

Relativamente ao impacto da reengenharia do sistema referido, além da melhoria do desempenho da aplicação durante os períodos de maior carga de utilização do sistema, contribui para uma melhor qualidade de ensino. Consequentemente e considerando todos os

impactos atuais da aplicação SWiPE, é possível que ocorra uma degradação contínua do desempenho do sistema e a insatisfação dos utilizadores se este problema persistir sem solução.

Perante os impactos descritos da aplicação e as consequências que esta acarreta, estima-se que a resolução do problema culmine numa melhoria no que diz respeito à satisfação dos utilizadores, bem como à qualidade dos serviços proporcionados pelo SWiPE.

1.4 Objetivo(s), processo de investigação, método(s) e pergunta(s) claros, bem definidos e exequíveis

O principal objetivo deste projeto é proceder à reengenharia do software da aplicação SWiPE, com o intuito de permitir que a mesma lide, de forma eficiente, com variações nas cargas de utilização. Pretende-se, assim, evitar a necessidade de recorrer a hardware dispendioso e frequentemente subaproveitado em períodos significativos de utilização.

Deste modo, é possível identificar diversos objetivos com a finalidade de concretizar o principal objetivo, previamente mencionado. Estes objetivos podem ser delineados com base nos problemas identificados na aplicação SWiPE. Assim, neste estudo, almeja-se a melhoria do desempenho, a adaptabilidade dinâmica e a eficiência operacional do sistema visado.

A melhoria do desempenho da aplicação torna-se um objetivo essencial devido à contínua degradação do desempenho do sistema, principalmente durante os picos de carga de utilização, de forma a garantir tempos de resposta rápidos e consistentes nesses períodos. Isto traduz-se na necessidade de reduzir os tempos de resposta e evitar atrasos nos momentos de alta utilização, para que todos os estudantes, docentes e colaborados possam usar o sistema de uma forma natural e sem interrupções.

A utilização da aplicação SWiPE varia significativamente ao longo do tempo. Assim, a adaptabilidade dinâmica é um dos principais objetivos deste projeto, uma vez que visa transformar o sistema atual num que seja capaz de ajustar automaticamente os seus recursos aos períodos e demanda da aplicação. Desta forma, torna-se possível minimizar desperdícios em períodos de baixa utilização, bem como escalar recursos necessários para manter o desempenho do sistema nos momentos de maior utilização. Com este objetivo, pretende-se proporcionar à aplicação em estudo, uma operação eficiente e económica, garantindo que o sistema se ajuste automaticamente às variações de utilização verificadas.

Com o propósito de oferecer um serviço confiável e de qualidade sem recorrer a custos desnecessários, é importante promover a eficiência operacional no sistema atual. Neste sentido, este objetivo visa a otimização dos recursos, de forma que estes sejam utilizados apenas quando necessário, evitando o uso de hardware dispendioso para lidar com os picos de utilização durante todos os períodos de uso da aplicação SWiPE. Assim, torna-se viável a redução dos custos operacionais, assegurando que o sistema funcione de uma maneira sustentável e económica.

Inicialmente, é essencial realizar uma revisão detalhada de literatura sobre estratégias eficazes de reengenharia de software, capazes de lidar com variações nas cargas de utilização. De seguida, deve-se avaliar o desempenho atual da aplicação SWiPE e estabelecer os requisitos de desempenho e usabilidade que o sistema deve considerar nos cenários de baixa e alta utilização.

Considerando o exposto, é importante propor e desenvolver a estratégia de reengenharia mais adequada ao problema deste projeto, considerando as estratégias consultadas no primeiro objetivo proposto, com o propósito de melhorar a eficiência do sistema em diferentes níveis de carga de utilização. A escolha da estratégia deve ser realizada baseada numa avaliação de viabilidade económica.

Finalmente, deve ser realizada uma avaliação do impacto da estratégia implementada na aplicação SWiPE, tendo em conta as métricas previamente estabelecidas de desempenho e usabilidade.

1.5 Metodologia

Para garantir a qualidade dos documentos utilizados na revisão de literatura, será adotada a metodologia de pesquisa *Preferred Reporting Items for Systematic Reviews and Meta-Analyses* (PRISMA) (Page *et al.*, 2021). Publicada em 2019, esta metodologia reflete os avanços nas etapas de procura, identificação, seleção, análise, avaliação e síntese de estudos essenciais para a melhoria da transparência e qualidade de uma revisão sistemática.

Primeiramente, e tendo em conta os objetivos do projeto, foram definidas quatro questões com a finalidade de delinear o ponto crucial de pesquisa. De seguida, foram escolhidas diferentes bibliotecas e definidas palavras-chave para cada questão. Posteriormente, estas palavras-chave foram utilizadas como termos de pesquisa dos documentos. Finalmente, e com o propósito de obter os documentos mais apropriados para este projeto, foram estabelecidos critérios de exclusão e de inclusão.

1.5.1 Questões de Pesquisa

A evolução de uma aplicação simples para uma que combina as funcionalidades associadas à gestão de propostas e projetos curriculares em diversas unidades curriculares resultou num aumento significativo da complexidade do sistema.

Face a esta complexidade, surgiram questões essenciais relacionadas com a elasticidade e adaptabilidade da aplicação SWiPE. O objetivo deste documento é abordar estas questões e proporcionar soluções eficientes para garantir um desempenho consistente e uma adaptabilidade dinâmica na aplicação visada, sem incorrer em custos desnecessários durante os períodos de baixa utilização.

De acordo com a metodologia PRISMA, após a definição do principal objetivo, foram formuladas quatro questões relativas ao tema, com o propósito de apoiar a concretização do mesmo.

Na Tabela 1 estão identificadas as questões utilizadas, e destacados os pontos que direcionaram a procura por soluções eficazes para o problema em estudo.

Tabela 1: Questões de Pesquisa

	Questões de Pesquisa
QP1	Quais estratégias de reengenharia de software têm sido propostas na literatura?
QP2	Quais são os requisitos críticos para o desempenho e usabilidade em aplicações com diferentes cargas de utilização?
QP3	Quais são as estratégias de otimização de software eficazes na melhoria da eficiência e da adaptabilidade de aplicações em cenários variáveis de carga?
QP4	Quais são os contributos da MEAN *stack* na melhoria da elasticidade e escalabilidade de uma aplicação?

1.5.2 Bibliotecas de Documentos

Com as questões de pesquisa definidas, e de modo a conduzir uma revisão de literatura, é necessário identificar as bibliotecas de documentos a serem utilizadas.

A seleção das bibliotecas desempenha um papel fundamental na condução de uma pesquisa eficaz, de modo a garantir a análise de referências pertinentes ao estado de arte. Com o intuito de obter um resultado mais fidedigno, foram usadas duas bibliotecas. Este aspeto permitiu uma revisão abrangente e aprofundada da literatura relevante para o tópico em questão. Em ambas, foram aplicados os mesmos operadores e palavras-chave nas pesquisas realizadas.

Na Tabela 2 estão identificadas as bibliotecas utilizadas como meio de procura de documentos.

Tabela 2: Bibliotecas de Documentos

Identificador	Biblioteca	*Url*
WoS	Web of Science	https://www.webofscience.com/
ACM	ACM Digital	https://dl.acm.org/

A escolha destas bibliotecas deve-se ao facto de apresentarem um maior número de resultados pertinentes para o objeto de estudo deste projeto, bem como à facilidade na combinação de todos os domínios estudados.

1.5.3 Domínios e palavras-chave

A definição clara dos domínios e das palavras-chave desempenha um papel crucial na orientação e no refinamento da pesquisa.

Esta escolha visa delinear os parâmetros específicos relacionados à elasticidade e adaptabilidade de aplicações em ambientes de carga variável, proporcionando uma procura focada e abrangente.

Na Tabela 3 estão representados os domínios e palavras-chave aplicados ao longo da pesquisa de documentos.

Tabela 3: Domínios e palavras-chave

Identificador	Domínio	Palavras-chave
D1	Reengenharia de Software	(“software reengineering” AND (“strategies” OR “techniques” OR ”methods” OR “definition”))
D2	Elasticidade e aplicações com diferentes cargas de utilização	(“elasticity” OR “dynamic workload” OR “workload changes”)
D3	Requisitos críticos relativos ao desempenho e usabilidade	("performance metrics" OR "performance requirements" OR "usability metrics" OR "usability requirements")
D4	Estratégias de otimização de software	("optimization" AND "software" AND ("techniques" OR "methods" OR "strategies"))
D5	MEAN *stack*	“MEAN *stack*”

A partir da utilização destas palavras-chave como termos de pesquisa, foram realizadas diferentes combinações para refinar os resultados. Este processo culminou na obtenção de uma *query* que representasse todos os documentos relevantes, capazes de dar resposta às perguntas efetuadas anteriormente.

Durante a pesquisa foram ainda documentadas todas as combinações que não produziram os resultados esperados.

Na Tabela 4 encontram-se representadas todas as combinações mencionadas.

Tabela 4: Combinação dos domínios e os respetivos resultados

Domínios	Resultados		
	WoS	ACM	Total
D1	10	68	78
D2	221	215	436
D3	275	668	943
D4	194	1.659	1.853
D5	16	32	48
D2 AND D3	49	64	113
D2 OR D3	495	819	1.314
D2 AND D4	35	125	160
D2 OR D4	414	1.749	2.163
(D2 AND D3) AND (D2 AND D4)	0	57	57
(D2 AND D3) OR (D2 AND D4)	84	132	216
D1 AND (D2 AND D3) OR (D2 AND D4)	0	1	1
D1 OR (D2 AND D3) OR (D2 AND D4)	94	335	429
(D1 OR (D2 AND D3) OR (D2 AND D4)) AND D5	0	0	0
(D1 OR (D2 AND D3) OR (D2 AND D4)) OR D5	103	349	452

Com base na análise da tabela anterior, e com a finalidade de obter os documentos utilizados como referência no estado de arte, foi utilizada a *query* identificada na Tabela 5.

Tabela 5: *Query* Final

Query Final
(("software reengineering" AND ("strategies" OR "methods" OR "definition" OR "techniques")) OR (((“elasticity” OR “dynamic workload” OR “workload changes”) AND (“optimization” AND “software” AND ("techniques" OR "methods" OR "strategies"))) OR ((“elasticity” OR “dynamic workload” OR “workload changes”) AND (“performance metrics” OR “performance requirements” OR "usability metrics" OR "usability requirements")))) OR (“MEAN stack”)

1.5.4 Avaliação da Qualidade dos Resultados: Critérios de Inclusão e Exclusão

De forma a utilizar a informação que mais se adequa ao problema descrito neste documento foram aplicados alguns critérios de inclusão e de exclusão.

A aplicação destes critérios foi extremamente importante não apenas para evitar a perda de tempo na leitura de todos os documentos, mas também para garantir a obtenção de uma lista de documentos mais consistentes e focados nas questões representadas na Tabela 1.

Na Tabela 6 estão representados os critérios de inclusão aplicados.

Tabela 6: Critérios de Inclusão

	Critérios de Inclusão
CI1	O documento pertence ao domínio da engenharia informática
CI2	O documento contribui significativamente para os domínios de estudo

Na Tabela 7 estão representados os critérios de exclusão aplicados.

Tabela 7: Critérios de Exclusão

	Critérios de Exclusão
CE1	O documento encontra-se fora do espaço temporal 2017-2023
CE2	O documento não está escrito em inglês
CE3	O documento não é um artigo

1.5.5 Extração dos Documentos

Na Figura 1 encontra-se representado o processo de extração de documentos relevantes para responder às questões efetuadas.

Para encontrar entradas duplicadas de documentos foi utilizado o JabRef's. Com a aplicação desta ferramenta foi possível encontrar 1 documento duplicado, o que facilitou a sua remoção.

Após esta etapa, foram considerados 452 documentos. No entanto, após a leitura dos resumos dos mesmos, apenas 124 documentos foram identificados como relevantes. De seguida, efetuou-se a leitura integral de todos os documentos, confirmando que todos se encontravam dentro do domínio selecionado. Desta forma, e tendo em conta os documentos selecionados,

procedeu-se à avaliação dos critérios de qualidade, o que resultou na remoção de 28 documentos do estudo.

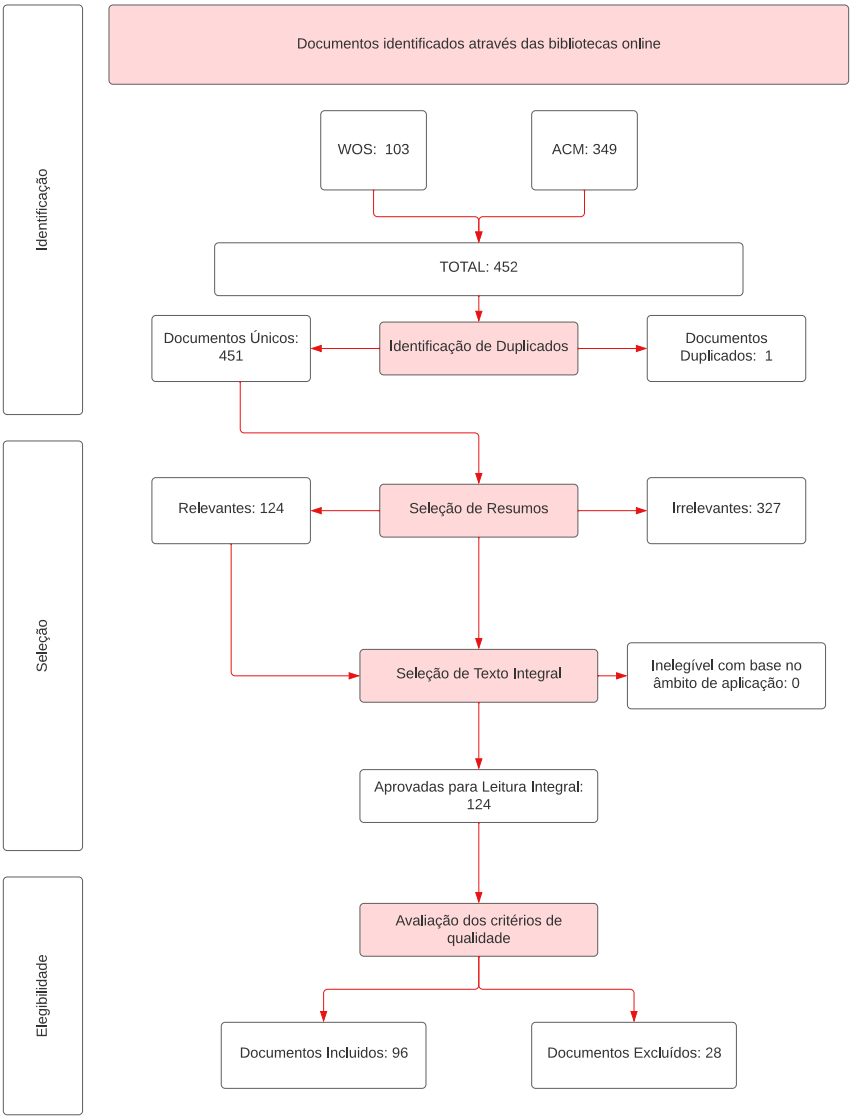

Figura 1: Extração de Documentos

1.6 Estrutura da Dissertação

Esta dissertação encontra-se organizada em seis capítulos.

No primeiro capítulo, é descrito o trabalho desenvolvido, os objetivos traçados, a interpretação analítica, crítica e ética do problema a resolver e a metodologia de pesquisa.

No segundo capítulo, são apresentados e analisados trabalhos relacionados com este projeto, que poderão contribuir para a elaboração do mesmo.

No terceiro capítulo é realizada uma análise da aplicação SWiPE. Desta forma, é descrita a arquitetura da aplicação com base no modelo C4+1, são descritos os principais processos do sistema, os desafios atuais e análise das necessidades de escalabilidade do SWiPE. Por fim, também são definidos objetivos de desempenho e elasticidade.

O quarto capítulo refere-se ao desenho da nova solução. Neste capítulo é definida a estratégia de otimização a ser implementada no sistema SWiPE e seguidamente é realizado o desenho da mesma.

No quinto capítulo é descrito o processo de implementação da nova solução. Para além disso, são expostos os resultados obtidos a partir dos testes realizados sob a implementação efetuada e posteriormente, os mesmos são analisados.

No último capítulo, é realizado um resumo de todos os objetivos alcançados, as limitações do projeto desenvolvido e uma reflexão sobre as melhorias que podem ser efetuadas futuramente.

2. Estado de Arte

Este capítulo descreve o estado atual referente à temática deste projeto. Para tal, foi efetuada uma revisão literária baseada nas quatro perguntas mencionadas na Tabela 1. O objetivo é garantir que, no momento da conceção da solução, seja possível ponderar e escolher as abordagens mais relevantes para pôr em prática.

2.1 QP1 - Quais estratégias de reengenharia de software têm sido propostas na literatura?

Normalmente, o tema da reengenharia de software é associado com sistemas legados (*legacy)*. Estes sistemas foram desenvolvidos antes do surgimento de métodos de engenharia de software mais modernos e constituem geralmente, uma parte essencial de um sistema específico. Com o passar do tempo, estes sistemas tornaram-se mais complexos, o que contribuiu para a dificuldade de manutenção e para a diminuição da sua importância operacional.

Assim, estes sistemas mais antigos representam desafios para as organizações que dependem deles. Para além disso, persiste ainda a dúvida sobre a capacidade funcional contínua dos sistemas legados.

A reengenharia de software surge como uma abordagem de resolução de problemas associados a sistemas de software mais antigos, geralmente chamados de legados (Hussain, Bhatti and Rasool, 2017). Este processo oferece uma variedade de estratégias, dependendo do estado técnico do sistema, da importância do sistema para o negócio e da estrutura organizacional da operação. Ou seja, a escolha de uma estratégia adequada ao problema visado é extremamente difícil, uma vez que exige uma análise completa do sistema e dos recursos disponíveis.

2.1.1 Estratégias de reengenharia de software

Entre as estratégias mencionadas estão a remoção total do sistema, a revisão de diretrizes de negócio ou a continuidade de esforços através de uma manutenção regular do sistema de software. Além destas estratégias, são ainda consideradas a reescrita seletiva de componentes do sistema, bem como a reconstrução completa do sistema com uma arquitetura totalmente nova.

A criação de um novo sistema pode ser considerada como a estratégia mais viável quando o sistema apresenta problemas críticos que impedem a atualização ou manutenção do mesmo. Deste modo, quando os custos de manutenção superam os custos da construção de um novo sistema ou quando o mesmo não é flexível a aceitar modificações, esta estratégia deve ser utilizada.

Já a reutilização de software existente é um processo de desenvolvimento de software que envolve a utilização de uma ou mais partes de um software já existente. Trata-se de um processo relativamente simples, sendo que consiste em utilizar componentes já existentes e necessários de outros sistemas, como pedaços de código, design, entre outros. É uma estratégia a adotar quando uma organização pretende economizar dinheiro, reduzindo significativamente o número de falhas de sistema, e tem o objetivo de alcançar a eficiência e eficácia de um sistema.

Por último, no que diz respeito à reengenharia de software, particularmente à reescrita seletiva de componentes do sistema, é possível afirmar que se trata das estratégias mais comuns para atualizar sistemas legados. Esta abordagem é capaz de produzir um produto aprimorado com as funcionalidades desejadas do sistema e dedica-se à construção de novos componentes ou ao desenvolvimento de um sistema totalmente novo baseado em engenharia avançada.

Esta estratégia pode ser definida em três fases: engenharia inversa, reestruturação e engenharia avançada.

O método de engenharia inversa, conforme definido por Chikofsky e Cross (Jovanovikj *et al.*, 2019), é uma abordagem crucial no contexto da reengenharia de software. Foca-se principalmente numa análise minuciosa do sistema com o objetivo de criar representações do mesmo numa forma diferente ou num nível de maior abstração.

Nesta abordagem é realizada uma análise de todas as especificações do sistema e são identificadas as que devem ser mantidas. Este processo pode ser considerado desafiante, tendo em conta que as especificações de um sistema estão constantemente a ser alteradas, devido ao estado técnico do mesmo ou regras de negócio.

Desta forma, aquando da análise das especificações do sistema, é possível obter uma compreensão clara de todas as funcionalidades e dados de um produto. Este aspeto promove um entendimento mais preciso relativamente ao que está a acontecer e o que deve ser melhorado.

Relativamente ao processo mencionado anteriormente, e embora esta abordagem seja mais complexa, é significativamente menos arriscada e de menor custo.

De acordo com (Jovanovikj *et al.*, 2019), a reestruturação é definida como a transformação de uma forma de representação para outra no mesmo nível relativo de abstração, preservando o comportamento externo do sistema.

Um dos seus principais objetivos sugere a clarificação de alguns aspetos do sistema relativos à qualidade ou estrutura, de modo a alcançar as especificações especificas de negócio.

Por último, conforme (Jovanovikj *et al.*, 2019), a engenharia avançada é o processo tradicional de transformar abstrações de alto nível e designs lógicos independentes de implementação, numa implementação física de um sistema.

Na abordagem de engenharia avançada, são utilizadas algumas técnicas responsáveis pela extração automática de código. Contrariamente ao processo de engenharia inversa, esta abordagem começa pelos requisitos e culmina na avaliação do comportamento e desempenho do sistema.

Na Figura 2 estão representadas as diferentes técnicas utilizadas durante o processo de reengenharia de software, de forma a facilitar a identificação visual clara do que acontece em cada uma das abordagens.

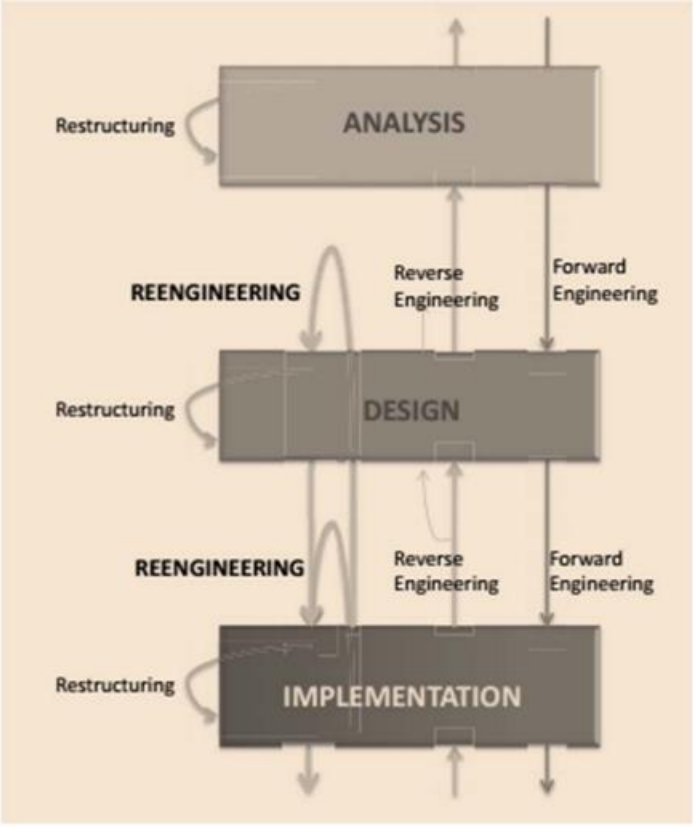

Figura 2 - Processo de Reengenharia de Software (Hussain, Bhatti and Rasool, 2017)

Embora o sistema atual não seja considerado um sistema legado (*legacy*), os desafios enfrentados são similares aos desafios do problema visado neste documento, especialmente no que diz respeito à adaptabilidade e eficiência operacional. A implementação de estratégias que maximizem a eficiência do sistema e que garantam a adaptabilidade contínua do mesmo é essencial, visto que a carga de trabalho sofre alterações significativas, seja por mudanças sazonais, seja por outros fatores.

Assim como no caso de sistemas legados referido, é possível aplicar a reengenharia de software como uma abordagem valiosa com o objetivo de melhorar o sistema existente. Assim, é fundamental realizar uma análise completa ao sistema atual e avaliar todos os recursos que este tem à sua disposição aquando da escolha de estratégia de reengenharia.

2.1.2 Exemplos de reengenharia de software

Um exemplo de reengenharia de software encontrado foi o processo aplicado ao sistema Unidade Virtual de Tributação (Da Silva, Yan De Lima Justino and Adachi, 2022), também conhecido por UVT, com o objetivo de lidar com os problemas associados ao sistema mais antigo.

A modularização inadequada das entidades e os estrangulamentos associados ao desempenho do sistema relativos à comunicação síncrona foram alguns dos problemas descobertos no sistema UVT que motivaram a implementação da reengenharia mencionada.

Para lidar com estes obstáculos, foi escolhido o processo *Service-oriented Process for Reengineering and DevOps* (SPReaD). Para além desta decisão, foi implementada a arquitetura orientada para serviços (SOA) com foco no desenvolvimento de microsserviços, de forma a aumentar a escalabilidade, a modularidade do sistema, bem como a sua autonomia.

O antigo sistema UVT dificultava a escalabilidade e a modularidade do mesmo devido à existência de um forte acoplamento entre a camada de lógica de negócio e a camada de apresentação. O acoplamento forte implica que o código responsável por "renderizar" a interface do utilizador e de guardar as respostas do mesmo está diretamente associado à lógica de negócios subjacente. Assim, quaisquer alterações na lógica de negócio podem exigir modificações na camada de apresentação e vice-versa.

A utilização de microsserviços estabeleceu limites para cada serviço e permitiu a existência de um acoplamento mais flexível. Desta forma, a alteração da arquitetura monolítica promoveu a melhoria da adaptabilidade e capacidade de resposta à evolução dos requisitos.

A adoção do processo SPReaD e a criação de microsserviços também promoveram a resolução do problema associado à falta de modularização das entidades de negócio. Assim, o SPReaD permitiu a instalação de uma arquitetura modular e de fácil gestão.

Relativamente aos processos de reengenharia de software, a fase de engenharia inversa foi utilizada nomeadamente associada à análise atual do sistema, incluindo a identificação dos

componentes existentes, a compreensão da lógica de negócios implementada e o mapeamento da arquitetura atual.

Também foi incluída neste processo a fase de engenharia avançada. Nesta fase, foram utilizadas as descobertas da fase descrita anteriormente de modo a redesenhar, recriar e reimplementar o sistema. Um exemplo desta fase é a definição de uma nova arquitetura do sistema, isto é, a implementação de microsserviços com base nos componentes identificados e a reformulação da lógica de negócios.

As fases referidas podem ser repetidas, permitindo a melhoria contínua e a adaptação do sistema às mudanças nos requisitos.

No projeto referido em (Mulcahy and Huang, 2017), o foco é a resolução dos problemas enfrentados pelas partes interessadas dos sistemas "*Enterprise Resource Planning*", ERP. À medida que a tecnologia evolui, os ERP são considerados sistemas cada vez mais antigos e torna-se, consequentemente, mais dispendioso manter sistemas similares a estes, adaptar o sistema às novas tecnologias ou integrar soluções de terceiros.

Neste projeto, são referidos vários problemas associados ao processo de reengenharia de software pretendido. A substituição não é economicamente viável, devido à unicidade dos sistemas ERP. Outro problema referido é ainda o facto de muitos ERP serem sistemas legados, o que significa que são sistemas que necessitam de ser adaptados às novas tecnologias e às soluções externas.

Para além destes, a modificação de sistemas aumenta, de forma geral, a complexidade dos mesmos, o que pode resultar em custos economicamente mais elevados em termos de engenharia de software para sistemas legados. A necessidade de o sistema possuir componentes autónomos com a capacidade de resolver a complexidade do mesmo integram os problemas identificados no projeto estudado.

A análise do sistema ERP antigo, a compreensão de todos os seus componentes e a identificação de gestores de interação autónoma (AIMs) integram a fase de engenharia inversa do processo de reengenharia de software.

O documento analisado refere os problemas enfrentados pelas partes interessadas ao lidar com a manutenção do sistema e menciona que a decisão de iniciar o processo de reengenharia de software surgiu com a necessidade de aumentar a eficiência e autonomia do sistema.

Tal como referido, a engenharia avançada é responsável pela transformação de conceitos lógicos e abstrações de alto nível para a implementação física do sistema. Neste documento, menciona que a engenharia avançada foi utilizada de forma a gerar casos de teste a partir de modelos de teste abstratos. Este processo envolveu transformações de modelo em modelo, bem como uma transformação de modelo para texto, de forma a gerar código de teste.

Contrariamente ao caso exposto anteriormente, o projeto relativo ao sistema ERP não menciona alterações na arquitetura do sistema associadas à reengenharia de software. Neste

projeto, derivado do processo de reengenharia, foram realizadas uma migração de linguagem de COBOL para C++ e uma alteração no sistema operativo e protocolo de comunicação. A nível do sistema operativo, a adaptação a um novo ambiente envolveu a alteração do minicomputador HP3000, a correr no sistema operativo MPE/iX para um sistema operativo Windows. A nível de protocolo de comunicação, ocorreu a mudança de SOAP para FTP.

Também é possível observar o caso da metodologia M2K (*Mappping to Know ou Methodology to Know*) e as refatorações associadas ao contexto da reengenharia de software (Cassol and Arévalo, 2018).

Esta metodologia foi projetada para a reengenharia de aplicações legados programadas em C. O principal objetivo da M2K é realizar a análise do código C atual e criar um modelo de classe refatorizado.

Esta abordagem é composta por duas etapas principais: a examinação do código e o mapeamento. Na primeira etapa, é utilizada uma ferramenta responsável pela extração de diversas entidades C, tais como módulos, variáveis, tipos de dados abstratos e funções de código.

Quanto à reengenharia de software, é possível inferir que a engenharia inversa foi utilizada nesta etapa, dado que esta tem como principal objetivo compreender e extrair informação acerca do sistema existente. Já na etapa associada ao mapeamento, um especialista realiza uma intervenção manual para identificar o conjunto final de classes que serão representadas no código legado. Novamente, é possível identificar o uso de engenharia avançada, sendo que o especialista envolvido nesta etapa utiliza a informação gerada para orientar a transformação das entidades extraídas num modelo de classes refatorizado.

Em conclusão, a transição da fase de engenharia inversa para a de engenharia avançada é caracterizada pela participação de especialistas que tomam decisões baseadas na compreensão dos objetivos de implementação e das funcionalidades do sistema. Os princípios da engenharia inversa são utilizados como um guião para a extração inicial de informação detalhada acerca do código atual enquanto as etapas subsequentes são responsáveis pela criação de um design mais sofisticado e refinado.

2.2 QP2 - Quais são os requisitos críticos para o desempenho e usabilidade em aplicações com diferentes cargas de utilização?

No que diz respeito à segunda questão definida, a avaliação do desempenho e usabilidade é essencial no desenvolvimento de sistemas eficientes e eficazes em ambientes dinâmicos, onde a carga de trabalho pode variar significativamente. Esta avaliação não é adotada apenas para garantir uma experiência otimizada para o utilizador, mas também para contribuir para a identificação de diferentes problemas e áreas de melhoria de um sistema.

Assim, é crucial realizar avaliações sob diferentes cenários para garantir que o sistema é robusto e capaz de lidar com as mudanças relativas à carga de utilização. Para atingir este objetivo, é utilizada uma variedade de métricas capazes de proporcionar uma compreensão completa do sistema.

2.2.1 Métricas relativas à usabilidade

O estudo da interface do utilizador é fundamental para a analisar e estabelecer métricas relativas à usabilidade de um sistema de software.

A interface do utilizador (UI) é vital uma vez que é o meio utilizado para a interação do mesmo com o sistema. Em (Miraz, Ali and Excell, 2021) refere que a simplicidade deve ser preservada, tanto na implementação de código desta interface como na sua utilização, mesmo com os avanços tecnológicos.

Para que a conceção da interface mencionada seja um sucesso, o modo de interação de um utilizador deve ser tido em conta. Por outras palavras, deve ser analisada a forma como os utilizadores interagem com a interface, uma vez que este desempenha um papel crucial para a sua eficácia. Além disso, é essencial rever a intenção e motivação do utilizador. Deve-se ter uma compreensão clara das razoes pelas quais os utilizadores usam a interface e o que os motiva a criar uma interface útil e funcional.

Adicionalmente, a usabilidade da interface do utilizador depende da facilidade com que os mesmos a utilizam. O contexto cultural também deve ser considerado, uma vez que diferentes culturas podem influenciar as expectativas e comportamentos durante a criação da UI. Por fim, a seleção da tecnologia para suportar a interface também pode impactar significativamente a forma como a mesma é concebida e usada pelo utilizador.

Considerando os fatores que influenciam o design de sucesso de uma interface do utilizador, Muhlhauser e Gurevych referem a existência de duas abordagens para a criação de uma UI.

Por um lado, a primeira abordagem referida permite que haja uma adaptação inteligente. Ou seja, este método visa ajustar a interface de forma a responder às necessidades do utilizador.

Por outro lado, a segunda abordagem é realizada através da criação de interfaces personalizadas, isto é, a criação de interfaces únicas com o propósito de atingir metas específicas. Embora este método possa oferecer algumas vantagens na personalização e adaptação às necessidades dos utilizadores, também proporciona o aumento da complexidade. Este aspeto poderá, posteriormente, representar alguns desafios a certos grupos de utilizadores, como os idosos, por exemplo.

O requisito de usabilidade não apresenta uma estratégia padrão para o sucesso e é definido pela Organização Internacional de Normalização (ISO) como a medida em que o produto pode

ser utilizado por utilizadores finais específicos com o fim de alcançar objetivos específicos com eficácia, eficiência e satisfação num contexto específico.

Os componentes críticos para a usabilidade de um sistema incluem a segurança e proteção, sendo que é essencial que a interface criada não seja prejudicial e ajude na redução de erros do utilizador. Estes componentes incluem ainda a eficácia, visto que a UI deve ser implementada de forma a ajudar nas tarefas do utilizador, e a eficiência das funcionalidades, de forma a agilizar todas as tarefas do utilizador.

Conforme mencionado anteriormente, a praticabilidade da interface deve ser um componente tido na definição de requisitos de usabilidade de um sistema. Isto, uma vez que a utilização da interface do utilizador não deve ser complexa e deve proporcionar uma experiência agradável ao utilizador. Deve ainda ser considerada a facilidade de aprendizagem, na medida em que esta curva de aprendizagem não deve ser acentuada e os utilizadores devem ter facilidade em recordar todos os procedimentos necessários à realização das funcionalidades promovidas pela interface.

Para garantir que a interface da aplicação atenda às expectativas dos utilizadores, isto é, de forma a fornecer uma experiência de utilizador positiva e eficaz, são necessárias estas métricas de usabilidade. A avaliação contínua destas métricas permite que a interface seja adaptada às demandas dos utilizadores.

2.2.2 Métricas relativas ao desempenho

Alguns exemplos de métricas relativas ao desempenho do sistema foram definidas em (da Silva Pinheiro *et al.*, 2023) e (Lehrig, Eikerling and Becker, 2017) tais como CDF, DP, DR, MRT e TP.

A métrica CDF, também conhecida por função de distribuição cumulativa, é responsável pela representação da probabilidade cumulativa de uma variável aleatória ser menor ou igual a um determinado valor. Relativamente ao desempenho, esta métrica pode ser utilizada de forma a compreender como os tempos de resposta se distribuem, identificando os percentis críticos.

A métrica DP, ou probabilidade de descarte, é essencial em sistemas que enfrentam desafios relacionados com a gestão da sobrecarga. Esta métrica indica a probabilidade de um pedido ser descartado, permitindo que sejam tomadas decisões informadas sobre como lidar com os picos de carga de utilização.

A métrica DR, também denominada de taxa de rejeição expressa a eficiência do sistema ao lidar com os pedidos durante os períodos de carga de utilização intensa. Uma taxa de rejeição alta pode indicar possíveis problemas de capacidade ou a necessidade de estratégias de rejeição mais sofisticadas.

A métrica MRT, igualmente chamada de tempo médio de resposta fornece uma medida agregada do desempenho do sistema, representando o tempo médio que o sistema demora na

resposta a um pedido. Variações no tempo médio de resposta podem indicar mudanças no desempenho do sistema, o que pode exigir uma investigação detalhada.

Finalmente, a métrica TP, frequentemente denominada por taxa de transferência, destaca a capacidade de o sistema realizar pedidos num determinado período. A monitorização desta métrica é crucial para garantir que a aplicação consiga responder às demandas dos utilizadores, especialmente em situações de carga variável.

Em suma, as organizações podem analisar e monitorizar estas métricas de forma contínua para ajustar dinamicamente os seus recursos, otimizar o desempenho dos seus sistemas e garantir uma experiência satisfatória de utilizador, mesmo em situações de carga de utilização variável.

A ferramenta Apache JMeter é utilizada para a realização de testes de desempenho. Tal como referido em (Al-Said Ahmad and Andras, 2019), esta ferramenta foi utilizada com o propósito de simular a carga de trabalho de diferentes sistemas. Neste documento, o Apache JMeter foi utilizado juntamente com scripts de teste, de forma a abordar as variadas métricas mencionadas e efetuar a comparação entre dois serviços.

Com a evolução da computação em nuvem e a adoção de arquiteturas de microsserviços e de tecnologias *serveless*, surgem novos desafios na avaliação do desempenho (Herbst *et al.*, 2018).

O aumento da adoção de sistemas e tecnologias de contentores como o Docker, proporciona a utilização de contentores implantados em máquinas virtuais por parte das aplicações. De forma a incluir aspetos de qualidade em várias camadas virtuais, é necessário ajustar as métricas para lidar com o aumento desta complexidade.

Podem ser utilizadas métricas que avaliem a latência em diferentes camadas, como a execução de funções *serveless* e operações em container. Para analisar a distribuição equitativa de recursos, isto é, de forma a avaliar como estão a ser distribuídos os recursos entre diferentes microsserviços ou funções *serveless* e garantir uma utilização equitativa, pode ser utilizada uma métrica dedicada à variação percentual desta distribuição durante um período de carga de utilização variável.

2.3 QP3 - Quais são as estratégias de otimização de software eficazes na melhoria da eficiência e da adaptabilidade de aplicações em cenários variáveis de carga?

A eficiência e a adaptabilidade das aplicações tornam-se fundamentais em sistemas onde as dinâmicas dos ambientes de execução são tão variáveis e imprevisíveis. Assim, os sistemas de software devem ser ajustados dinamicamente, de forma a otimizar a utilização dos recursos disponíveis e garantir uma experiência eficiente e eficaz por parte do utilizador em todos os momentos de carga de trabalho.

Assim, é essencial a análise de diferentes técnicas e estratégias para aumentar a eficácia e adaptabilidade contínua das aplicações e, por consequência, otimizar o software do sistema.

2.3.1 Elasticidade horizontal vs Elasticidade vertical

A capacidade de dimensionar os recursos disponíveis de forma adaptativa e atempada, permitindo a resposta de pedidos em sistemas de carga de trabalho variáveis, é conhecida como elasticidade (Al-Dhuraibi *et al.*, 2017). Existem dois tipos de elasticidade: horizontal e vertical. Cada um possui características distintas que influenciam a escolha da sua utilização em diferentes cenários.

A elasticidade horizontal implica a replicação ou decomposição em instâncias independentes. Esta abordagem não é viável em sistemas onde a criação de várias instâncias do mesmo componente é impraticável e não funciona bem com aplicações MATLAB e AutoCAD. A elasticidade horizontal, permite a adição ou remoção de instâncias de recursos, proporcionando uma escalabilidade granular. No entanto, esta estratégia pode causar uma sobrecarga no sistema durante o processo de arranque das instâncias.

A elasticidade vertical refere-se ao refinamento minucioso da capacidade numa única instância. Este método oferece mais flexibilidade e independência na replicabilidade do sistema. É responsável também pela eliminação de sobrecarga no arranque de instâncias e é aplicável em qualquer aplicação. Ao contrário da elasticidade horizontal, esta abordagem permite adicionar ou remover unidades reais de recursos, não é necessária a adição de máquinas adicionais como *load balancers* ou instâncias replicadas.

Tendo em conta estas definições, a elasticidade vertical é mais adaptável a qualquer aplicação enquanto a elasticidade horizontal depende da replicabilidade da aplicação. Relativamente a custos, a elasticidade horizontal como exige licenças para as réplicas utilizadas poderá ser menos económica e em relação ao desempenho, a elasticidade horizontal geralmente apresenta uma performance melhor.

Para uma melhor perceção dos processos mencionados, a Figura 3 apresenta um cenário no qual a elasticidade horizontal é aplicada e a Figura 4 por sua vez, apresenta um cenário onde a elasticidade vertical pode ser verificada.

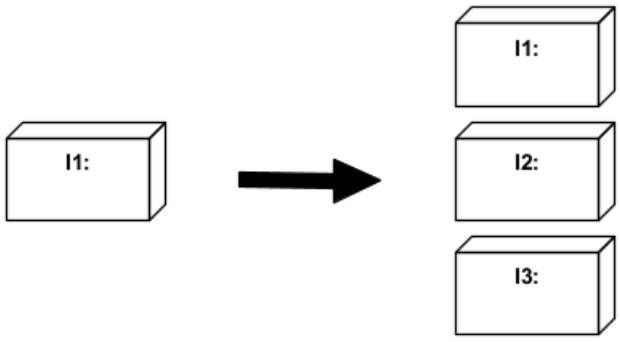

Figura 3: Elasticidade Horizontal

Figura 4: Elasticidade Vertical

2.3.2 *Load balancing*

Uma das técnicas consideradas para a otimização de sistemas com variações nas cargas de utilização é o balanceamento de carga dinâmico, também conhecido por *load balacing* (Mishra, Sahoo and Parida, 2020). É uma técnica responsável pela distribuição equitativa de tráfego entre os diversos recursos disponíveis de um sistema tais como contentores, servidores e máquinas virtuais. O objetivo desta técnica é maximizar o desempenho, melhorar a utilização dos recursos e garantir uma resposta eficiente aos pedidos.

Na Figura 5 é possível visualizar o comportamento desta técnica numa vista física.

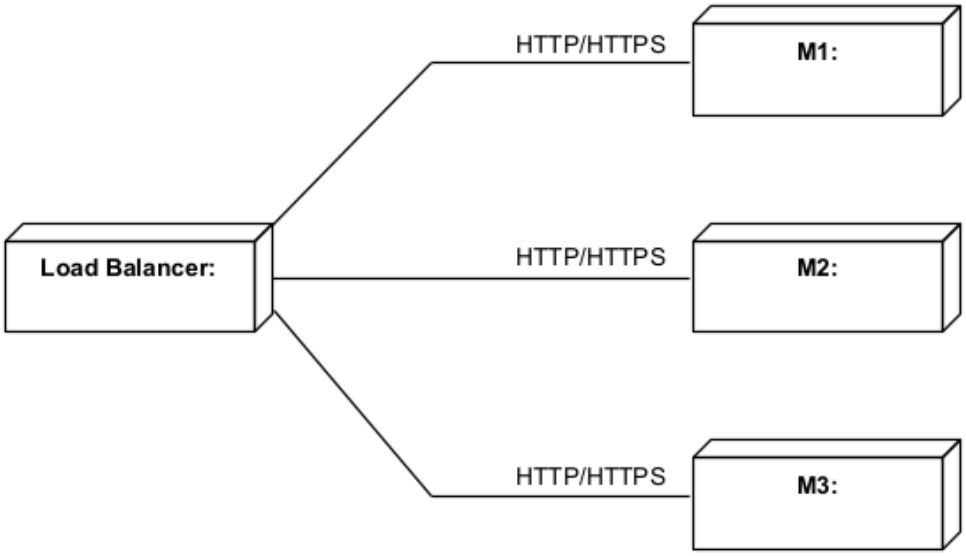

Figura 5: Load Balancer

Normalmente, quando é utilizado o *load balacing*, primeiramente é usado um sistema de monitoramento responsável pela agregação de informação relativa à carga de trabalho, uso dos recursos e do desempenho do sistema em tempo real. Com base nas informações obtidas, é utilizado um algoritmo de balanceamento de carga responsável pela tomada de decisões relativa à distribuição ou redirecionamento de tráfego. Desta forma, as decisões tomadas pelo algoritmo redirecionam os pedidos realizados por um utilizador para os recursos apropriados.

Em (Mishra, Sahoo and Parida, 2020) são referidos seis diferentes tipos de *load balancers*. Cada tipo é projetado tendo em conta as necessidades especificas da arquitetura e requisitos de um sistema.

Um tipo de *hardware load balancer* é uma unidade física responsável pela gestão de um servidor individual numa rede e é usado para distribuir o tráfego por vários servidores de rede.

Um *network load balancer* opera no nível de transporte, nível 4 do modelo OSI. É responsável pela distribuição de tráfego com base nas informações dos cabeçalhos TCP/UDP, nomeadamente endereços IP.

Um *application load balancer* opera no nível de aplicação, camada 7 do modelo OSI e é capaz de rotear o tráfego tendo em conta informações relativas ao conteúdo da solicitação HTTP/HTTPS tais como o *url*, cabeçalhos e cookies. Também oferecem cursos avançados como o balanceamento de carga baseado em cifras e protocolos SSL/TSL.

Um *classic load balancer* é uma solução de balanceamento de carga oferecida pela *Amazon Web Services* (AWS). Proporciona um equilíbrio de carga essencial em várias instâncias do *Elastic Cloud Compute* (EC2), também conhecido por *Amazon Elastic Load Balancing* e funciona no nível de solicitação e no de conexão. Este tipo de balanceamento é destinado a sistemas criados na rede EC2-Classic.

Elastic load balancer, também conhecido por *AWS load balancer* é responsável pela distribuição das diferentes tarefas de entrada pelas várias instâncias do Amazon EC2. Oferece três tipos de balanceadores: *application load balancer*, *network load balancer* e *classic load balancer*.

Por último o *HAProxy load balancer* funciona nas camadas 4 e 7 do modelo OSI. É usado principalmente no *proxy* inverso ou no ALOHA *load balancer.* O dispositivo de balanceamento de carga ALOHA fornece infraestruturas confiáveis e escaláveis e utiliza o HAProxy para desenvolver uma variedade de softwares de balanceamento de carga.

2.3.3 Microsserviços

Num cenário atual onde aplicações enfrentam desafios a lidar com variações da carga de trabalho imprevisíveis é importante existir uma abordagem flexível e adaptável de forma a garantir a eficiência e disponibilidade do sistema.

Outra estratégia para lidar com variações nas cargas de utilização é a adoção de microsserviços (Zeng *et al.*, 2022). Desta forma, é proposta a implementação de funcionalidade do sistema como serviços independentes.

A adoção de microsserviços permite o isolamento de serviços, isto é, promove a minimização de impactos de falhas nos serviços que não estejam a ser utilizados. Esta adoção possibilita também a elasticidade horizontal mencionada anteriormente de serviços específicos para lidar com picos de carga.

Assim, é possível realizar um ajuste dinâmico da capacidade de cada micro serviço. Esta possibilidade é importante em sistemas que tenham diferentes cargas de utilização mediante cada funcionalidade especialmente para evitar o subaproveitamento ou a sobrecarga prolongada de recursos.

Para além de servir como uma eficiência operacional, a adoção de microsserviços tem como benefício a economia de recursos. É possível implementar políticas de escala automática que sejam responsáveis pelo número de instâncias de um microsserviço com base nas métricas de desempenho e carga de trabalho. Desta forma, é possível reduzir o número de instâncias com o objetivo de economizar custos.

Outra vantagem da utilização de microsserviços é relativa à tolerância de falhas. Esta implementação garante a continuidade do serviço como um sistema, mesmo em caso de ocorrência de falhas em componentes específicos. Assim, a recuperação dos serviços afetados torna-se mais rápida e contribui para a minimização dos possíveis impactos no sistema.

Em (Da Silva, Yan De Lima Justino and Adachi, 2022) é referenciado este processo de implementação. Menciona como foi realizada a divisão das funcionalidades identificadas do projeto visado no documento com o intuito de as agrupar em diferentes microsserviços. Esta decomposição do sistema por funcionalidades foi possível graças à utilização do *Context Mapping*.

Context Mapping incorpora o conjunto de ferramentas disponíveis do *Domain Driven Design* (DDD) e contribui para uma melhor gestão da complexidade dos projetos de software. Este padrão aplica-se a qualquer tipo de cenário de desenvolvimento de software e fornece uma visão de alto nível que ajuda os programadores na tomada de decisões.

Na Figura 6, é possível observar do lado esquerdo uma aplicação implantada num servidor. No lado direito da figura é possível visualizar como a adoção de microsserviços foi implementada. Neste segundo caso, o primeiro componente 'App' foi transformado em três microsserviços, tal como refere esta estratégia.

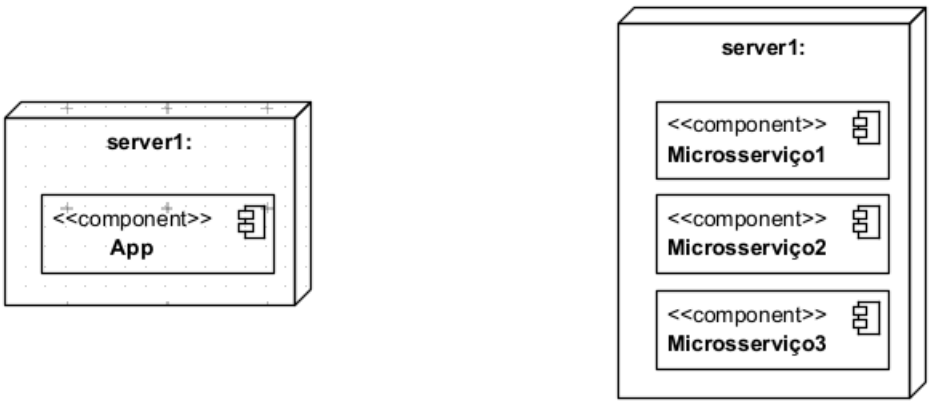

Figura 6: Aplicação monolítica vs Aplicação com microsserviços

2.3.4 Ferramentas de orquestração

Outra estratégia possível para resolver este problema é o uso de ferramentas de orquestração que facilitam a implantação, escalonamento e monitoramento dos serviços do sistema. Para

isto é possível implementar a virtualização baseada em máquinas virtuais e a virtualização baseada em contentores.

Na Figura 7 está representada a arquitetura das virtualizações mencionadas, evidenciando as diferenças entre as duas abordagens.

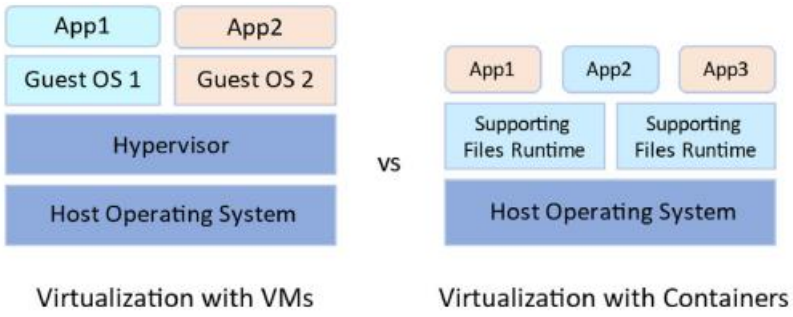

Figura 7: Virtualização baseada em Máquinas Virtuais vs Virtualização baseada em Contentores (Dragoni *et al.*, 2017)

A popularização da virtualização baseada em contentores (Dogani, Namvar and Khunjush, 2023) foi fortemente influenciada pela ascensão dos microsserviços com uma alternativa às arquiteturas monolíticas. A adoção de microsserviços, tal como referido anteriormente, promove a decomposição do sistema em componentes mais pequenos que podem ser implantados, atualizados e escalados de forma independente.

Embora que a implantação de microsserviços em máquinas virtuais é possível, a implantação em contentores oferece maior escalabilidade, flexibilidade e agilidade.

A virtualização baseada em máquinas virtuais envolve o uso de máquinas virtuais. Uma máquina virtual é uma replicação de um computador físico que tem o seu próprio sistema operacional, aplicações e recursos. A arquitetura da virtualização baseada em máquinas virtuais também envolve um software de virtualização, o *hypervisor*, que se encontra responsável pela administração e distribuição dos recursos físicos entre as máquinas virtuais disponíveis.

Cada máquina virtual opera de forma independente e é capaz de operar vários sistemas operativos. A execução de cada microsserviço numa máquina virtual separada requer a replicação do sistema operativo, contribuindo para o aumento do custo económico desta solução. De forma a diminuir este custo, é possível executar vários microsserviços na mesma máquina virtual, porém este método invalida os benefícios da divisão de uma aplicação monolítica em microsserviços.

Por outro lado, os contentores têm o seu próprio sistema de ficheiros, processos e recursos isolados embora partilhem o kernel do sistema operativo anfitrião. Os ficheiros, as variáveis ambientais e as bibliotecas estão entre os componentes que os contentores encapsulam para o funcionamento da aplicação.

Os contentores oferecem mais vantagens relativamente ao uso de máquinas virtuais uma vez que contribuem para uma maior agilidade, uma portabilidade aprimorada e oferecem arquivos mais leves, isto é, conseguem empacotar os recursos necessários num espaço menor. A virtualização baseada em contentores permite que vários programas funcionem numa instância do sistema operacional do *host*.

As ferramentas de orquestração de contentores como o Docker e Kubernetes permitem a gestão de contentores e a autonomização de execução de instâncias por parte dos seus administradores. Estas ferramentas visam maximizar a escalabilidade, implantação e orquestração. A adição e remoção automática de recursos permitem a redução de custos e o aumento da satisfação do cliente.

Ambas as ferramentas, Docker e Kubernetes, simplificam o processo de implantação de aplicações em ambientes em contentores, são projetadas para facilitar a orquestração de microsserviços, integram recursos de segurança responsáveis pela garantia da integridade e segurança dos contentores em execução e permitem a automação de tarefas operacionais tais como a recuperação automática de falhas de contentores.

Uma outra abordagem é o escalamento dinâmico dos serviços dos diferentes módulos do sistema. Desta forma, é necessária a implementação de algoritmos de escalamento inteligentes que sejam capazes de ajustar automaticamente a escala dos microsserviços em resposta à carga de utilização do sistema (Kashyap and Singh, 2023).

Para garantir uma utilização eficiente e eficaz, esta abordagem inclui a previsão da carga de trabalho e a programação de recursos, tem o potencial de melhorar o desempenho do sistema e de reduzir os custos económicos associados ao mesmo. Permite também a redução do tempo de inatividade de recursos e evita o excesso de recursos em utilização.

São utilizados algoritmos tanto no escalamento de máquinas virtuais e contentores, mencionados anteriormente, de modo determinar a sua gestão para otimização do uso de recursos e a distribuir as instâncias de contentores em diferentes nós para otimizar a eficiência, respetivamente.

Tal como referido, também são utilizados algoritmos de balanceamento de carga de forma a distribuir dinamicamente os pedidos entre os servidores disponíveis com o objetivo de evitar a sobrecarga em alguns servidores e a subutilização de outros.

Também é possível a utilização de algoritmos de aprendizagem automática que podem ser incorporados com o intuito de melhorar a capacidade de o sistema prever padrões de carga e ajustar a escala de recursos de uma forma mais inteligente (Soni and Kumar, 2022).

2.4 QP4 - Quais são os contributos da MEAN *stack* na melhoria da elasticidade e escalabilidade de uma aplicação?

A MEAN *stack* é um conjunto de quatro tecnologias modernas que são utilizadas no processo de desenvolvimento de aplicações web dinâmicas e escaláveis, usando apenas uma linguagem de programação em todo o desenvolvimento, JavaScript. É composta pelas tecnologias: MongoDB, Express, Angular e Node.js (Holmes, 2013).

A partir da aplicação da MEAN *stack* é possível realizar o desenvolvimento *full stack* de uma aplicação, ou seja, o desenvolvimento de todas as partes de uma aplicação ou website com esta *stack* é viável. Um desenvolvimento *full stack* é representado por todos os processos desde a base de dados que contém toda a informação armazenada do negócio, *backend* onde toda a lógica do negócio é tratada e aplicada e o *frontend* responsável por disponibilizar toda a informação de negócio aos utilizadores da aplicação via uma interface web.

2.4.1 MEAN – Node.js

Relativamente às tecnologias desta *stack*, o Node.js é uma *framework* de desenvolvimento de código aberto gratuita que permite aos desenvolvedores programar na mesma linguagem tanto no lado do cliente como no lado do servidor (*Node.js*, 2024). A sua arquitetura orientada a eventos permite que o sistema seja capaz de suportar operações de entrada/saída (I/O) de forma assíncrona (*Node.js*, 2024) . Desta forma, o servidor Node.js consegue processar diversos pedidos simultaneamente sem ficar bloqueado à espera de que outras operações I/O sejam concluídas. Assim, o Node.js torna-se uma abordagem muito razoável para a construção de sistemas que exigem uma alta escalabilidade.

Ao contrário dos tradicionais servidores *multithread*, um servidor Node.js é *single-thread* com *loop* de eventos. Para cada pedido de cliente, um servidor tradicional *multithread* cria uma *thread* responsável pela resposta da solicitação efetuada (Holmes, 2013). Como os pedidos são realizados por uma *thread* independente, este mecanismo promove a redução da probabilidade de falha de uma solicitação impactar outro pedido como a facilidade em processar diferentes solicitações simultaneamente, porém promove também a sobrecarga do sistema com o aumento de números de *threads* concorrentes o que pode degradar o desempenho do mesmo.

Por outro lado, um servidor *single-thread* com *loop* de eventos lida com todos os pedidos numa única *thread*. Utiliza um modelo de operações I/O não bloqueante e orientado a eventos ao invés de bloquear as operações de entrada/saída demoradas (Holmes, 2013). A sua arquitetura *single-threaded event loop* permite identificar os pedidos de bloqueio e os de não bloqueio com o intuito de manter a fila de pedidos limitada e processar imediatamente os pedidos de não bloqueio. Como resultado, o servidor em questão controla o processo de novas solicitações enquanto aguarda pela conclusão das operações assíncronas.

Esta arquitetura aumenta a capacidade do servidor para suportar uma quantidade significativa de conexões simultâneas tornando-o altamente escalável. Ao mesmo tempo evita a sobrecarga na gestão de múltiplas *threads* o que resulta num menor uso de memória e CPU (Holmes, 2013).

A tecnologia Node.js tornou-se uma tecnologia extremamente popular também pela sua comunidade ativa e crescente assegurando um suporte robusto para a resolução de problemas no desenvolvimento das aplicações. O Node.js vem com o NPM (Holmes, 2013), um gerenciador de pacotes que facilita a instalação, atualização e gestão de bibliotecas e módulos JavaScript o que permite que os programadores ampliem as suas aplicações mais facilmente.

Para garantir que a aplicação SWiPE seja capaz de lidar com variações significativas nas cargas de utilização, é fulcral realizar uma análise das tecnologias de *backend* disponíveis. A escolha entre servidores *multithreaded* tradicionais e servidores *single-threaded* com *loop* de eventos é uma decisão importante.

Na Tabela 8 é apresentada uma comparação dos dois métodos, destacando as suas principais características, vantagens e desvantagens referidas anteriormente.

Tabela 8: Comparação entre Servidor *Multithreaded* Tradicional e Servidor *Single-Threaded Event Loop*

Característica	Servidor *Multithreaded* Tradicional	Servidor *Single-Threaded Event Loop*
Arquitetura	Utiliza diversas *threads* no processo dos pedidos. Cada *thread* lida com um pedido.	Utiliza uma única *thread* para todos os pedidos, operando num *loop* de eventos.
Gestão de conexões	Quando há muitas conexões simultâneas, geralmente há um *thread* único que gere cada conexão. Isso pode resultar numa sobrecarga do sistema.	Todas as conexões são geridas por uma única *thread*, utilizando operações de I/O não bloqueantes.
Eficácia dos recursos	A sobrecarga do sistema associada à criação e gestão de múltiplas *threads* promove o aumento o uso de memória e CPU.	A ausência de múltiplas *threads* promove a redução do uso de memória e CPU.
Escalabilidade	O número de *threads* que o sistema operacional pode criar e gerir é limitado.	Devido ao modelo não bloqueante, é altamente escalável e capaz de lidar com um grande número de conexões simultaneamente.
Exemplo de Aplicações	Aplicações que exigem processamento paralelo intensivo.	Aplicações como APIs RESTful, aplicações em tempo real e que exigem muitas operações de I/O e concorrência.

Tendo em conta o problema em questão acerca da reengenharia da aplicação SWiPE, dada a necessidade em lidar com variações na carga de utilização de forma eficiente e escalável, optar por um servidor *Single-Threaded Event Loop* é a escolha mais recomendada.

Devido ao seu modelo *Single-Threaded Event Loop,* a tecnologia Node.js é extremamente eficaz na gestão de uma grande quantidade de conexões simultâneas. Deste modo, esta tecnologia é vantajosa para lidar com os picos de utilização da aplicação SWiPE durante os momentos de alta demanda.

Além disso, o Node.js é capaz de escalar dinamicamente os recursos de forma que estes sejam utilizados eficientemente durante os períodos de alta e baixa utilização da aplicação. Outro motivo da escolha do Node.js deve-se ao facto de oferecer um desempenho superior para as operações executadas pela aplicação SWiPE como o envio de emails e todas as interações com a base de dados, melhorando a eficiência geral do sistema.

Desta forma, para além de ser requisitado no problema referente a este documento na medida em que é pedido a construção de uma solução em MEAN, Node.js é a escolha mais adequada para a reengenharia da aplicação SWiPE.

2.4.2 MEAN – Express

Express é uma *framework* web simples e flexível para Node.js que facilita a construção de soluções robustas e escaláveis (*Express.js*, 2024). Oferece uma ampla gama de recursos que permitem tanto o desenvolvimento de APIs como aplicações web tradicionais, mantendo uma sintaxe simples e intuitiva.

Esta *framework* utiliza uma arquitetura baseada em *middleware* permitindo aos programadores a adição de novas funcionalidades de uma forma modular (*Express.js*, 2024) . Além disso, possui um sistema de roteamento flexível, possibilitando definir rotas para os diferentes métodos HTTP (GET, POST, PUT, DELETE). Possui também uma gama vasta de bibliotecas e plugins que podem ser facilmente integrados numa aplicação como bibliotecas de autenticação e validação de dados como o Joi de forma a promover o aumento de segurança e eficiência dos sistemas.

O uso de Express na aplicação SWiPE é benéfico uma vez que é construído sobre o Node.js e herda o seu modelo I/O não bloqueante e *loop* de eventos (Holmes, 2013), resultando num alto desempenho e escalabilidade para aplicações que requerem a manipulação intensiva de solicitações tal como a aplicação SWiPE durante os diferentes períodos de utilização da mesma. A arquitetura baseada em *middleware* e a flexibilidade do roteamento oferecida pela tecnologia Express (Holmes, 2013) permite um desenvolvimento mais rápido e de fácil implementação de novas funcionalidades ou ajustes no sistema.

2.4.3 MEAN – MongoDB

O MongoDB é uma das tecnologias fundamentais da MEAN stack. Esta tecnologia é classificada como um programa de base de dados não relacional (NoSQL) e é reconhecida por armazenar os dados em documentos semelhantes a JSON que permite a existência de uma representação flexível e hierárquica das informações (Holmes, 2013). Desta forma, o MongoDB não impõe um

esquema fixo para o armazenamento dos dados proporcionado o aumento na facilidade aquando dos momentos de alterações dinâmicas nas estruturas dos dados.

Relativamente ao desempenho, o MongoDB é otimizado para operações de leitura e de escrita rápidas, utilizando índices avançados e mecanismos de armazenamento eficientes (*MongoDB*, 2024). Possui também um driver oficial para Node.js resultando na facilidade de integração com aplicações construídas em Express.

A escolha de base de dados aquando da construção de uma nova solução é uma decisão crítica no design de sistemas modernos, especialmente quando se procura alcançar a escalabilidade, flexibilidade e desempenho em aplicações web dinâmicas como o SWiPE. Assim, é importante compreender os diferentes tipos de bases de dados existentes e realizar uma comparação das características entre as duas com a finalidade de escolher a indicada para a aplicação SWIPE.

Os principais tipos de bases de dados são as bases de dados relacionais e não relacionais.

Uma base de dados relacional apresenta um esquema fixo de dados em tabelas, colunas e relações predefinidas. Desta forma, é possível garantir a integridade dos dados através de chaves primarias e estrangeiras, porém, qualquer alteração na estrutura dos dados acrescenta um nível de dificuldade elevado na dificuldade aquando da migração.

Relativamente ao desempenho das bases de dados relacionais, estas utilizam índices e operações *join* com a finalidade de relacionar a informação entre tabelas. Esta prática é importante especialmente em operações complexas em grandes volumes de dados uma vez que aumenta o desempenho da base de dados. Este tipo de base de dados é geralmente escalonado verticalmente, ou seja, tal como foi referido anteriormente, implica o aumento da capacidade de uma única máquina.

Ao contrário deste tipo de base de dados, as bases de dados não relacionais utilizam um modelo de dados flexível sem um esquema fixo. Assim, na eventualidade de existir uma alteração dinâmica na estrutura dos dados, a migração destes não deve ser complexa. O armazenamento dos dados em documentos BSON proporciona às bases de dados não relacional um modelo de dados flexível e permite a formação de dados mais complexos.

Em relação ao seu desempenho, estes tipos de bases de dados são otimizados para operações de leitura e escrita rápidas, geralmente são escalonadas horizontalmente o que implica a distribuição dos dados e cargas de trabalho por múltiplos servidores aumentando a eficiência do sistema na gestão de grandes volumes dados e taxas de tráfego elevadas.

Na Tabela 9 é apresentada uma comparação das duas bases de dados, destacando as suas principais características referidas anteriormente.

Tabela 9: Comparação entre Base de Dados Relacional e Base de Dados Não Relacional

Característica	Base de Dados Relacional	Base de Dados Não Relacional
Modelo de dados	Utilizado um esquema fixo.	Não é utilizado um esquema fixo, é flexível.
Escalabilidade	Vertical.	Horizontal.
Desempenho	Ótima em operações complexas	Ótima em grandes volumes de leitura/escrita.
Escalabilidade	O número de *threads* que o sistema operacional pode criar e gerir é limitado.	Devido ao modelo não bloqueante, é altamente escalável e capaz de lidar com um grande número de conexões simultaneamente.
Integridade dos dados	Suportada por chaves primárias e estrangeiras.	Não é suportada de forma direta.
Flexibilidade dos dados	Baixa.	Alta.

Tendo em conta a tabela anterior e as necessidades do sistema SWiPE na exigência de um suporte de alta flexibilidade, desempenho e escalabilidade para lidar com as diferentes cargas de utilização, a escolha indicada é uma base de dados não relacional devido à sua estrutura flexível que permite adaptações rápidas às mudanças dos requisitos dos dados sem existir a necessidade de uma migração complexa do esquema de dados. A possibilidade de escalonamento horizontal também é uma vantagem perante os picos de tráfego durante os períodos de alta utilização da aplicação visto que garante um desempenho consistente e eficaz.

Uma vez que a escolha do tipo de base de dados indicada é uma não relacional, a escolha do MongoDB para a reengenharia da aplicação SWiPE é justificada e encontra-se em concordância com o requisito deste documento em construir uma solução em MEAN.

2.4.4 MEAN - Angular

Angular é uma *framework* web de desenvolvimento de código aberto que permite a criação de aplicações rápidas e fiáveis aos programadores (*Angular*, 2024). A sua arquitetura é baseada em componentes, ou seja, o Angular promove uma arquitetura modular onde a aplicação é dividida em componentes independentes que podem ser reutilizáveis. Assim, esta arquitetura facilita a criação e manutenção de uma aplicação modular e escalável.

Uma das principais características do Angular é a propriedade *two-way binding.* Esta propriedade proporciona um meio de partilha de informação aos componentes da aplicação, desta forma, esta característica permite a observação e a modificação de valores simultaneamente entre componentes pai e filho. Também possui uma interface de linha de comandos, Angular CLI, que facilita a criação, desenvolvimento e manutenção das aplicações

Angular. Esta ferramenta automatiza tarefas comuns como a criação de novos componente e serviços bem como a execução de testes.

Esta *framework* utiliza como linguagem o TypeScript permitindo um desenvolvimento de código mais seguro visto que esta linguagem ajuda a prevenir erros comuns de codificação. Também é possível observar que possui uma comunidade ativa e existe uma vasta quantidade de recursos úteis para a resolução de problemas.

Tal como foi realizado do lado do servidor, é importante comparar diferentes *frameworks* utilizados no lado do cliente com a finalidade de escolher a melhor *framework* para a reengenharia da aplicação SWiPE. Como o Angular já foi descrito anteriormente, foi escolhida a *framework* React com objetivo de perceber que existem diferentes frameworks que podem ser utilizadas no problema em questão. No final será realizado um estudo comparando as duas *frameworks* populares e com base nas suas características será efetuada a melhor escolha para a reengenharia.

React é uma biblioteca de JavaScript flexível utilizada para a construção de interfaces para o utilizador (*React*, 2024). Tal como o Angular, a sua arquitetura é baseada em componentes, ou seja, React promove a criação de componentes reutilizáveis que podem ser combinados de forma a construir interfaces mais complexas.

Ao contrário do Angular, React utiliza um DOM virtual. Desta forma é possível minimizar as manipulações diretas no DOM real, melhorando a performance da aplicação. Esta *framework* também não partilha a propriedade *two-way binding*. Em contraste ao Angular, utiliza *one-way binding.* Isto significa que existe um fluxo de dados unidirecional aumentando a compreensão do programador relativamente ao percurso dos dados pela aplicação.

Na Tabela 10 estão representadas as principais características das *frameworks* referidas.

Tabela 10: Angular e React

Característica	Angular	React
Arquitetura	Baseada em componentes.	Baseada em componentes.
Linguagem	TypeScript	JavaScript
Data Binding	*Two-way binding*	*One-way binding*
Desempenho	Boa	Boa
DOM	DOM real	DOM virtual
Aprendizagem	Pode ser difícil	Dificuldade moderada
Documentação	Bem detalhada.	Bem detalhada.

2.4 QP4 - Quais são os contributos da MEAN *stack* na melhoria da elasticidade e escalabilidade de uma aplicação?

Com base na informação descrita, ambas as *frameworks* representam uma boa escolha para a reengenharia da aplicação SWiPE. React seria uma boa escolha para o desenvolvimento da nova solução devido à sua flexibilidade e do seu desempenho otimizado, contudo a escolha para o problema em questão é a *framework* Angular, uma vez que apresenta uma arquitetura robusta e ferramentas integradas que facilitam a criação e manutenção de uma aplicação modular e escalável e proporciona um suporte completo para TypeScript.

3. Análise da aplicação SWiPE

Este capítulo descreve a análise da aplicação efetuada, através da exposição da sua arquitetura baseada no modelo C4+1, da descrição dos seus processos principais, desafios atuais da aplicação e da análise das necessidades de escalabilidade da mesma. Por fim, foram definidos objetivos de desempenho e elasticidade.

Tal como descrito no tópico Descrição do Problema, a aplicação SWiPE, inicialmente foi desenvolvida com o objetivo de leitura de ficheiros CSV e posterior envio de emails correspondentes a cada linha desses ficheiros. Eventualmente, com a evolução da aplicação, esta passou a incorporar funcionalidades anteriormente executadas sobre os ficheiros CSV. Desta forma, a aplicação transformou este sistema num com uma elevada interação humana, destinada a apoiar a gestão de propostas e projetos de várias unidades curriculares.

O sistema atual do SWiPE é baseado na MEAN stack, onde o *frontend*, *backend* e base de dados estão implantados numa única máquina. O modo de implantação referido tem funcionado até ao momento, porém o seu funcionamento tem vindo a degradar, especialmente nos períodos com grandes cargas de utilização.

3.1 Arquitetura

Este tópico apresenta a arquitetura implementada referente ao sistema SWiPE. Para descrever a arquitetura será utilizado o modelo C4+1. Desta forma será visível os diferentes pontos de vista da arquitetura da aplicação em diferentes níveis do software.

3.1.1 Nível 1

No primeiro nível, tendo por base o modelo C4+1, o sistema é descrito como um todo. Para este nível, a vista lógica e a vista de processos serão apresentadas.

3.1.1.1 Vista lógica

Tendo em conta o modelo referido, a partir da Figura 8, é possível identificar o sistema SWiPE, bem como uma interface denominada 'UI Utilizadores'. Esta interface representa todos os utilizadores que interagem com a aplicação.

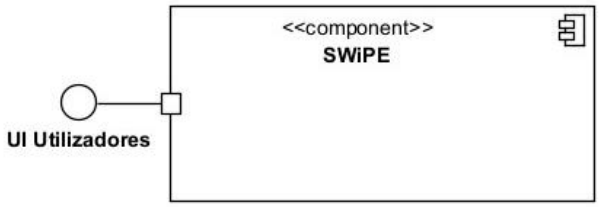

Figura 8: Nível 1: Vista Lógica

3.1.1.2 Vista de processos

Ainda no nível 1, a *user story* 'Registo de uma Organização' é apresentado na Figura 9. Devido ao nível em que está incluído, o diagrama de sequência exposto não apresenta um nível de detalhe elevado.

O ator Anónimo representa um utilizador que não se encontra com sessão iniciada no sistema SWiPE e a *lifeline* exposta representa o sistema visado. Neste diagrama é percetível as ações que o utilizador Anónimo necessita de efetuar com o objetivo de registar uma Organização na aplicação.

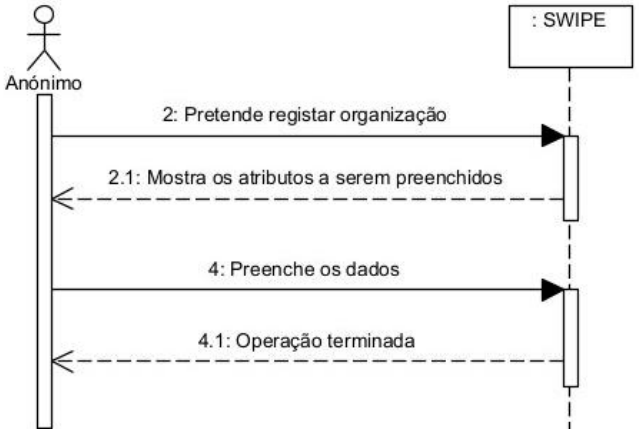

Figura 9: Nível 1: Vista de Processos - Registo de uma Organização

De forma a mostrar diferentes processos, com diferentes atores e diferentes ações realizadas, foi utilizada a *user story* 'Tornar Proposta Favorita' para a criação de um novo diagrama de sequência.

Na Figura 10, o ator representado é um Estudante. Neste processo, o ator já se encontra com a sessão iniciada na aplicação SWiPE e pretende tornar uma proposta favorita. Ao contrário do processo anterior, o Estudante não é solicitado para preencher campos. A partir do diagrama é

possível observar que o sistema apresenta uma lista de proposta disponíveis para o mesmo e este deve escolher uma com o propósito de a tornar favorita. O sistema, por sua vez, tratará das operações necessárias para atender ao pedido do ator.

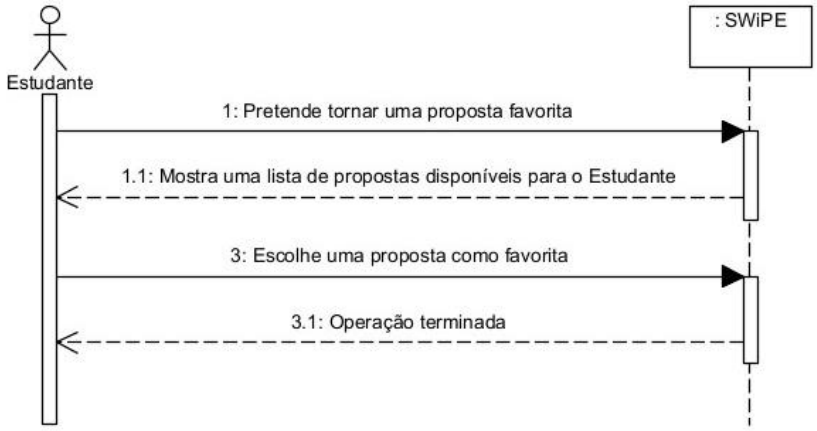

Figura 10: Nível 1: Vista de Processos - Tornar Proposta Favorita

3.1.2 Nível 2

Este nível apresenta um maior detalhe relativamente ao anterior. Assim, neste tópico é realizada uma descrição dos contentores do sistema SWiPE.

3.1.2.1 Vista lógica

Tal como referido, na Figura 11 é possível identificar os contentores do sistema.

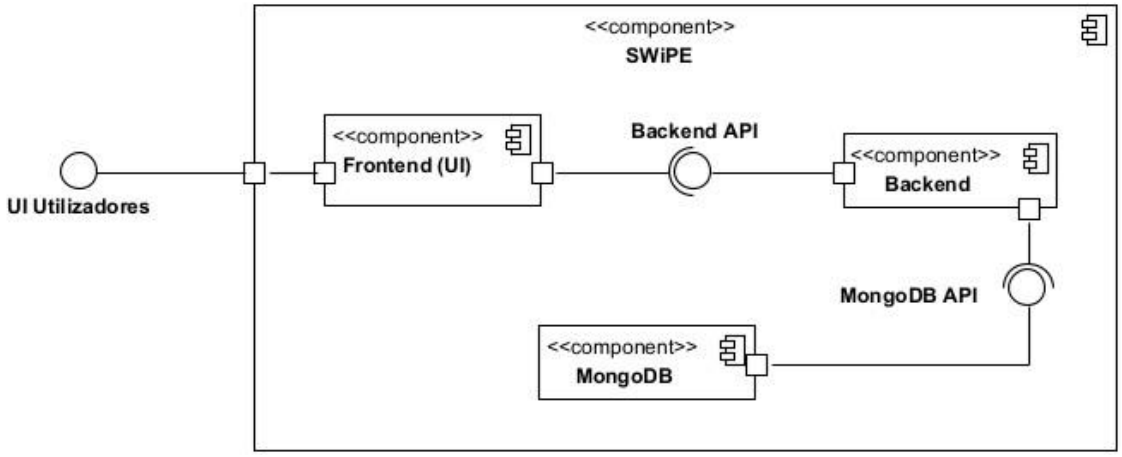

Figura 11: Nível 2: Vista Lógica

O contentor 'Frontend (UI)' é responsável pela interface utilizada pelos utilizadores do sistema. O contentor 'Backend' é responsável pela lógica de negócio da aplicação e o contentor 'MongoDB' é encarregue pelo armazenamento dos dados do sistema.

Perante a figura, também é visível a comunicação entre os contentores definidos. O 'Frontend (UI)' comunica com o 'Backend' e consome os dados do mesmo. O mesmo tipo de comunicação é mantido entre o 'Backend' e o contentor 'MongoDB', ou seja, o 'Backend' consome os dados do contentor da base de dados.

Para além das comunicações referidas, o contentor 'Frontend (UI)', uma vez que está associado à aplicação Angular, é o único contentor a manter uma interação direta com os utilizadores do sistema. Este contentor, como referido anteriormente, também comunica com o 'Backend' com o intuito de aceder aos seus serviços.

O contentor 'Backend' interage com o contentor 'MongoDB' com o objetivo de obter os dados referentes à lógica de negócio do sistema SWiPE.

3.1.2.2 Vista de implementação

A Figura 12 apresenta a vista de implementação do sistema. O package 'UI' representado encontra-se associado ao contentor 'Frontend UI' mencionado previamente. Este componente está associado aos ficheiros da aplicação em Angular utilizados para a execução da mesma.

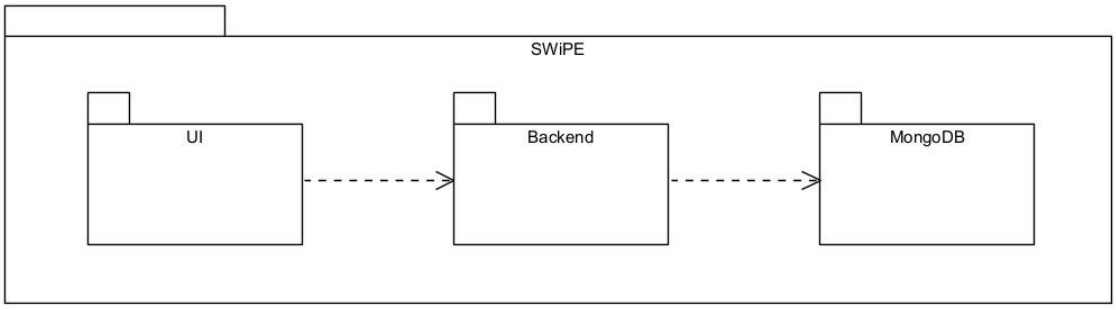

Figura 12: Nível 2: Vista de Implementação

3.1.2.3 Vista de implantação

Atualmente todos os componentes já mencionados encontram-se alocados na mesma máquina.

Assim, na Figura 13, para além destes componentes, é possível observar uma máquina 'localhost:'. Esta máquina é utilizada para representar a aplicação Angular que é executada no Browser de muitas máquinas. Consequentemente, estas máquinas pedem ao componente 'Frontend' apresentado no contentor 'Servidor:' os ficheiros da aplicação *frontend* com o fim de executar a mesma no Browser. Desta forma, o componente denominado 'Frontend' serve como um servidor HTTP.

Para além disso, nesta vista é possível depreender o protocolo utilizado na comunicação entre os diferentes contentores. É de referir também que a máquina 'localhost:' apenas comunica com os componentes 'Frontend' e 'Backend'.

Por sua vez, o componente 'Backend' recebe os pedidos da aplicação SWiPE, através de protocolos HTTP/HTTPS, processa os mesmos e comunica com o componente 'MongoDB' de forma a obter a informação guardada na base de dados.

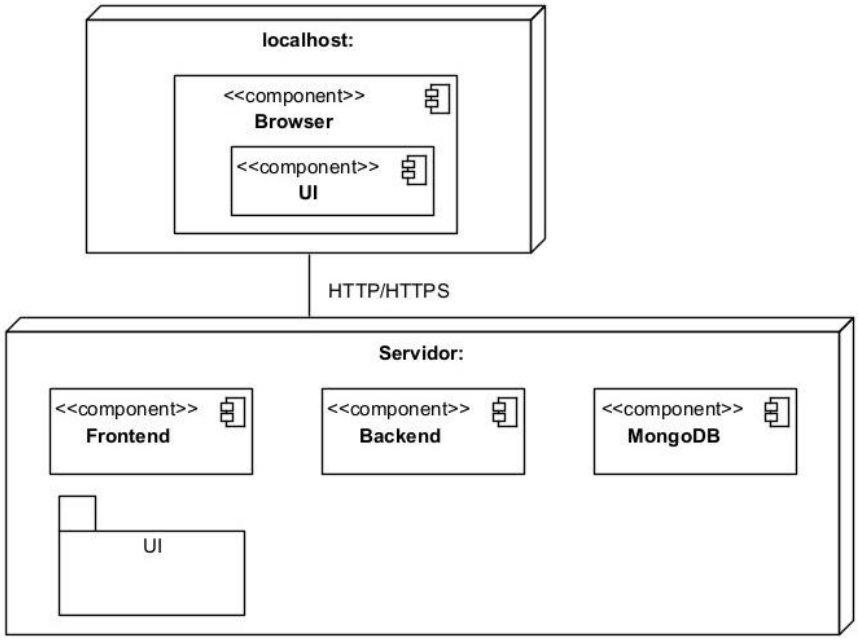

Figura 13: Nível 2: Vista Física

3.1.3 Nível 3 (Backend)

O nível 3 descreve os componentes dos contentores do sistema. Este capítulo foca-se apenas no componente Backend e são apresentadas as vistas lógica e de implementação do mesmo.

3.1.3.1 Vista lógica

A Figura 14 apresenta com mais pormenor aspetos de software relativos ao componente 'Backend'.

Fora do componente, a 'Backend API', tal como mencionado previamente, encontra-se responsável pela comunicação entre os componentes 'Frontend (UI)' e 'Backend'. Dentro do componente, a primeira camada visível é encarregue pela interação direta com o exterior, sendo um ponto de entrada e saída de dados do componente.

O componente 'Routing' lida com a definição das rotas de API, ou seja, é o responsável pelo mapeamento dos endereços *url* dos pedidos para funções específicas do *backend*. Ainda na mesma camada, o componente 'Schema' representa o modelo de dados utilizado pela aplicação. Este modelo define a estrutura de dados das coleções de MongoDB. O 'Data Model' representado, equivale ao modelo de dados que interage com o repositório. Este componente tem como objetivo transformar os dados brutos em objetos de domínio da aplicação.

Na camada ‘Interface Adapters Layer’, o componente ‘Controller’ recebe as solicitações do cliente, através do ‘Routing’. Após a receção dos pedidos, este componente é responsável por interpretar os mesmos e invocar o serviço indicado para o seu processamento. Para além disso, o componente ‘Controller’ também é responsável por formatar as respostas de envio para o cliente, neste caso o ‘Frontend (UI)’. O ‘DTO’ representa os objetos utilizados para transferir os dados entre a camada de aplicação e o domínio. Por sua vez, o ‘Repository’ permite a realização de operações de leitura e escrita de dados por parte da aplicação. Desta forma, este componente encontra-se conectado a ’MongoDB API’.

A camada ‘Application Business Rules’ contém a lógica de negócio do sistema. O componente representado nesta camada é responsável pela implementação de regras de negócio, algoritmos e validações necessárias para o funcionamento da aplicação.

A última camada apresenta o componente ‘Domain Model’. Este representa as entidades e objetos de negócio com que a aplicação lida.

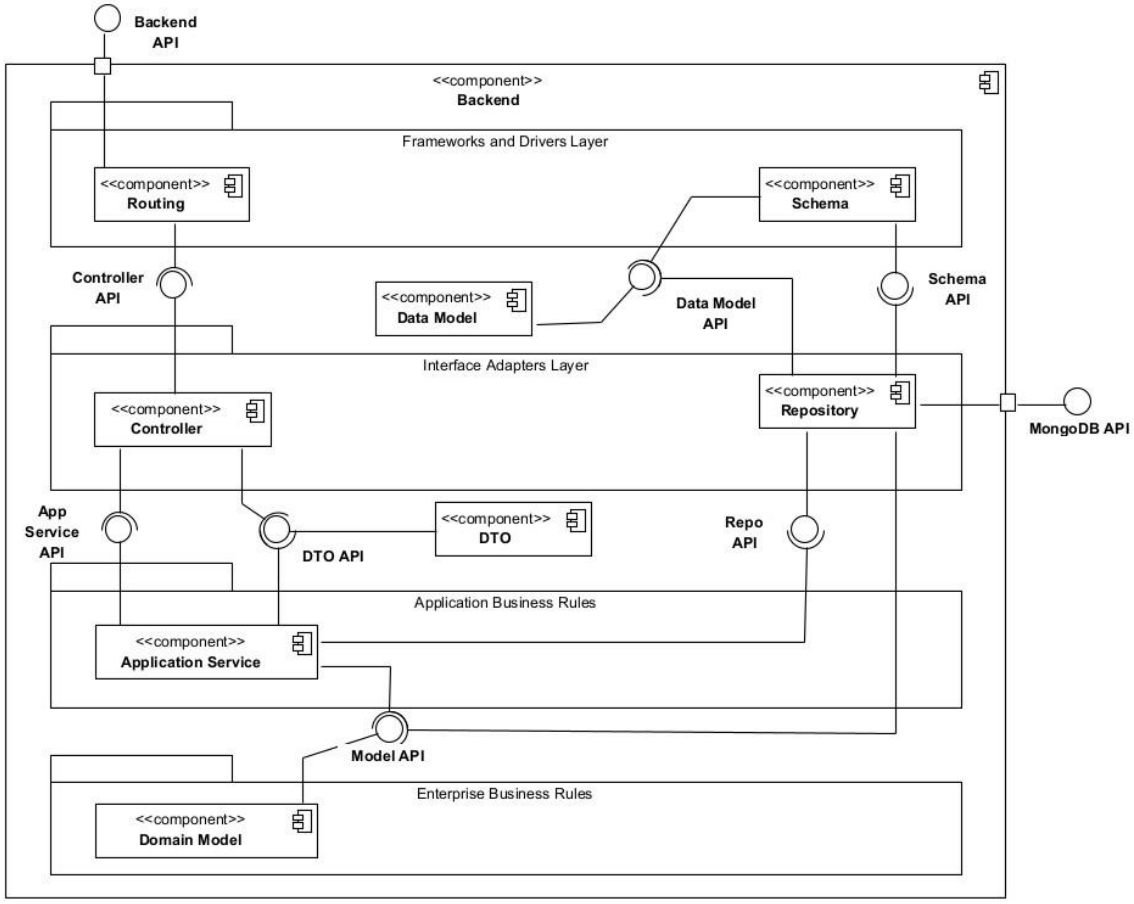

Figura 14: Nível 3: Vista Lógica Backend

3.1.3.2 Vista de implementação

Na Figura 15, com o desenho da vista de implementação do Backend, no nível 3, é possível visualizar como os diferentes componentes do Backend estão organizados em camadas e como a comunicação entre os mesmos é realizada.

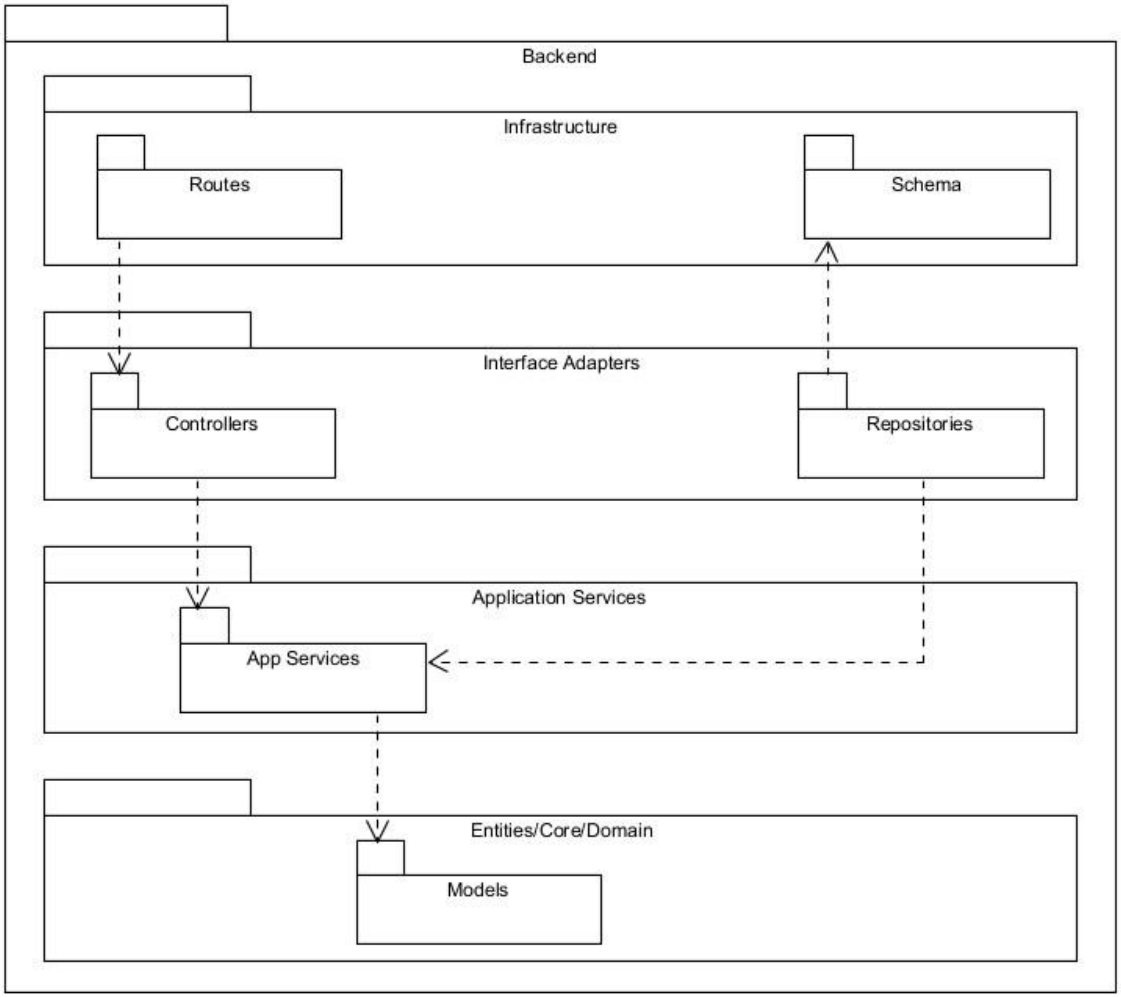

Figura 15: Nível 3: Vista de Implementação Backend

3.2 Processos Principais

Após uma análise detalhada do sistema atual SWiPE, foi possível identificar três processos principais. Estes são o registo de uma organização, todo o processo da gestão de uma proposta, isto é, desde o momento em que uma proposta é submetida até ao momento em que o regente da unidade curricular (RUC) aprova a proposta e todo o processo de gestão da formalização.

Para identificar as operações envolvidas em cada um dos processos, as tabelas (Tabela 11, Tabela 12, Tabela 13) foram criadas de modo a compreender melhor as *user stories* e cenários de cada processo identificado.

Tabela 11: Operações relativas ao processo: Registo de organização

Ator	*User Story*	Descrição
Anónimo	Registo de uma organização	Um anónimo inicia o processo de criação de uma nova organização e indica o gestor responsável
Gestor	Registo do gestor responsável	Envio de um email de confirmação ao gestor responsável e gestor deve confirmar
Gestor	Registo completo da organização	O gestor recebe um email acerca da validação da organização e se a resposta for positiva deve continuar com o processo de registo da organização, associando a UC, introduzindo os colaboradores e outros gestores
Administrador	Decisão do registo da nova organização	O administrador aprova ou rejeita a nova organização. Um email da decisão tomada deve ser enviado ao gestor

Tabela 12: Operações relativas ao processo: Proposta

Ator	*User Story*	Descrição
Proponente	Submissão de uma nova proposta	O proponente submete na plataforma uma nova proposta
RUC	Análise da proposta	RUC recebe um email acerca da submissão de uma nova proposta e procede com a análise e revisão da mesma
RUC	Publicação da proposta	RUC toma decisão sobre a publicação da proposta e um email é enviado ao proponente acerca da decisão do RUC
Estudante	Consulta das propostas disponíveis	Estudante consulta todas as propostas publicadas
Estudante	Torna uma proposta favorita	Perante a lista de propostas publicadas o Estudante seleciona uma proposta como favorita
Estudante	Candidatura a uma proposta favorita	O Estudante candidata-se a uma proposta favorita. A candidatura só é possível se a proposta estiver selecionada como favorita pelo Estudante. Após a candidatura, um email de notificação da ação é enviado ao proponente da proposta

Proponente	Atribuição de uma Proposta a um Estudante	O proponente perante uma lista de candidaturas de diferentes estudantes escolhe um estudante para atribuir a sua proposta. Um email de notificação da atribuição é enviado ao estudante em questão
Estudante	Decisão sobre a atribuição de uma proposta	Estudante aceita ou rejeita a proposta. Um email sobre a sua decisão é enviado ao RUC, se o Estudante recusar. Se o estudante aceitar a proposta, um email deve ser enviado ao estudante em questão com a confirmação da atribuição da proposta e deve ser enviado um email de retirada de candidatura a outras propostas
RUC	Decisão sobre o projeto final	Aprovação ou rejeição do projeto final. Email de notificação da rejeição deve ser enviado apenas ao proponente. Email de notificação da aceitação deve ser enviado ao proponente e ao estudante

Tabela 13: Operações relativas ao processo: Formalização

Ator	*User Story*	Descrição
Estudante	Convite para formar equipa	O estudante seleciona o seu orientador, coorientador e supervisor. Um email deve ser enviado a todos utilizadores como forma de convite
Membros da equipa	Decisão acerca do convite para equipa	Os utilizadores devem responder ao convite
Estudante	Criação do documento de formalização	Estudante começa a elaborar a formalização e submete a mesma. Um email é enviado pelo sistema de forma a informar o orientador responsável que a formalização foi submetida
Orientador	Validação do documento de formalização	Orientador recebe o email de notificação e deve proceder com a validação do documento. Um email é enviado consoante a tomada de decisão do orientador. Se o documento não for validado, o estudante deve receber um email com esta decisão. Se for validado, um email é enviado ao RUC
RUC	Convite a revisor	RUC recebe o email da submissão da formalização e convida revisor. Um email é enviado para o utilizador escolhido
Revisor	Decisão sobre o convite	O revisor recebe o email de convite e toma uma decisão sobre o mesmo. Se não for aceite, um email é enviado para o RUC

Revisor	Revisão da formalização	Se revisor aceitar o convite, deve rever a formalização. Aquando do fim da revisão, um email é enviado ao RUC, informando-o da conclusão desta ação
RUC	Decisão sobre a formalização final	Aprovação ou rejeição da formalização final. Email de notificação da rejeição ou da aceitação deve ser enviado ao estudante

3.3 Desafios Atuais

Tal como já foi referido anteriormente, o desempenho do sistema SWiPE é uma preocupação crítica que se manifesta de múltiplas formas. Antes de mais, a sobrecarga do servidor é um problema recorrente, onde os componentes *backend*, *frontend* e base de dados competem por recursos limitados numa única máquina. Durante os períodos de maior utilização, esta competição resulta na degradação significativa do desempenho do SWiPE, dando origem a respostas mais demoradas por parte da aplicação.

Além disso, a latência elevada é um problema deste sistema. A falta de uma infraestrutura distribuída resulta em tempos de resposta mais demorados. Esta situação não só influencia negativamente a experiência do utilizador, como limita a eficiência operacional da aplicação.

A implantação do sistema SWiPE limita severamente a escalabilidade do sistema. De momento, aplicando a escalabilidade vertical, é possível aumentar os recursos da máquina, no entanto, esta abordagem pode tornar-se impraticável até certo ponto. Cada máquina tem um limite físico de expansão, ou seja, uma máquina possui um número máximo de núcleos de CPU ou de memoria que pode ser instalado. Desta forma, no momento em que estes limites são atingidos é impossível a adição de mais recursos operacionais à máquina.

Por outro lado, a escalabilidade horizontal seria um fator importante a aplicar no sistema. Ao adicionar mais servidores para distribuir a carga de trabalho, de forma a atender às necessidades da aplicação e ser capaz de responder às solicitações dos utilizadores e ao aumento de dados.

Quanto à implantação da aplicação, também é possível concluir que a confiabilidade do sistema é comprometida por um único ponto de falha. Desta forma, como todos os componentes do sistema estão hospedados na mesma única máquina, qualquer falha de hardware ou software provoca uma interrupção total do serviço.

Os desafios mencionados têm um impacto direto e significativo em diferentes áreas. Tal como referido ao longo deste documento, a degradação do desempenho do sistema contribui para tempos de carregamento de páginas web e resultados mais longos, afetando negativamente a satisfação do utilizador.

A confiabilidade reduzida do sistema também pode contribuir para uma perceção negativa por parte dos utilizadores que podem considerar a aplicação pouco confiável. A eficiência operacional também é prejudicada uma vez que os programadores responsáveis pela aplicação devem dedicar o seu tempo na resolução dos problemas associados ao desempenho do sistema em vez de se focarem em melhorias e/ou inovações.

Quanto aos custos monetários associados ao tempo de inatividade ou períodos de baixa utilização, esses aumentam substancialmente.

Os desafios atuais deste sistema assentam a necessidade na reestruturação da arquitetura da aplicação para aumentar o desempenho, a escalabilidade e confiabilidade. A implementação de estratégias como a elasticidade horizontal são essenciais para assegurar a longevidade e eficiência do sistema.

3.4 Análise das Necessidades de Escalabilidade do SWiPE

Garantir a escalabilidade do sistema é um dos principais objetivos deste projeto/documento e como tal, é fundamental assegurar que o sistema SWiPE funcione de forma eficiente e confiável para corresponder às necessidades do crescente número de utilizadores e volume de dados. A análise das necessidades de escalabilidade da aplicação consiste em avaliar os requisitos atuais e futuros, identificando as áreas mais importantes que necessitam de melhorias.

Atualmente, o sistema SWiPE encontra-se implantado numa única máquina e enfrenta os desafios mencionados no tópico anterior. Para abordar as questões relativas ao desempenho da aplicação, é necessária a implementação de soluções para a redução da latência através de estratégias como a otimização de consultas e técnicas de pré-processamento. Desta forma, é possível a redução dos tempos altos de reposta. Relativamente aos períodos de elevada carga de utilização, o sistema deve garantir um nível de desempenho aceitável. As abordagens de balanceamento de carga e a capacidade de escalar horizontal para lidar com a demanda crescente são importantes para a resolução desta necessidade.

Garantir a confiabilidade do sistema também é uma das necessidades da aplicação. É importante eliminar os únicos pontos de falha. Implementar redundância de forma a eliminar estes pontos únicos de falha é possível através da replicação de componentes críticos e a criação de novas infraestruturas de alta disponibilidade com o objetivo de assegurar que as falhas de um componente não comprometam as funcionalidades de outro componente.

3.5 Definição de Objetivos de Desempenho e Elasticidade

Para garantir que a reengenharia da aplicação SWiPE atinja os objetivos de melhorar a eficiência e adaptabilidade da mesma na gestão das variações de cargas de utilização sem recorrer a hardware dispendioso, é essencial estabelecer métricas claras de desempenho. Estas métricas

têm como objetivo a avaliação da nova solução, de todas as alterações implementadas e garantir que as necessidades dos utilizadores foram atendidas.

Após o estudo das métricas existentes, neste documento a métrica referente ao tempo médio de resposta será tido em conta.

Esta métrica diz respeito ao tempo que a aplicação demora ao responder aos pedidos do utilizador e tem como finalidade diminuir o tempo de resposta de modo a garantir uma experiência mais responsiva ao utilizador. Ao avaliar a nova solução será tido em conta o valor alvo de inferior a 100ms nas solicitações realizadas. Tal como referido no capítulo Métricas relativas ao desempenho, a ferramenta Apache JMeter pode ser utilizada para medir o tempo de resposta da aplicação SWiPE.

A métrica referente à taxa de erros também será importante para avaliar a solução construída, de forma a minimizar a taxa de erros com o objetivo de melhorar a confiabilidade da aplicação. Esta métrica mede a quantidade de solicitações que resultam em erros. Tal como na métrica anterior, a ferramenta Apache JMeter pode ser utilizada para calcular a taxa de erros com base nas repostas dos pedidos HTTPS, monitorizando os erros de cliente (4xx) e erros de servidor (5xx). Com esta métrica é pretendido atingir um valor alvo inferior a 1%.

Por último, como o principal objetivo do presente documento é a reengenharia de uma aplicação para lidar com a elasticidade da carga de utilização, torna-se importante avaliar a escalabilidade da solução desenvolvida.

Assim, com a ferramenta Apache JMeter, é possível simular um aumento progressivo na carga utilizando *threads*, ou seja, é possível simular o aumento gradual de número de utilizadores e desta forma medir o impacto dos pedidos dos mesmos na resposta da aplicação. Esta métrica tem como objetivo garantir que a aplicação possa escalar horizontalmente sem existir a degradação significativa do desempenho do sistema.

4. Desenho da Nova Solução com Elasticidade Horizontal

Este tópico descreve a definição de estratégias de otimização para o sistema SWiPE. Seguidamente, é descrito mais detalhadamente os conceitos da alternativa escolhida como a mais indicada e posteriormente serão apresentados diferentes cenários de solução desenhados, aplicáveis à aplicação SWiPE.

4.1 Definição de Estratégias de Otimização

Neste tópico será descrita a utilização do método *Analytic hierarchy process* (AHP) (Vaidya and Kumar, 2006) e como este foi importante para a tomada de decisão relativa à melhor estratégia de otimização para a reengenharia da aplicação SWiPE.

4.1.1 Método AHP

As estratégias de otimização são cruciais para garantir que as necessidades dos utilizadores da aplicação SWiPE são asseguradas de forma eficiente e eficaz. Por forma a compreender onde os esforços por parte dos programadores se devem centrar, o método *Analytic hierarchy process* (AHP) (Vaidya and Kumar, 2006) é importante para estabelecer a prioridade entre os objetivos.

O método AHP é uma metodologia de tomada de decisão multicritério proposta por Thomas Saaty em 1970. Este método promove a resolução de problemas complexos que envolvem múltiplos critérios, permitindo uma análise sistemática e quantitativa das alternativas disponíveis.

Este processo consiste em várias etapas. Inicialmente, é importante clarificar o objetivo principal do projeto, identificar os critérios e subcritérios que influenciam a decisão e de seguida

identificar as alternativas possíveis. Após esta etapa, é necessário estruturar o problema numa hierarquia formada por critérios, subcritérios e alternativas. Para cada nível de hierarquia, devem ser construídas matrizes de comparação paritárias e deve ser utilizada a escala fundamental de Saaty para avaliar a importância relativa dos elementos.

Na Figura 16, é possível verificar os valores desta escala.

Nivel de importância	Definição	Explicação
1	Igual importânica	As duas atividades contribuem igualmente para o objetivo
3	Fraca importância	A experiência e o julgamento favorecem levemente uma atividade em relação à outra
5	Forte importância	A experiência e o julgamento favorecem fortemente uma atividade em relação à outra
7	Muito forte importância	Uma atividade é muito fortemente favorecida em relação a outra
9	Importância absoluta	A evidência favorece uma atividade em relação a outra com o mais alto grau de certeza
2,4,6,8	Valores intermediários	Quando se procura uma condição de compromisso entre duas definições

Figura 16: Escala fundamental de Saaty

4.1.2 Aplicação do método AHP na definição de estratégias de otimização

De acordo com este método, o primeiro passo é a definição do objetivo principal. Assim, tal como foi referido anteriormente, o objetivo principal do projeto em questão é melhorar a eficiência e a adaptabilidade da aplicação SWiPE na gestão das variações de cargas de utilização, sem recorrer a hardware dispendioso.

No contexto atual, os componentes do sistema atual em MEAN *stack,* ou seja, o *backend*, *frontend* e base de dados operam todos na mesma máquina. Desta forma, é importante definir o critério Desempenho uma vez que a performance do sistema pode estar comprometida devido à limitação de recursos compartilhados numa única máquina e por sua vez, não é capaz de responder rapidamente nem de lidar com grandes volumes de solicitações.

posteriormente, foi identificado o critério Escalabilidade dado que é o grande objetivo deste documento. É essencial utilizar este critério no método AHP na medida em que este se refere à capacidade de um sistema crescer e gerir os aumentos de carga de utilização, sem perdas significativas de desempenho.

Por último, o Custo Monetário foi selecionado como um critério a ter em consideração uma vez que qualquer alteração na infraestrutura da solução deve ser financeiramente viável e justificar

o investimento realizado. Este critério engloba os investimentos necessários para implementar e realizar a manutenção da nova solução da aplicação SWiPE.

Relativamente às alternativas utilizadas nesta metodologia, foram utilizadas as estratégias identificadas na secção 2.3.

A Figura 17 representa a árvore hierárquica de decisão alcançada, composta pelos critérios definidos e pelas alternativas a considerar.

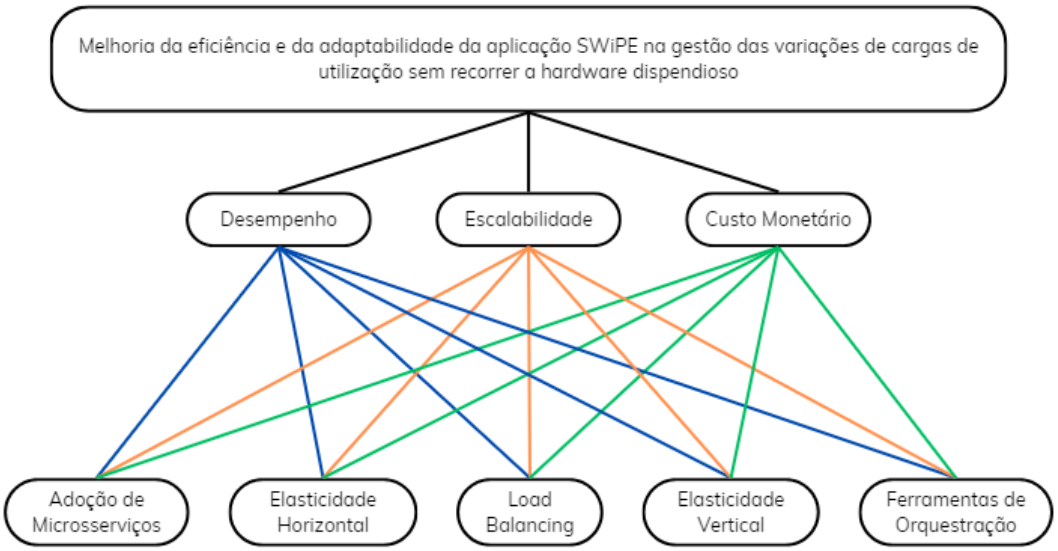

Figura 17: Árvore hierárquica de decisão

Com o intuito de atribuir níveis de importância nas comparações efetuadas ao longo do método AHP, foi utilizada a escala de Saaty referida anteriormente. Com base nesta escala, a Tabela 14 apresenta a matriz de comparação consoante os critérios selecionados no passo anterior.

Tabela 14: Matriz de comparação dos critérios de Segundo Nível

	Desempenho	Escalabilidade	Custo Monetário
Desempenho	1	3	5
Escalabilidade	1/3	1	3
Custo Monetário	1/5	1/3	1

A partir da matriz de comparação representada, é possível observar que o critério Desempenho foi avaliado como ligeiramente mais importante que o critério Escalabilidade devido à necessidade em manter todas as funcionalidades do sistema durante os picos de carga de utilização. Tanto o critério Escalabilidade como o Desempenho foram avaliados como mais importantes que o critério Custo Monetário. Os valores presentes representados com o número 1 devem-se ao facto de que sempre que um critério é comparado consigo mesmo, é igualmente importante a si mesmo.

Por observação da matriz representada é possível concluir que o critério Desempenho, uma vez que possui um valor mais alto em comparação com os restantes critérios, é de extrema importância para o sucesso do sistema. Se este critério falhar, o sistema pode não ser capaz de responder rapidamente às necessidades dos utilizadores, independentemente da escalabilidade ou do custo.

Por sua vez, o critério Escalabilidade reflete a necessidade que o sistema tem em crescer e responder às mudanças na carga de utilização. No entanto, não é tao crucial como o desempenho, uma vez que um sistema escalável não é eficaz se o seu desempenho não for satisfatório.

Embora o Custo Monetário seja um critério relevante, mediante o método AHP foi considerado como o menos crítico dos três critérios identificados. O foco encontra-se mais centrado em assegurar um bom desempenho e escalabilidade.

Depois, os valores da matriz de comparações foram normalizados, por forma a que todos os fatores estejam na mesma unidade. Isto é executável a partir da divisão de cada valor da matriz pelo total da sua respetiva coluna. Estes valores estão presentes na Tabela 15 e na Tabela 16.

Tabela 15: Matriz de comparação dos critérios de Segundo Nível - Soma

	Desempenho	Escalabilidade	Custo Monetário
Desempenho	1	3	5
Escalabilidade	1/3	1	3
Custo Monetário	1/5	1/3	1
Soma	1,5333	4,3333	9

No sentido de identificar a ordem de importância de cada critério é calculada o peso relativo, com base na média aritmética dos valores de cada linha da matriz normalizada.

Tabela 16: Matriz normalizada dos critérios de Segundo Nível

	Desempenho	Escalabilidade	Custo Monetário	Peso Relativo
Desempenho	0,6522	0,6923	0,5556	0,6333
Escalabilidade	0,2174	0,2308	0,3333	0,2605
Custo Monetário	0,1304	0,0769	0,1111	0,1062

A partir dos resultados obtidos, o critério desempenho aparece em primeiro lugar. De forma a avaliar a consistência dos valores apresentados e por isso avaliar a consistência dos pesos relativos é necessário realizar o cálculo da razão de consistência.

Tabela 17: Avaliação da consistência das prioridades relativas

	Desempenho	Escalabilidade	Custo Monetário	Peso Relativo (1)	Matriz Original X Peso Relativo (2)	2/1
Desempenho	0,6522	0,6923	0,5556	0,6333	1,9456	3,071973
Escalabilidade	0,2174	0,2307	0,3333	0,2605	0,7901	3,032969
Custo Monetário	0,1304	0,0769	0,1111	0,1062	0,3197	3,011202

De acordo com os valores obtidos na Tabela 17, e usando como n o valor 3, a partir da tabela proveniente do método AHP é possível calcular a razão de consistência (RC) utilizando as seguintes equações, (1) e (2)

$$IC = \frac{\lambda_{max} - n}{n-1} = \frac{\left(\frac{3{,}071973 + 3{,}032969 + 3{,}011202}{3}\right) - 3}{3-1} = \frac{3{,}038715 - 3}{2} = \frac{0{,}038715}{2} = 0{,}019357 \quad (1)$$

$$RC = \frac{IC}{0{,}58} = \frac{0{,}019357}{0{,}58} = 0{,}033375 \quad (2)$$

Visto que o resultado obtido do RC é inferior a 0,10, podemos concluir que os valores dos pesos relativos utilizados são consistentes.

De seguida foi construída uma matriz de comparação para cada critério com as alternativas identificadas anteriormente. Estas matrizes encontram-se representadas na Tabela 18, Tabela 19 e Tabela 20.

Tabela 18: Matriz de comparação das alternativas - Desempenho

	Elasticidade Horizontal	Elasticidade Vertical	*Load Balancing*	Microsserviços	Ferramentas de Orquestração
Elasticidade Horizontal	1	3	2	3	3
Elasticidade Vertical	1/3	1	1/2	1	1

Load Balacing	1/2	2	1	2	2
Microsserviços	1/3	1	1/2	1	1
Ferramentas de Orquestração	1/3	1	1/2	1	1

Resumidamente, na Tabela 18 estão representadas as comparações efetuadas entre alternativas relativas ao critério Desempenho. A partir da matriz e dos números apresentados, conclui-se que a elasticidade horizontal oferece o melhor desempenho, devido à capacidade de escalar horizontalmente e adicionar mais instâncias conforme necessário.

Por sua vez, a elasticidade vertical pode apresentar um desempenho inferior em relação a soluções que incorporem a alternativa anterior, pois apenas melhora a performance de uma única instância.

A alternativa *Load Balancing* melhora o desempenho do sistema, distribuindo a carga de trabalho, mas pode não ser tão eficiente como a elasticidade horizontal em sistemas com cargas de utilização altas.

A adoção de microsserviços pode ter um desempenho menor devido à sobrecarga de comunicações entre os microsserviços.

A utilização de ferramentas de orquestração auxilia a gestão e coordenação de múltiplas instâncias, porém não afetam diretamente o desempenho de um sistema.

Tabela 19: Matriz de comparação das alternativas - Escalabilidade

	Elasticidade Horizontal	Elasticidade Vertical	*Load Balancing*	Microsserviços	Ferramentas de Orquestração
Elasticidade Horizontal	1	4	3	2	2
Elasticidade Vertical	1/4	1	1/2	1/3	1/3
Load Balacing	1/3	2	1	1/2	1/2
Microsserviços	1/2	3	2	1	1
Ferramentas de Orquestração	1/2	3	2	1	1

A Tabela 19 foi criada tendo em conta que a elasticidade horizontal é mais escalável visto que proporciona o aumento de instâncias conforme as necessidades do sistema. A elasticidade vertical, por sua vez, oferece uma escalabilidade menor uma vez que se encontra limitada relativamente ao aumento de recursos de uma única máquina.

A alternativa *Load Balancing* auxilia a distribuir a carga de utilização, porém não é capaz de aumentar o número de instâncias utilizadas logo, esta alternativa como uma das menos escaláveis.

Por outro lado, a adoção de microsserviços fornecem uma boa escalabilidade ao permitir a criação e manutenção de múltiplos serviços independentes integrantes da aplicação. As ferramentas de orquestração são consideradas essenciais para a gestão do escalamento de aplicações complexas.

Tabela 20: Matriz de comparação das alternativas – Custo Monetário

	Elasticidade Horizontal	Elasticidade Vertical	*Load Balancing*	Microsserviços	Ferramentas de Orquestração
Elasticidade Horizontal	1	2	1	1/2	1
Elasticidade Vertical	1/2	1	1/2	1/3	1/2
Load Balacing	1	2	1	1/2	1
Microsserviços	2	3	2	1	2
Ferramentas de Orquestração	1	2	1	1/2	1

Relativamente à matriz construída para o critério Custo Monetário, é possível concluir que a alternativa considerada como mais económica é a elasticidade vertical, uma vez que esta apenas é responsável pelo upgrade dos recursos de uma única máquina.

As alternativas Elasticidade Horizontal, *Load Balancing* e Ferramentas de Orquestração foram consideradas como alternativas com custos semelhantes correspondendo a um nível moderado. A alternativa Microsserviços foi avaliada como a menos económica devido à sua complexidade e infraestrutura adicional.

Após realizar os cálculos dos testes de consistência de cada matriz de comparação de alternativas produzida, foi possível concluir que todas apresentam um valor inferior a 0,10 e por isso podemos aferir que os valores dos pesos relativos são consistentes.

Por último, de forma a obter a melhor escolha composta para as alternativas, foi utilizada a operação da multiplicação de uma matriz representante dos pesos relativos de cada alternativa por critério por um vetor representado pelos valores dos pesos relativos dos critérios definidos.

Em (3), é possível observar que a alternativa com maior valor referente a 0,37, a elasticidade horizontal é a alternativa avaliada como a mais indicada para a reengenharia da aplicação SWiPE, de acordo com os critérios definidos e dos seus respetivos pesos relativos. Em (3), as letras D,E e C correspondem aos critérios Desempenho, Escalabilidade e Custo Monetário, respetivamente.

$$\begin{array}{c} \\ \text{Elasticidade Horizontal} \\ \text{Elasticidade Vertical} \\ \textit{Load Balacing} \\ \text{Microsserviços} \\ \text{Ferramentas de Orquestração} \end{array} \begin{array}{c} \begin{array}{ccc} D & E & C \end{array} \\ \begin{bmatrix} 0{,}39 & 0{,}38 & 0{,}20 \\ 0{,}12 & 0.07 & 0{,}10 \\ 0{,}23 & 0.12 & 0{,}20 \\ 0{,}12 & 0{,}21 & 0{,}30 \\ 0{,}12 & 0{,}21 & 0{,}20 \end{bmatrix} \end{array} \times \begin{bmatrix} 0{,}63 \\ 0{,}26 \\ 0{,}11 \end{bmatrix} = \begin{bmatrix} 0{,}37 \\ 0{,}11 \\ 0{,}20 \\ 0{,}17 \\ 0{,}15 \end{bmatrix} \quad (3)$$

4.2 Teoria sobre Elasticidade horizontal

Elasticidade horizontal foi identificada como a melhor solução perante a resolução dos desafios enfrentados pelo sistema SWiPE. Esta abordagem promove o escalamento eficiente em resposta ao aumento da carga de utilização, através da distribuição das solicitações dos diferentes utilizadores da aplicação entre as múltiplas instâncias de servidor criadas.

Conforme o descrito em Elasticidade horizontal vs Elasticidade vertical (cf. Secção 2.3.1), a elasticidade horizontal refere-se à capacidade de adicionar e/ou remover instâncias de um recurso de computação, com o intuito de lidar com variações de cargas de utilização. Esta estratégia usada para a otimização de aplicações é frequentemente utilizada através de tecnologias como a virtualização e contentores (Buyya *et al.*, 2009). O objetivo é gerir rapidamente as múltiplas instâncias, ativando ou removendo instâncias tendo por base a necessidade do sistema.

A disponibilidade do sistema como um todo também é aumentada com a aplicação da abordagem da elasticidade horizontal através da distribuição da carga de trabalho entre múltiplas instâncias. Assim, se uma instância falhar, outras instâncias podem assumir a carga e minimizar o impacto da falha no sistema (Dragoni *et al.*, 2017).

Durante os períodos de elevada carga de utilização, o componente *backend* pode limitar o desempenho do sistema SWiPE, devido ao processamento da lógica de negócio e às interações com a base de dados. Com a utilização de contentores e ferramentas de orquestração como Kubernetes (Burns *et al.*, 2016), novas instâncias do componente *backend* podem ser rapidamente provisionadas no sentido de distribuir a carga de processamento.

Por outro lado, o componente *frontend*, responsável pela interface do utilizador, também pode ser escalado horizontalmente. *Load balancers* podem ser utilizados para distribuir o tráfego entre diversas instâncias do *frontend*, assegurando que a interface do utilizador permaneça responsiva, mesmo durante os picos de utilização do sistema SWiPE.

As mesmas estratégias podem ser aplicadas ao nível do componente base de dados.

4.3 Tecnologias Utilizadas

A transição da aplicação para uma arquitetura capaz de suportar a abordagem elasticidade horizontal envolve a integração de diferentes tecnologias, que facilitam a distribuição de carga de trabalho, a replicação de serviços e a orquestração eficiente dos recursos. De seguida estas tecnologias, já abordadas em secção 2.3, serão descritas com maior pormenor. Serão escolhidas as tecnologias utilizadas na solução do problema em questão, justificando as escolhas, bem como os papéis representados na nova arquitetura do SWiPE.

4.3.1 Ferramentas de Orquestração

Este tópico já se encontra detalhado anteriormente em Ferramentas de orquestração, porém este foca-se mais na arquitetura das virtualizações, em detrimento das ferramentas utilizadas para a implementação de arquiteturas de virtualização.

Assim, Docker (*Docker*, 2024) é uma tecnologia de "contenerização" de aplicações que permite a criação de contentores com todos os packages e dependências necessárias para correr uma aplicação. O uso de contentores garante o isolamento do software perante os diversos ambientes de produção. Limitações e problemas de isolamento de contentores e entre contentores está fora do âmbito deste projeto.

No caso da aplicação SWiPE, o Docker pode ser utilizado de forma a desacoplar os componentes do sistema já referidos, permitindo que cada um seja escalado independentemente. A aplicação de contentores assegura que os diferentes componentes do sistema possam ser replicados em diferentes servidores, facilitando a elasticidade horizontal.

Associado à aplicação de contentores, plataformas como Kubernetes (*Kubernetes*, 2024) responsáveis pela automatização da implantação, escalonamento e as operações realizadas em contentores são importantes, contudo não são de utilização obrigatória.

Esta ferramenta é essencial para a elasticidade horizontal, uma vez que permite a gestão de múltiplas instâncias da aplicação SWiPE de forma eficiente, para além de disponibilizar *load balancers* que permitem a distribuição automática de carga de trabalho. A Kubernetes oferece também recursos como a escalabilidade automática, balanceamento de carga e recuperação automática, o que é vital para garantir que o sistema SWiPE possa lidar com as variações de carga de utilização sem interrupções.

4.3.2 Load Balancers

Tal como descrito anteriormente, a utilização de *load balancers* é uma técnica fundamental para a distribuição das solicitações dos serviços entre os múltiplos servidores ou instâncias de uma aplicação. O objetivo desta técnica foca-se em evitar que um servidor ou recurso fique sobrecarregado, melhorando a disponibilidade, eficiência e desempenho geral do sistema. Esta estratégia otimiza o uso de recursos assegurando a continuidade dos serviços das aplicações e pode ser implementada de diferentes formas com diferentes algoritmos de balanceamento de carga de trabalho (Moharir *et al.*, 2020).

Para realizar o balanceamento da carga de trabalho, vários algoritmos podem ser utilizados de modo a distribuir as solicitações, cada um com as suas vantagens e desvantagens dependendo do contexto da aplicação.

O algoritmo Round Robin, é o algoritmo mais simples conhecido, no qual as solicitações são distribuídas sequencialmente entre os servidores ou instâncias disponíveis (Moharir *et al.*, 2020). Este algoritmo é de fácil implementação, porém não tem em consideração o estado atual dos servidores e instâncias, o que pode resultar numa distribuição desigual da carga.

A distribuição dos pedidos aos servidores e instâncias com o menor número atual de conexões ativas também é uma técnica utilizada no balanceamento de carga (Moharir *et al.*, 2020). Esta técnica é ideal para situações nas quais a carga de trabalho por conexão pode variar significativamente.

O algoritmo IP Hash, por outro lado, calcula o *hash* baseado no endereço de IP do cliente e permite a distribuição das solicitações ao servidor ou instância correspondente (Moharir *et al.*, 2020). Este algoritmo é especialmente útil aquando da necessidade dos pedidos de um mesmo cliente serem atendidos pelo mesmo servidor.

Dada a necessidade da aplicação SWiPE conseguir lidar com as variações das cargas de utilização, o algoritmo relativo ao número de conexões ativas parece ser o mais indicado. A utilização deste algoritmo irá permitir que o sistema se adapte dinamicamente às variações da carga de utilização, garantindo que nenhum componente do sistema seja sobrecarregado.

Tal como referido anteriormente, existem diferentes formas de implementar os balanceadores de carga. Em (Moharir *et al.*, 2020) são mencionados dois tipos de *load balancers*: *hardware load balancer* e *software load balancer.*

O balanceamento de carga baseado em *hardware* envolve a utilização de dispositivos físicos dedicados à gestão da distribuição do tráfego de solicitações. O balanceamento de carga baseada em software envolve a utilização de programas ou serviços responsáveis pela gestão da distribuição dos pedidos realizados.

Os *software load balancers* oferecem uma flexibilidade na configuração superior aos *hardware load balancers* uma vez que são executados no *hardware* existente. Paralelamente apresentam

um custo monetário inicial inferior, visto que não obriga à adição de novo *hardware*. Na Tabela 21 as principais características estão apresentadas de uma forma sucinta.

Tabela 21: *Hardware Load Balancer* vs *Software Load Balancer* (Moharir *et al.*, 2020)

	Hardware Load Balancer	*Software Load Balancer*
Implementação	Implementado através de dispositivos físicos	Implementado em software
Custo Monetário	Custo inicial superior a *Software Load Balancer*	Custo inicial inferior a *Hardware Load Balancer*
Escalabilidade	Adição de novos componentes de hardware	Através da adição de novas instâncias
Flexibilidade de configuração	Alta, contudo, é menos flexível	Alta
Exemplos	F5 Load Balancer, A10 Load Balancer	Nginx, HAProxy

Na sequência da tabela apresentada, num cenário descrito para o sistema SWiPE, um sistema prestes a ser modificado para apresentar uma arquitetura distribuída com elasticidade horizontal, um *software load balancer* é a melhor escolha para o problema visado.

Para além da elevada escalabilidade e flexibilidade proporcionadas por um balanceador de carga baseado em software, este pode ser facilmente integrado com as ferramentas de orquestração consideradas previamente, facilitando a automação e a gestão do sistema. Também foi escolhido como opção a implementar em diferentes cenários de implementação da nova solução, uma vez que a implementação de um *hardware load balancer* pode representar um investimento monetário inicial alto, que não se justifica.

Desta forma, os *load balancers* Nginx e HAProxy foram escolhidos para análise, na medida em que já tinham sido brevemente descritos, com o objetivo de optar por um deles para a implementação na aplicação SWiPE

Ambos os balanceadores de carga oferecem serviços de distribuição fortes e são capazes de realizar um desempenho eficiente mesmo com um número elevado de conexões simultâneas (*HAProxy vs NGINX: Performance & Benchmark — RapidSeedbox*, 2024). Todavia, o *load balancer* HAProxy, como é especializado em balancear a carga de trabalho, especialmente tráfego de alta concorrência e alto volume, é ideal para sistemas que exigem uma resiliência e disponibilidade alta como a SWiPE.

Relativamente à configuração dos dois tipos, o balanceador Nginx é conhecido pela sua configuração simples e intuitiva, enquanto que HAProxy possui uma configuração mais detalhada (*HAProxy vs NGINX: Performance & Benchmark — RapidSeedbox*, 2024).

Concluindo, ambos os balanceadores são capazes de proporcionar uma solução viável na distribuição de solicitações. Embora o HAProxy seja ideal para o sistema SWiPE uma vez que lida com grandes volumes de tráfego onde a disponibilidade e a resiliência são imprescindíveis, como o sistema atual corre em HTTPS, e considerando que o *load balancer* Nginx oferece um serviço de proxy reverso, bem como um bom trabalho em balancear cargas de trabalho, este tipo de balanceador de carga é a escolha mais indicada para o sistema SWiPE.

4.4 Lógica de Implementação do Sistema

Tal como referido anteriormente, a aplicação SWiPE é representada por três componentes: o *frontend*, responsável pela interface do utilizador, o *backend*, responsável pela gestão da lógica de negócio e interações com a base de dados e a base de dados responsável pelo armazenamento dos dados do sistema. Estes componentes estão hospedados numa única máquina.

A arquitetura atual é adequada para aplicações com uma carga de utilização baixa. No entanto, para suportar um crescimento futuro e assegurar uma alta disponibilidade e desempenho, é necessário evoluir a arquitetura para uma solução mais escalável e resiliente.

O desacoplamento físico dos componentes, i.e. a implantação dos contentores em nós distintos, é uma estratégia essencial na melhoria da escalabilidade do SWiPE. Apesar da solução pensada não ser referente a uma migração para microsserviços, de forma a facilitar a escalabilidade horizontal, a modularização do sistema é importante com o objetivo de permitir o funcionamento independente dos componentes da aplicação (Newman, 2020).

Posto isto, quando os componentes se encontram alocados em diferentes máquinas, cada um pode ser escalado de forma independente com base nas necessidades do componente específico. Assim é possível evitar o provisionamento de recursos computacionais, bem como a redução de custos operacionais dado que é permitido que estes recursos sejam alocados onde são necessários.

A capacidade de escalar o componente *backend* independentemente do *frontend*, ou seja, a vantagem de aumentar a capacidade de processamento do *backend* sem a necessidade de aumentar a mesma no componente *frontend* resulta numa utilização mais eficiente de recursos, custos operacionais mais baixos e numa melhor gestão do sistema em períodos com cargas de trabalho variáveis.

O desacoplamento dos componentes também torna a manutenção dos mesmos mais fácil. Isto é, modificações e/ou atualizações podem ser aplicadas a componentes individuais sem afetar o sistema como um todo. Esta estratégia, reduz o tempo de *downtime* e facilita a implementação de novas funcionalidades, na medida em que cada componente pode ser tratado de forma isolada (Nygard, 2007).

Para além de aumentar a escalabilidade e manutenibilidade do sistema, o desacoplamento dos componentes do SWiPE também permite o aumento da confiabilidade do mesmo e, desta forma, combater o desafio identificado anteriormente. A separação dos componentes aumenta a confiabilidade do sistema uma vez que a falha de um componente não compromete os restantes componentes. Ou seja, se o componente *backend* falhar, o *frontend* ainda pode funcionar e apresentar mensagens de erro apropriadas enquanto o problema é resolvido.

O desempenho do sistema, por outro lado, também pode ser melhorado através do desacoplamento físico dos componentes. Nos períodos de elevada carga de utilização, o desempenho do sistema, atualmente, pode ser comprometido devido à competição por recursos entre os diferentes componentes.

Deste modo, a separação física dos componentes permite a otimização para cada um destes, melhorando o desempenho do sistema como um todo (Richardson, 2019). Por exemplo, a base de dados pode ser otimizada independentemente da lógica de negócios ou da interface do utilizador.

Na Figura 18, é possível observar uma proposta do SWiPE após o desacoplamento físico dos seus componentes e distribuídos em máquinas diferentes.

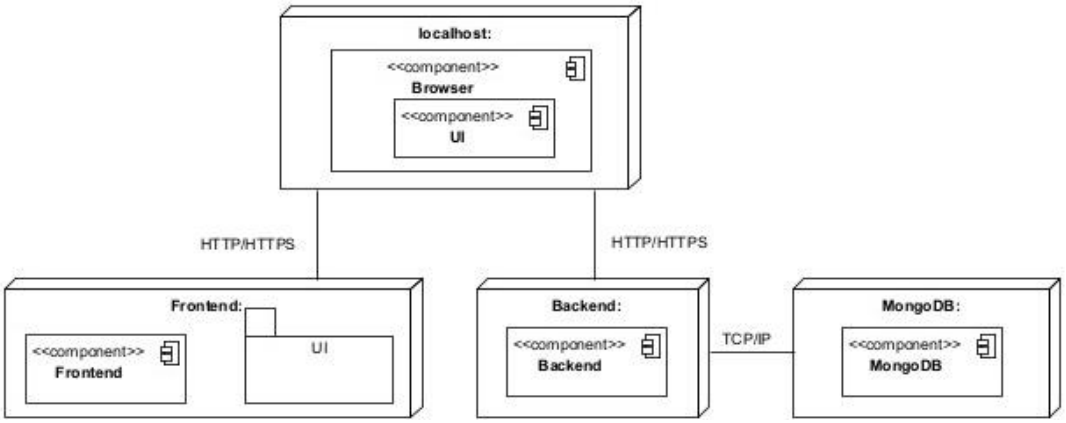

Figura 18: Arquitetura Proposta do Sistema SWiPE após o Desacoplamento Físico de Componentes

Tendo em conta que o custo monetário é um fator importante a ter em atenção aquando da reengenharia de software da aplicação visada, a arquitetura proposta na figura não será considerada uma vez que implica a adição de mais dois servidores físicos.

4.5 Cenários de Escalabilidade Horizontal

Este tópico explora diferentes cenários de escalabilidade horizontal para a aplicação SWiPE, detalhando possíveis arquiteturas e estratégias de expansão, com o auxílio das tecnologias descritas anteriormente.

4.5.1 Cenário 1: Escalabilidade Simples com Contentores Docker

No primeiro cenário de escalabilidade horizontal, o sistema SWiPE é reestruturado de forma a utilizar contentores Docker, e assim separar os três componentes identificados do sistema. Neste caso, o foco é a separação dos componentes da aplicação.

Assim sendo, na Figura 19 é possível observar os componentes envolvidos nesta solução. Foi utilizado um container responsável pela entrega dos ficheiros da aplicação *frontend* ao Browser (Frontend:), um container que executa a lógica de negócio da aplicação (Backend:) e por fim, um container com o objetivo de hospedar o banco de dados (MongoDB:). Nesta figura também se encontra representada o contentor 'localhost:' mencionado anteriormente responsável pela execução da aplicação Angular no Browser.

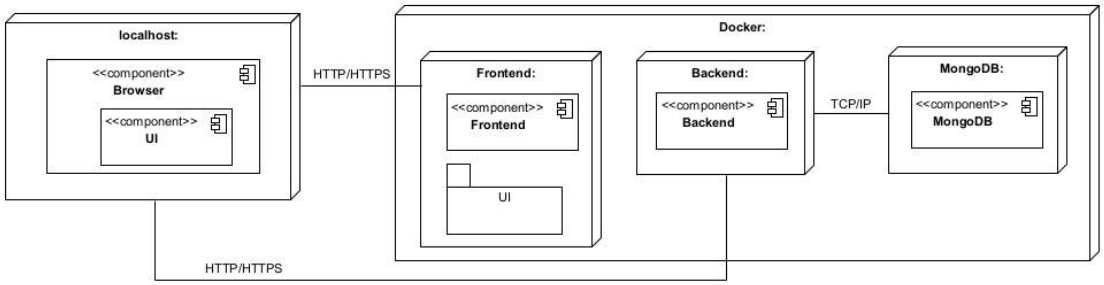

Figura 19: Escalabilidade Horizontal - Cenário 1

Para esta implantação, visando a "contenerizaçao" dos componentes será usada a tecnologia Docker. Na figura o objeto Docker é a representação do servidor no qual os contentores apresentados devem ser executados.

Esta visão de implantação simples ilustra como os diferentes componentes do SWiPE são implantados em contentores isolados. É uma estratégia simples capaz de aumentar o isolamento e portabilidade da aplicação, oferecendo uma base sólida para futuras evoluções, que serão estudadas em cenários descritos posteriormente, como, por exemplo, a introdução de múltiplas instâncias e *load balancers*.

4.5.2 Cenário 2: Escalabilidade com Duas Instâncias de Backend com Contentores Docker e Load Balancer

Neste cenário, a arquitetura de implantação da aplicação SWiPE é composta pelos contentores representados no cenário anterior. Uma instância adicional do container do *backend* foi introduzida na estrutura e como tal também foi adicionado um *load balancer* de forma a distribuir as requisições entre as duas instâncias do *backend*.

Assim, na Figura 20 encontram-se representados os componentes mencionados e como balanceador de carga será tido em conta o *load balancer* escolhido anteriormente, o Nginx.

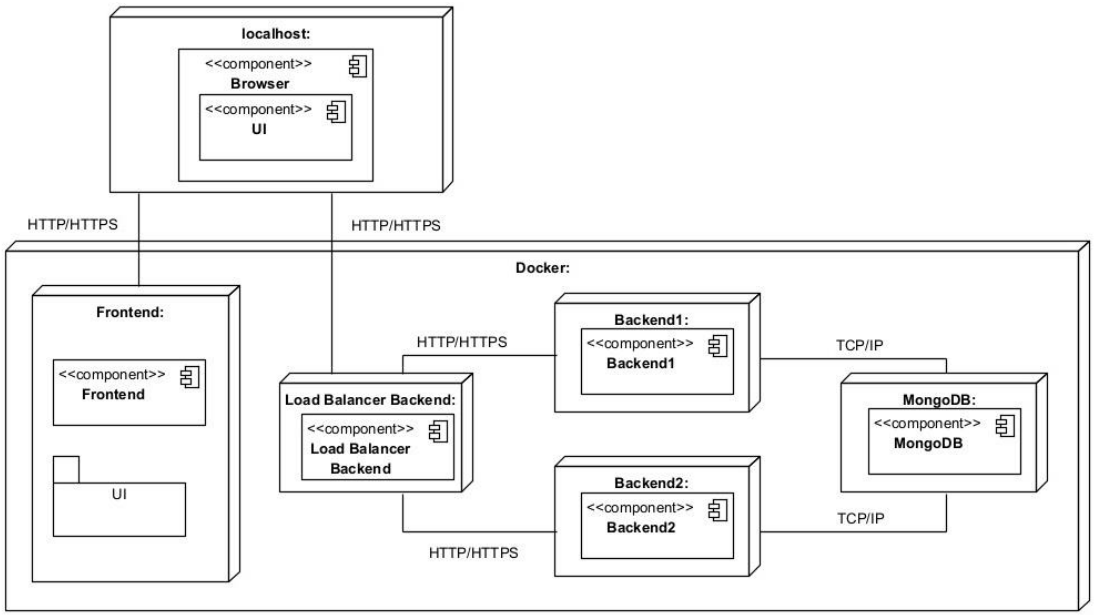

Figura 20: Escalabilidade Horizontal: Cenário 2

Após um cenário proposto com o intuito de avaliar o isolamento e o respetivo desempenho e disponibilidade do sistema, o cenário 2 é importante uma vez que avalia o comportamento da aplicação através do aumento de instâncias dos componentes do sistema.

A decisão de aumentar apenas o número de instâncias do *backend* sem aumentar proporcionalmente o número de instâncias de *frontend* e base de dados é justificada a partir dos seguintes fatores: carga de trabalho associada ao componente, custo monetário e a complexidade associada à escalabilidade da base de dados.

Relativamente à carga de trabalho, o componente de *backend* lida com a maior parte de processamento de dados. Como tal, é normal que sofra uma carga mais intensiva devido ao volume de solicitações que recebe e à complexidade das operações que realiza. Consequentemente é necessário aumentar o número de instâncias.

O componente Frontend, que representa um servidor HTTP, por sua vez, apenas é responsável pela entrega dos ficheiros da aplicação Angular ao Brower, o que reduz a necessidade de o escalar na mesma proporção que o *backend*. A base de dados é um recurso crítico e muitas vezes é um dos obstáculos dos sistemas para alcançar um bom desempenho. No entanto, a escalabilidade horizontal das bases de dados é mais complexa que os componentes já definidos. Esta complexidade deve-se à necessidade de sincronização de dados entre as diferentes instâncias de base de dados criadas.

Assim, neste cenário, ao escalar apenas o componente *backend*, é possível compreender se ao aumentar instâncias de *backend* a carga de trabalho na base de dados é aliviada. A confirmar-se, este componente conseguirá operar de forma mais eficiente sem a necessidade imediata de instâncias de base de dados.

Para além disso, adicionar apenas mais instâncias de *backend* é uma solução mais económica do que escalar os restantes componentes simultaneamente.

4.5.3 Cenário 3: Escalabilidade com Duas Instâncias de Backend e MongoDB com Contentores Docker e Load Balancers

Neste cenário, a aplicação SWiPE é configurada para suportar elasticidade horizontal em todos os seus componentes, utilizando *load balancers* para a distribuição de solicitações.

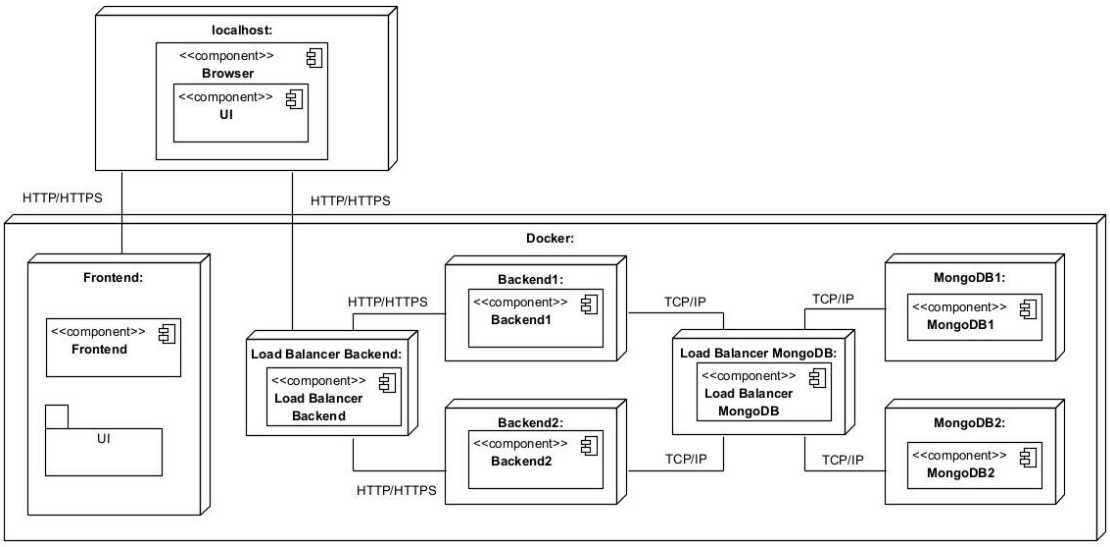

Figura 21: Escalabilidade Horizontal: Cenário 3

Perante a Figura 21 é possível concluir que o 'Load Balancer Backend:' fica responsável pela receção dos pedidos do Browser e pela distribuição dos mesmos entre as instâncias do *backend*. Por sua vez, 'Load Balancer MongoDB' é responsável pela gestão dos pedidos para a base de dados, distribuindo as solicitações efetuadas entre as instâncias replicadas da base de dados.

Este cenário foi considerado como uma possível solução uma vez que todos os componentes se encontram replicados e balanceados, garantindo a disponibilidade do sistema em todos os momentos possíveis, sem que a falha de uma instância comprometa o sistema.

O escalamento horizontal também consegue ser promovido com este cenário, aumentando o número de instâncias de qualquer um dos componentes conforme for necessário. Já os *load balancers* permitem que a carga de trabalho seja distribuída de forma a não sobrecarregar nenhum componente e melhoram o desempenho geral do sistema.

Tal como no cenário anterior, os *load balancers* utilizados serão Nginx.

4.5.4 Cenário 4: Escalabilidade Automatizada com Kubernetes

Este cenário aborda a implementação de ferramentas como Kubernetes na arquitetura da aplicação SWiPE com o objetivo de aumentar a escalabilidade da mesma, bem como a eficiência do sistema. Neste cenário, uma vez que não é possível afirmar qual o componente que suporta uma maior carga de trabalho, a arquitetura é organizada de uma forma homogénea, aproveitando ao máximo as capacidades do Kubernetes.

Na Figura 22, é possível observar a implantação de um master node. Este componente é responsável pela gestão do cluster Kubernetes, ou seja, pela coordenação das atividades do cluster que envolvem a organização dos *pods* e a gestão e manutenção das suas configurações.

O worker node representado é o componente no qual os contentores Docker estão a ser executados e cada um é responsável por hospedar os pods. Cada *pod* pode conter um ou mais contentores Docker.

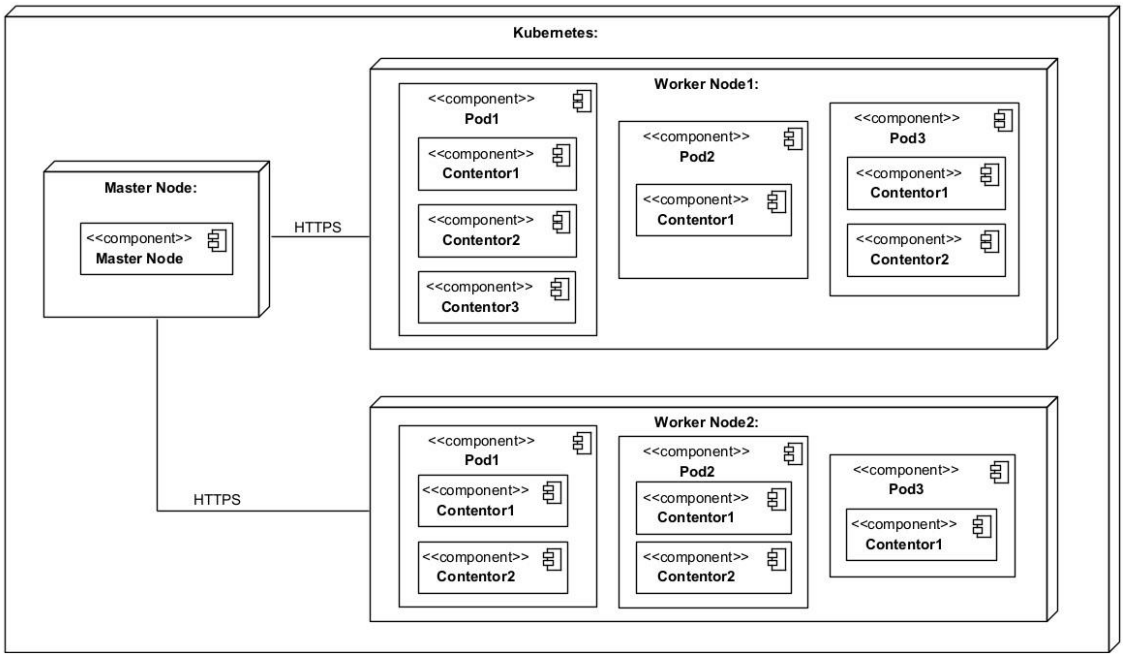

Figura 22: Arquitetura Física Abstrata de Kubernetes

Neste cenário, tal como nos anteriores, todos os componentes foram mantidos isolados uns dos outros e a maior diferença deste cenário é o uso de Kubernetes. Com este cenário pretende-se avaliar como a utilização de Kubernetes influencia os problemas relativos à elasticidade de carga de utilização da aplicação e à necessidade de gestão dos recursos disponíveis.

Em princípio, a ferramenta Kubernetes permite ajustar o número de réplicas de *pods*. Por conseguinte, a aplicação é capaz de se escalar horizontalmente em resposta aos picos da carga de utilização sem a existência de intervenção manual por parte dos programadores. Este facto garante um bom desempenho do sistema sem a sobrecarga dos seus recursos.

Desta forma, este cenário apresenta uma vista física semelhante à do cenário anterior. Na Figura 23, é possível visualizar que a única diferença entre os dois se encontra na representação dos contentores com o atributo 'N'. Isto deve-se ao facto da letra 'N' representar que, neste cenário, é viável a criação de mais instâncias dos respetivos contentores.

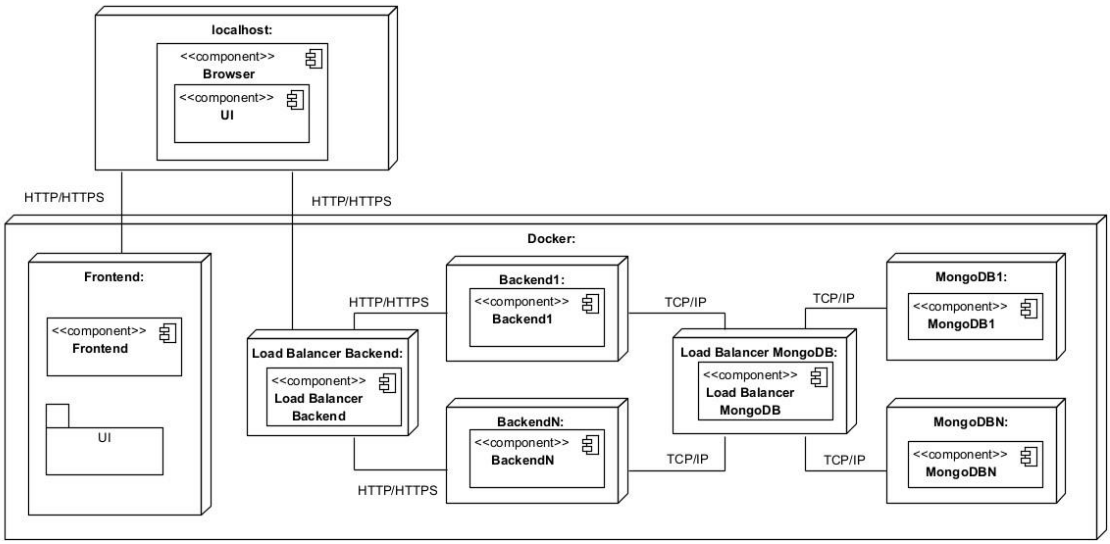

Figura 23: Escalabilidade Horizontal: Cenário4

5. Implementação da Solução

Este capítulo tem como foco a descrição dos detalhes e resultados das implementações das soluções analisadas e desenhadas previamente. Está dividido em tópicos: descrição da implementação da aplicação, escrita na base de dados e testes.

5.1 Descrição da Implementação

A implementação de uma solução que seja capaz de lidar com a elasticidade da carga de utilização da aplicação SWiPE é um processo crucial para a mesma. Este processo envolve uma restruturação da arquitetura do sistema e é detalhado em quatro cenários distintos. Cada um dos processos é representado por uma abordagem incremental para o escalamento e a melhoria do desempenho do sistema.

5.1.1 Cenário 1: Escalabilidade Simples com Contentores Docker

Tal como foi mencionado previamente, este cenário visa a separação dos componentes do sistema em contentores isolados com o auxílio da ferramenta Docker.

Recapitulando, a aplicação foi dividida em três contentores: 'Frontend:' responsável pela interface do utilizador, 'Backend:' responsável pela gestão da lógica de negócio da aplicação e 'MongoDB:' responsável pelo arquivo dos dados e informações do sistema.

Estes contentores são geridos e orquestrados utilizando a ferramenta Docker Compose que proporciona a inicialização, execução e monitorização da aplicação. Neste sentido, para cada um dos componentes *backend* e *frontend*, foi criado um Dockerfile que define o ambiente de desenvolvimento para a execução dos mesmos.

A configuração de *backend* foi realizada em duas etapas: a construção do *backend* e a configuração do Nginx para atuar como proxy reverso. Inicialmente, no ficheiro Dockerfile referente ao componente *backend* foi utilizada a imagem oficial do Node.js como base para o

container. Esta imagem contém o conteúdo necessário para o ambiente de execução de uma aplicação Node.js.

Seguidamente, foi definido o diretório de trabalho dentro do container como '/app' e foram copiados os ficheiros 'package.json' e 'package-lock.json' para o container bem como realizada a execução do comando 'npm install' com o objetivo de instalar todas as dependências necessárias para o projeto. Para além disso, como último passo da construção do backend, foi copiado todo o código da aplicação para o container visado.

Para a configuração do Nginx como proxy reverso foi utilizada a imagem 'nginx:alpine' e instalado o Node.js no container Nginx. Esta instalação é necessária uma vez que é pretendido correr uma aplicação Node.js no container. Em seguida, foram copiados os arquivos da aplicação criados na primeira etapa do Dockerfile para o container Nginx.

Para servir a aplicação através de HTTPS foram copiados os certificados SSL necessários para o container. Foi criado também um ficheiro de configuração personalizado do Nginx para o container de forma a definir as regras de proxy reverso e a configuração do uso de SSL.

Por fim foi exposta a porta 443 que é a porta padrão para HTTPS e foi criado um script para a inicialização do Nginx e a execução da aplicação Node.js dentro do container Nginx.

```
# Utiliza a imagem base oficial do Node.js
FROM node:18 as build

# Define o diretório de trabalho dentro do container
WORKDIR /app

# Copia o package.json e package-lock.json para o diretório de trabalho
COPY package*.json ./

# Instala as dependências da app
RUN npm install

# Copia os arquivos da app para o diretório de trabalho
COPY . .

# Etapa 2: Usar Nginx como proxy reverso
FROM nginx:alpine

# Copia a app Node.js para o container Nginx
COPY --from=build /app /app

# Copia os certificados SSL para o Nginx
COPY ./encryption/mycert.pem /etc/nginx/mycert.pem
COPY ./encryption/rsa_private_key.pem /etc/nginx/rsa_private_key.pem

# Copiar a configuração do Nginx
COPY ./nginx.conf /etc/nginx/nginx.conf

# Cria o diretório para logs e scripts de inicialização
RUN mkdir -p /var/log/nginx /usr/local/bin

# Copia o script de inicialização
COPY start.sh /usr/local/bin/start.sh
RUN chmod +x /usr/local/bin/start.sh
```

```
# Expõe a porta 443 para HTTPS
EXPOSE 443

# Comando para correr script responsável por correr o Nginx e a app Nodejs
CMD ["/usr/local/bin/start.sh"]
```

Código 1: Cenário 1: Dockerfile - *backend*

A configuração do componente *frontend* foi configurada da mesma forma que o *backend*, seguindo as mesmas etapas descritas. A única diferença entre os dois ficheiros Dockerfile encontra-se num comando adicional no *frontend*, no qual o ficheiro constrói a aplicação Angular através do comando 'RUN npm run build' e depois copia os ficheiros da mesma para o Nginx de forma a executar a aplicação através de um servidor HTTP.

```
# Usa uma imagem base do Node.js
FROM node:18 as build

# Define o diretório de trabalho dentro do container
WORKDIR /app

# Copia o package.json e package-lock.json para o diretório de trabalho
COPY package*.json ./

# Instala as dependências da app
RUN npm install

# Copia os arquivos da app para o diretório de trabalho
COPY . .

# Constroi a app Angular
RUN npm run build

# Usa a imagem Nginx como proxy reverso
FROM nginx:alpine

# Copiar o build do Angular para o diretório padrão do Nginx
COPY --from=build /app/dist/SWiPE /usr/share/nginx/html

# Copiar os certificados SSL para o container
COPY ./encryption/mycert.pem /etc/nginx/mycert.pem
COPY ./encryption/rsa_private_key.pem /etc/nginx/rsa_private_key.pem

# Copiar a configuração personalizada do Nginx
COPY ./nginx.conf /etc/nginx/nginx.conf

# Expor a porta 443 para HTTPS
EXPOSE 443

# Iniciar o Nginx
CMD ["nginx", "-g", "daemon off;"]
```

Código 2: Cenário 1: Dockerfile - *frontend*

Relativamente à estrutura do ficheiro Docker Compose desenhado tanto para o serviço *frontend* como *backend,* foi definido o contexto de *build* para os respetivos ficheiros Dockerfile e mapeado as portas pretendidas. Para o serviço *backend* foi necessário passar a informação de variáveis de ambiente para o Docker Compose, fundamentais para o funcionamento da aplicação Node.js.

Relativamente ao serviço de base de dados, foi usada a imagem oficial do mongo, configurado um volume para a persistência dos dados e um comando para iniciar o MongoDB com replicação, essencial para garantir a integridade dos dados e a disponibilidade dos mesmos.

```
version: '3.8'
name: Cenario1

services:
  frontend:
    build:
      context:
        ./frontend/frontend2
    ports:
      - "4200:443”
    networks:
      - my-network
  backend:
    build:
      context:
        ./backend/backend2
    ports:
      - "3000:443"
    environment:
      - dbConnectionString=mongodb://mongo-
c1:27017/SWiPE2p?replicaSet=rs0&directConnection=true&readPreference=primary
    env_file:
      - ./backend/backend2/.env
    depends_on:
      - mongo-c1
    networks:
      - my-network
  mongo-c1:
    image: mongo:7.0
    container_name: mongo-c1
    ports:
      - "27017:27017"
    volumes:
      - mongo-c1_data:/data/db
    entrypoint:
      ["/bin/bash", "-c", "/usr/bin/mongod --replSet rs0 --bind_ip_all"]
    networks:
      - my-network
volumes:
  mongo-c1_data:

networks:
  my-network:
    driver: bridge
```

Código 3: Cenário 1: docker-compose

5.1.2 Cenário 2: Escalabilidade com Duas Instâncias de *Backend* com Contentores Docker e *Load Balancer*

Este cenário explora a implementação de escalabilidade horizontal para o componente *backend* Node.js, com a utilização de Docker para a gestão das instâncias e Nginx como balanceador de carga.

A configuração do *frontend* e base de dados permaneceu igual ao cenário anterior. As únicas alterações observadas neste cenário, relativamente à sua implementação, foi a definição da configuração do componente *backend* bem como a adição de um container Nginx responsável pela distribuição das solicitações entre as instâncias de *backend*.

Assim, o Dockerfile definido para o *backend* neste cenário apenas é composto pela etapa 1 descrita anteriormente, com a adição do comando para iniciar a aplicação Node.js, (CMD ["node", "--trace-warnings", "./bin/www"]).

Relativamente ao ficheiro docker-compose, foi adicionado um container Nginx, com os volumes para a sua configuração e os certificados SSL necessários para uma comunicação segura. Este serviço expõe a porta 3000 do container na mesma porta do host. Os serviços de backend1 e backend2 por sua vez, expõem a porta 3000 como a sua porta interna e as portas 3001 e 3002 do host, respetivamente.

Desta forma, e a partir do ficheiro de configuração do Nginx é possível que este container distribua as solicitações recebidas pelos contentores backend1 e backend2. Neste ficheiro de configuração é definido o algoritmo usado para a distribuição dos pedidos.

```
backend1:
    build:
      context: ./backend/backend2
      dockerfile: Dockerfile-cenario2
    ports:
      - "3001:3000"

backend2:
      build:
        context: ./backend/backend2
        dockerfile: Dockerfile-cenario2
      ports:
        - "3002:3000"

nginx:
    image: nginx:latest
    ports:
      - "3000:3000"
    volumes:
      - ./backend/backend2/nginx-cenario2.conf:/etc/nginx/nginx.conf
      - ./backend/backend2/encryption/mycert.pem:/etc/nginx/mycert.pem
      - ./backend/backend2/encryption/rsa_private_key.pem:/etc/nginx/rsa_privat
e_key.pem
    depends_on:
      - backend1
      - backend2
    networks:
```

```
      - my-network
```

Código 4: Cenário 2: Excertos do docker-compose

```
events {
    worker_connections 1024;
}
http {
    upstream backend_servers {
        least_conn;
        server backend1:3000;
        server backend2:3000;
    }
    server {
        listen 3000 ssl;
        server_name localhost;

        ssl_certificate /etc/nginx/mycert.pem;
        ssl_certificate_key /etc/nginx/rsa_private_key.pem;

        ssl_protocols TLSv1.2 TLSv1.3;
        ssl_ciphers 'ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-
SHA384:ECDHE-RSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384';
        ssl_prefer_server_ciphers off;

        location /api/ {
            proxy_pass https://backend_servers;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }
    }
    # Redirecionamento HTTP para HTTPS
    server {
        listen 80;
        server_name localhost;

        location / {
            return 301 https://$host$request_uri;
        }
    }
}
```

Código 5: Cenário2: Configuração Nginx

O algoritmo referido está presente no ficheiro da configuração do Nginx exposto, least_conn.

Para demonstrar a distribuição de cargas a ser realizada com sucesso, foi utilizada a ferramenta Apache JMeter com o objetivo de criar um utilizador cuja sigla incrementasse um valor relativamente ao último utilizador guardado na base de dados. Para concretizar esta ação, é necessário realizar dois pedidos ao *backend*, um para obter a sigla do último utilizador e outro para a criação do utilizador visado.

Assim, espera-se que os dois pedidos sejam enviados para contentores do *backend* diferentes. Desta forma, todos os componentes foram executados numa consola e a partir da Figura 24 é

possível observar que o pedido GET de utilizadores foi distribuído para o contentor backend2 e o POST foi distribuído para o contentor backend1.

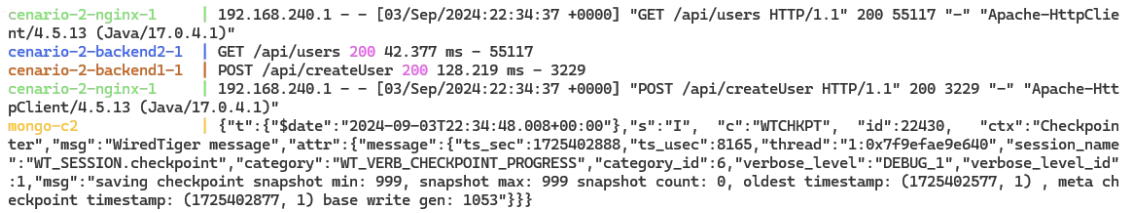

Figura 24: Demonstração da Distribuição dos contentores Backend

5.1.3 Cenário 3: Escalabilidade com Duas Instâncias de Backend e MongoDB com Contentores Docker e *Load Balancers*

O cenário 3 foi projetado de forma a demonstrar a escalabilidade da aplicação SWiPE através da utilização de contentores Docker e o balanceamento de carga. Para a sua implementação foram construídas duas instâncias de *backend* e utilizada a replicação da base de dados.

Assim, a configuração dos contentores de *backend* e de *frontend* foi idêntica ao cenário anterior.

Relativamente à base de dados foi acrescentada uma instância secundária do MongoDB encarregue por replicar os dados da instância primária (mongo-c3). Esta instância primária encontra-se responsável pela gestão das operações de escrita e leitura das informações da aplicação. O MongoDB foi configurado com um replica set para garantir a disponibilidade e a replicação dos dados das bases de dados.

A partir da configuração do replica set com as instâncias mongo-c3 e mongo-c3-replica definidas no docker-compose é possível criar um ambiente de replicação manual. Ou seja, as operações de escrita na instância primária são replicadas para a secundária e desta forma, existe a sincronização dos dados. Num cenário onde a instância primária falhe, a secundária é promovida automaticamente a primária, assegurando que a aplicação continue a funcionar sem interrupções.

A replicação de base de dados foi uma solução encontrada, uma vez que a aplicação SWiPE apenas apresenta um desempenho mais degradado em operações de leitura, ou seja, em ecrãs de listagem. Desta forma, não existe a necessidade de distribuir operações de escrita por diferentes instâncias de base de dados.

No código exposto é possível observar a adição realizada ao docker-compose de forma a adicionar a replicação do MongoDB.

```
version: '3.8'
name: Cenario-3

services:
  mongo-c3:
    image: mongo:7.0
```

```
    container_name: mongo-c3
    ports:
      - "27017:27017"
    volumes:
      - mongo-c3_data:/data/db
    entrypoint:
      ["/bin/bash", "-c", "/usr/bin/mongod --replSet rs0 --bind_ip_all"]
    networks:
      - my-network
  mongo-c3-replica:
    image: mongo:7.0
    container_name: mongo-c3-replica
    ports:
      - "27019:27017"
    volumes:
      - mongo-c3-replica_data:/data/db
    entrypoint:
      ["/bin/bash", "-c", "/usr/bin/mongod --replSet rs0 --bind_ip_all"]
    networks:
      - my-network
volumes:
  mongo-c3_data:
  mongo-c3-replica_data:
networks:
  my-network:
    driver: bridge
```

Código 6: Cenário 3: Excertos do docker-compose

Considerando as configurações de base de dados expostas, a conexão do *backend* à base de dados é: “mongodb://mongo-c3:27017,mongo-c3-replica:27017/SWiPE2p?replicaSet=rs0&readPreference=secondaryPreferred”.

Para demonstrar que a replicação da base de dados está configurada corretamente, foi utilizada a ferramenta JMeter para a criação de novos utilizadores. Inicialmente foi inserido na base de dados apenas um utilizador com a sigla 1190429. Na Figura 25 é possível verificar a existência deste utilizador, mesmo que a escrita do mesmo não tenha sido na instância secundária.

```
rs0 [direct: secondary] SWiPE2p> db.users.find()
[
  {
    _id: ObjectId('66d888d9cda3c837c3ac9c83'),
    sigla: '1190429',
    email: '1190429@isep.ipp.pt',
    nome: 'Beatriz',
    sobrenome: 'Vaz',
    admin: true,
    helpdesk: true,
    __v: 0
  }
]
rs0 [direct: secondary] SWiPE2p>
```

Figura 25: MongoDB: Instância Secundária

Posteriormente foram efetuados pedidos adicionais para a criação de mais utilizadores. Mais uma vez, estas operações de escrita apenas foram registadas na instância primária do MongoDB.

Ainda que as operações de escrita não foram efetuadas na instância secundária, os utilizadores criados devem ser replicados automaticamente para a mesma. Na Figura 26 é possível comprovar que os registos mencionados já se encontram disponíveis.

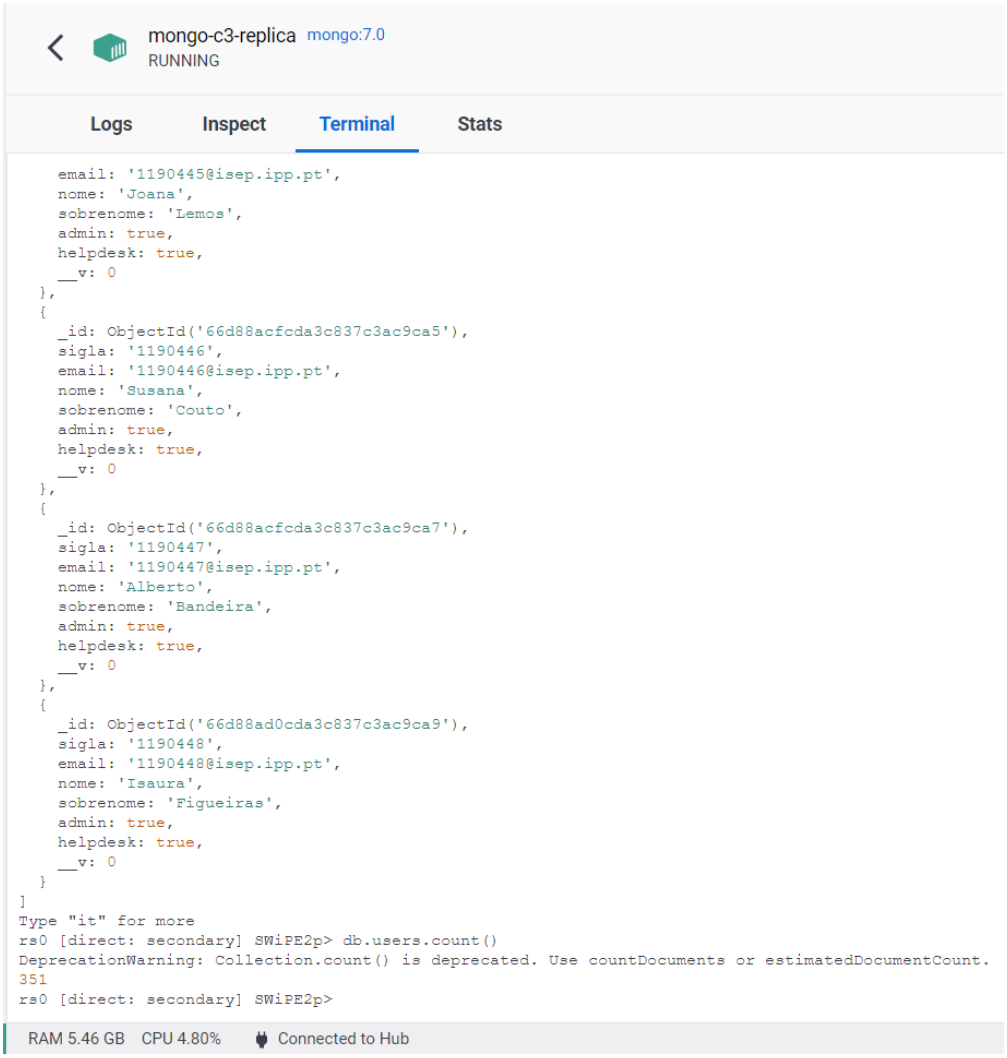

Figura 26: MongoDB: Instância Secundária após operação de escrita

5.1.4 Cenário 4: Escalabilidade Automatizada com Kubernetes

Este cenário promove a implementação de escalabilidade automática num ambiente Kubernetes. A escalabilidade automática é essencial para assegurar que o sistema é capaz de lidar com diferentes cargas de trabalho sem a intervenção manual. Desta forma, ao contrário dos cenários mencionados anteriormente, neste cenário a alteração manual de réplicas de cada componente disponíveis no sistema SWiPE não é necessária.

Para construir a arquitetura desejada, foi configurado o *Deployment*, *Service* e *Ingress* para os componentes *backend* e *frontend*. Para a base de dados MongoDB, foi utilizado um *StatefulSet* e um *Service.*

Um Deployment num ambiente Kubernetes representa um recurso encarregue da gestão de *pods* replicados, na medida em que assegura que esteja um número desejado de *pods* a serem executados em qualquer momento.

O *Deployment* é responsável por definir a imagem do componente a ser usada, a porta que o container expõe, por construir volumes dentro do container referentes a certificados SSL, bem como pela definição do número de réplicas (*pods*) que devem ser executadas simultaneamente. Para este cenário, inicialmente foi configurado a execução do sistema com apenas uma réplica de cada componente.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: 1190429/cenario4-backend:latest
        ports:
        - containerPort: 3000
        volumeMounts:
        - name: ssl-certs
          mountPath: /etc/nginx/mycert.pem
          subPath: mycert.pem
        - name: ssl-certs
          mountPath: /etc/nginx/rsa_private_key.pem
          subPath: rsa_private_key.pem
        resources:
          requests:
            cpu: "500m"
            memory: "1Gi"
          limits:
            cpu: "1000m"
            memory: "2Gi"
      volumes:
      - name: ssl-certs
        secret:
          secretName: ssl-certs
```

Código 7: Cenário 4: Configuração do *Deployment*

No código exposto é possível observar a configuração mencionada com os respetivos valores. O mesmo tipo de configuração foi utilizado no *frontend* com a exceção da porta utilizada que foi a porta 4200.

Para os dois componentes *backend* e *frontend*, foram utilizadas as mesmas configurações definidas nas imagens dos cenários referidos anteriormente. Para além destas configurações, no componente *backend*, as variáveis de ambiente foram adicionadas ao ficheiro Dockerfile

utilizado para a execução da imagem usada neste cenário. Relativamente à conexão à base de dados foi definida a linha: 'ENV dbConnectionString=mongodb://mongo-c4-service:27017/SWiPE2p?replicaSet=rs0&readPref erence=secondaryPreferred', mesmo que o sistema inicie com uma réplica de MongoDB, a adição de novas é considerada e possível. Desta forma, se a adição de novas réplicas for verificada, o componente *backend* já se encontra preparado para tal.

O *Service* é utilizado no ambiente Kubernetes com o fim de expor a execução da aplicação num conjunto de *pods* como um serviço de rede. Tanto para o componente *backend* como *frontend* foram definidas as portas de exposição.

A código exposto referente ao *Service* implementado define as portas mencionadas. O atributo *port* especifica a porta no *Service*, *targetPort* representa a porta dentro dos *pods* para onde o tráfego é redirecionado e *nodePort* é a porta em cada nó que expõe o serviço.

```
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 443
      targetPort: 3000
      nodePort: 32001
  type: NodePort
```

Código 8: Cenário4: Configuração do *Service*

O *Ingress* é um recurso utilizado num ambiente Kubernetes com o intuito de definir as regras de roteamento externo. Oferece também serviços de balanceamento de carga.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: backend-ingress
  namespace: cenario-4
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/secure-backends: "true"
    nginx.ingress.kubernetes.io/load-balance: "least_conn"
spec:
  rules:
  - host: localhost
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: backend-service
            port:
              number: 3000
  tls:
```

```
  - hosts:
    - localhost
    secretName: ssl-certs
```

Código 9: Cenário4: Configuração do *Ingress*

A partir do código apresentado é possível observar que o algoritmo usado para distribuir a carga de trabalho permaneceu o mesmo relativamente aos cenários anteriores. As configurações expostas na figura foram aplicadas também no componente *frontend*.

Por sua vez, o componente MongoDB foi configurado de uma forma diferente relativamente à configuração descrita dos componentes *backend* e *frontend*.

A utilização de um tipo de recurso *SatefulSet* invés da utilização de um *Deployment* é crucial na medida em que o recurso utilizado permite que, mesmo que os *pods* sejam reiniciados, o estado e os dados associados a cada *pod* sejam preservados. O uso do recurso *StatefulSet* contribui para a manutenção da integridade do replica set do mongoDB.

Neste recurso é também indicado o número de réplicas que devem ser executadas no arranque do sistema num ambiente de Kubernetes.

No *Service* definido, ao instruir que o *clusterIP* seja *None* é permitido que cada *pod* construído tenha acesso através de um DNS único. Neste cenário, esta propriedade é importante na medida em que num ambiente com replica set a facilidade da replicação e distribuição de dados é proporcionada. Isto porque cada instância de MongoDB consegue conectar-se corretamente aos outros membros do set.

No código seguinte encontram-se as configurações relativas ao MongoDB.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo-c4
  labels:
    app: mongo-c4
spec:
  serviceName: mongo-c4-service
  replicas: 1
  selector:
    matchLabels:
      app: mongo-c4
  template:
    metadata:
      labels:
        app: mongo-c4
    spec:
      containers:
      - name: mongo-c4
        image: mongo:7.0
        ports:
        - containerPort: 27017
        command:
        - /bin/sh
        - -c
        - mongod --replSet rs0 --bind_ip_all
```

```
          volumeMounts:
          - name: mongo-c4-data
            mountPath: /data/db
          resources:
            requests:
              cpu: "500m"
              memory: "512Mi"
            limits:
              cpu: "1000m"
              memory: "1Gi"
    volumeClaimTemplates:
    - metadata:
        name: mongo-c4-data
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
          requests:
            storage: 5Gi
---
apiVersion: v1
kind: Service
metadata:
  name: mongo-c4-service
  labels:
    app: mongo-c4
spec:
  ports:
  - port: 27017
    targetPort: 27017
  clusterIP: None
  selector:
    app: mongo-c4
```

Código 10: Cenário 4: Configuração do *StatefulSet* e *Service* do MongoDB

Relativamente à escalabilidade automatizada, o ambiente Kubernetes oferece várias abordagens para a sua implementação. A partir da página oficial deste ambiente foi possível identificar a implementação das estratégias *Horizontal Pod Autoscaler* (*Horizontal Pod Autoscaling | Kubernetes*, 2024) e *Vertical Pod Autoscaler* (*Autoscaling Workloads | Kubernetes*, 2024).

A primeira refere-se a uma ferramenta essencial à automática escalabilidade capaz de aumentar o número de réplicas de *pods* do sistema tendo em conta os recursos utilizados pelos componentes integrantes do mesmo (*Horizontal Pod Autoscaling | Kubernetes*, 2024). O CPU pode ser usado como fator contribuinte para este aumento.

Por outro lado, a estratégia *Vertical Pod Autoscaler* permite ajustar automaticamente as solicitações e limites de recursos dos contentores com base no histórico da sua utilização e nas suas necessidades (*Autoscaling Workloads | Kubernetes*, 2024).

Perante as duas abordagens, foi escolhida a estratégia *Horizontal Pod Autoscaler* tendo em conta a necessidade da aplicação SWiPE em se ajustar dinamicamente ao número de réplicas disponíveis com base na carga de trabalho. Esta abordagem é ideal para escalar horizontalmente bem como para manter o desempenho do sistema em todos os períodos de carga de trabalho.

Para a implementação da estratégia escolhida, apenas foi tido em conta o CPU. Para isso, foi definido o valor 50 como targetCPUUtilizationPercentage. No código apresentado encontra-se representado a configuração da abordagem para o componente *backend*. As mesmas definições foram implementadas para o componente MongoDB.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: backend-hpa
  labels:
    app: backend
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: backend-deployment
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Código 11: Cenário4: Configuração *Horizontal Pod Autoscaler*

A replicação do *frontend* não foi efetuada uma vez que depois de construída, a aplicação Angular consiste em ficheiros estáticos que são carregados para o Browser.

A escolha de configurar o Horizontal Pod Autoscaler para manter a utilização da CPU em torno dos 50% permite que o sistema seja capaz de responder rapidamente a aumentos de carga. Desta forma, o tempo necessário para a adição de novas réplicas também é reduzido.

Para além disso, a principal razão para a escolha deste valor deve-se ao fator desempenho. O valor 50% representa um equilíbrio razoável entre a manutenção da aplicação altamente responsiva e a sobrecarga dos recursos disponíveis. Isto é, num cenário onde a utilização da CPU estiver constantemente acima dos 50%, os *pods* definidos encontram-se sobrecarregados, o que implica a criação de mais réplicas com o intuito de distribuir a carga de trabalho.

5.2 Escrita na Base de Dados

Para cada um dos cenários descritos, foi utilizada a ferramenta Apache JMeter, de forma a inserir dados em cada um dos ambientes de uma forma mais automática.

Para obter um número considerável de dados para a avaliação dos cenários construídos, foram criados 400 utilizadores, dos quais 50 são gestores de organizações, 10 são diretores de curso e 20 são RUC de uma unidade curricular. Adicionalmente também foram acrescentados 10 cursos, 20 UC, 40 edições, 4000 inscrições, 2000 docentes, 50 organizações e 10000 propostas.

De forma manual, a partir da interface do utilizador foram criados dois anos letivos, 2023-2024 e 2024-2025 e dois períodos letivos, 1SEM e 2SEM, essenciais à criação de edições.

Na Figura 27 é possível observar os pedidos realizados para obter a quantidade de entidades referidas anteriormente.

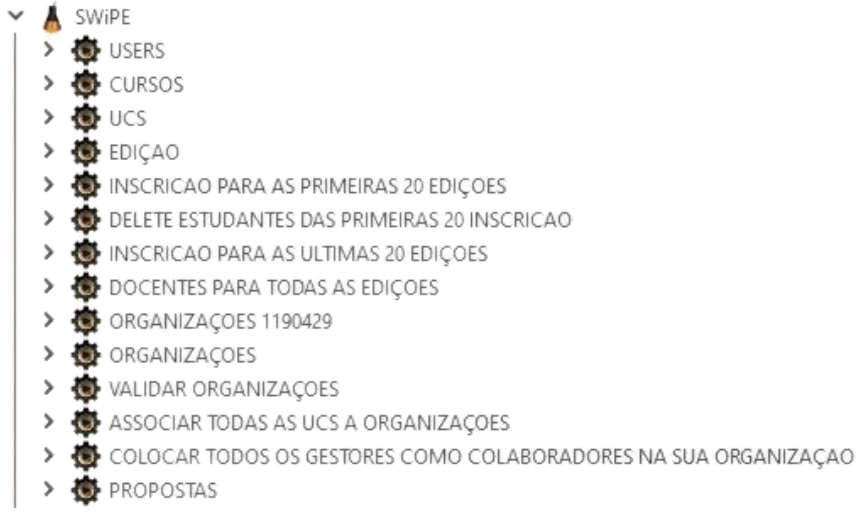

Figura 27: Escrita na Base de Dados a partir do Apache JMeter

Para conseguir aceder a todos os menus da interface do utilizador, o utilizador com a sigla 1190429 tem todo o tipo de papéis disponíveis na aplicação. No entanto, numa situação normal, de funcionamento normal do sistema, uma situação similar nunca deve ocorrer.

5.3 Testes

No presente tópico, é demonstrado o processo de desenvolvimento de testes realizados que têm como objetivo avaliar a escalabilidade, desempenho e consistência dos cenários de arquitetura configurada. Para isto, foi essencial a verificação do comportamento das novas soluções propostas perante diferentes cenários de carga e falhas, com o intuito de entender se a integridade dos dados e a continuidade operacional do sistema conseguia ser mantida sem a degradação do seu desempenho.

Para os testes desenvolvidos foi utilizada a ferramenta Apache JMeter, usada com o propósito de simular o tráfego dos utilizadores da aplicação a partir do envio de pedidos HTTPS.

Foram utilizados testes de carga num *endpoint* específico dedicado à listagem das propostas disponíveis para um determinado utilizador, neste caso um aluno. O *endpoint*: 'GET /api/propostas/listar/todaspropostas/{id}' foi usado em todos os cenários implementados anteriormente, variando o número de *threads* (utilizadores simultâneos) com o intuito de medir tempos de respostas, taxa de erros e comportamentos sob carga.

Para avaliar o comportamento do sistema implementado sob diferentes cargas, foram configuradas 10, 50, 100 e 200 *threads*. Para cada cenário de carga, foram ainda realizados testes com diferentes *ramp-ups* para avaliar como a aplicação se comporta perante o aumento gradual da carga de trabalho.

O *ramp-up* refere-se ao período de tempo em segundos configurados na ferramenta JMeter durante o qual gradualmente aumenta o número de threads que executam a simulação do teste

de carga. Tem como objetivo simular o comportamento real dos utilizadores do sistema visado, bem como evitar a sobrecarga instantânea no servidor sob teste.

Sem a existência de um *ramp-up*, se todas as *threads* fossem inicializadas simultaneamente, o sistema sob teste pode ficar sobrecarregado rapidamente e desta forma, não seria possível obter dados reais sobre o desempenho do sistema sob uma carga de trabalho progressiva. Ao alterar este campo, é possível entender a resiliência do sistema e o seu comportamento relativo a picos de carga de utilização.

Para todos os testes realizados, foram guardadas para uma posterior análise, as métricas que dizem respeito o tempo médio de resposta, taxa de erros e *throughput*.

O tempo médio de resposta é importante uma vez que apresenta o tempo médio que o sistema leva para responder a uma solicitação, a taxa de erros representa o percentual de requisições resultantes de erros, 4xx e 5xx e a métrica *throughput* apresenta o número de pedidos processados por segundo.

5.3.1 Testes: Cenário1

Para registar os testes e resultados obtidos, foi definida a Tabela 22.

Tabela 22: Testes: Cenário1

Thread	*Ramp-up*	Tempo Médio de Resposta [ms]	Taxa de Erros [%]	*Throughput* / pedidos [s]
10	10	22105	0	0.28651
50	10	60062	100	0.71623
50	30	59233	94	0.55855
50	50	55182	70	0.47102
50	120	5948	0	0.40819
100	10	60322	100	1.40625
100	30	60183	100	1.11345
100	50	73642	100	0.65469
100	120	54923	81	0.55904
100	240	5799	0	0.41243
200	10	65906	100	2.35258
200	30	60197	100	2.1978
200	120	60008	99.50	1.11359
200	240	58026	92.50	0.66906

200	400	27238	0	0.47292

A partir dos dados adquiridos, é possível concluir que, para o teste 10 *threads*, o sistema conseguiu processar as solicitações efetuadas sem problemas significativos com uma carga leve de utilização. Já para os testes realizados com 50 *threads*, um *ramp-up* de 120 e 240 permite que o sistema consiga lidar melhor com a carga, reduzindo significativamente o tempo de resposta e eliminando a taxa de erros.

Nos testes relativos a 100 *threads*, aumentar o tempo de *ramp-up* até aos 240 segundos foi fundamental para evitar erros e melhorar o desempenho da aplicação. Perante os testes com 200 *threads*, é possível inferir que um *ramp-up* mais gradual, até aos 400 segundos, foi necessário para suportar a alta carga de utilização do sistema.

Em suma, o *ramp-up* tem um impacto no desempenho do sistema na medida em que o sistema apresentava frequentemente erros e tempos de resposta muito elevados aquando dos *ramp-ups* mais curtos.

Também é possível concluir que, com cargas de utilização mais leves, o sistema conseguiu melhorar o desempenho com *ramp-ups* mais longos, demonstrando que o sistema é capaz de lidar com este tipo de cargas se for dado tempo suficiente para distribuir a carga. Já com cargas de utilização mais elevadas, o sistema teve mais dificuldade em lidar com as mesmas, no entanto com *ramp-ups* mais longos foi possível estabilizar o desempenho da infraestrutura e reduzir os erros apresentados anteriormente.

5.3.2 Testes: Cenário 2

Para registar os testes e resultados obtidos, foi definida a Tabela 23.

Tabela 23: Testes: Cenário 2

Thread	*Ramp-up*	Tempo Médio de Resposta [ms]	Taxa de Erros [%]	*Throughput* / pedidos [s]
10	10	69	0	1.10558
50	10	55	0	5.07666
50	30	61	0	1.69716
50	50	60	0	1.01908
50	120	61	0	0.4296
100	10	49	0	10.05530
100	30	61	0	3.36022
100	50	132	0	2.01812

100	120	73	0	0.84135
100	240	67	0	0.42076
200	10	50	0	19.96805
200	30	52	0	6.69120
200	120	61	0	1.67421
200	240	66	0	0.83729
200	400	66	0	0.50244

Considerando a tabela representada, o sistema apresentou um desempenho estável em todos os testes realizados, com diferentes cargas e valores de *ramp-up*, mesmo nos cenários com 200 *threads*. Este cenário de testes é significativamente diferente do cenário anterior, o que indica que a adição do *load balancer* e a distribuição entre as instâncias do *backend* foram eficazes na gestão de diferentes cargas de utilização por parte da aplicação SWiPE.

Também é possível aferir que o tempo médio de resposta permaneceu baixo em quase todos os testes o que sugere que o sistema se encontra preparado para a escalabilidade horizontal. Relativamente à métrica *throughput*, o sistema apresenta uma boa capacidade de processamento, especialmente em cenários com *ramp-ups* baixos, o que permite concluir que o SWiPE neste cenário 2, é capaz de processar um número elevado de pedidos simultâneos de forma eficiente.

5.3.3 Testes: Cenário 3

Para registar os testes e resultados obtidos, foi definida a Tabela 24.

Tabela 24: Testes: Cenário 3

Thread	*Ramp-up*	Tempo Médio de Resposta [ms]	Taxa de Erros [%]	*Throughput* / pedidos [s]
10	10	501	0	1.09529
50	10	69	0	5.05612
50	30	69	0	1.69699
50	50	110	0	1.01897
50	120	71	0	0.4294
100	10	55	0	10.04016
100	30	68	0	3.35627
100	50	66	0	2.01763
100	120	69	0	0.84129

100	240	66	0	0.42075
200	10	57	0	19.99800
200	30	55	0	6.68717
200	120	64	0	1.67396
200	240	70	0	0.83727
200	400	67	0	0.50242

Por observação da tabela anterior, é possível inferir que os valores são similares aos valores do cenário 2. Isto deve-se ao facto da única diferença entre os cenários ser a adição de uma réplica de MongoDB.

O sistema foi configurado com o atributo 'read preference' na *string* da conexão utilizada à base de dados usada na aplicação com a variável 'secondaryPreferred'. Assim, as leituras efetuadas devem ser realizadas na réplica de MongoDB criada. Na eventualidade da réplica não estar disponível, as leituras devem ser efetuadas na base de dados primária.

Deste modo, ambas as bases de dados criadas estão a ser utilizadas, o que indica que o problema inicial da aplicação SWiPE relacionado com o seu desempenho não está vinculado na capacidade de leitura das bases de dados.

5.3.4 Testes: Cenário 4

Para registar os testes e resultados obtidos, foi definida a Tabela 25.

Tabela 25: Testes: Cenário 4

Thread	*Ramp-up*	Tempo Médio de Resposta [ms]	Taxa de Erros [%]	*Throughput* / pedidos [s]
10	10	12258	0	0.49744
50	10	89708	0	0.50745
50	30	87142	0	0.42036
50	50	49692	0	0.51091
50	120	4414	0	0.41030
50	240	4040	0	0.20920
100	10	217173	0	0.44431
100	30	190129	0	0.45744
100	50	172557	0	0.43104
100	120	155120	0	0.36411

100	240	4233	0	0.41413
100	400	3848	0	0.25020
200	120	392559	0	0.38030
200	240	312460	0	0.36411
200	400	5252	0	0.49763

Perante os dados resultantes dos testes do cenário 4, é possível inferir que o ambiente Kubernetes é capaz de escalar horizontalmente os componentes do sistema. Todos os casos de teste deste cenário foram realizados quando apenas estava a ser executada uma instância de cada *pod* com o intuito de verificar o número de *pods* necessário para as condições (*thread* e *ramp-up*) a serem avaliadas.

A partir da realização dos testes foi possível observar a utilização de quatro *pods* de *backend* em todos os casos executados.

Foi também observado que, apesar da criação de novos *pods*, nos testes com 10 e 50 *threads* e até os 2 primeiros casos de teste com 100 *threads*, estes não registaram qualquer tipo de pedido. Esse comportamento pode ser resultante de uma demora na inicialização dos novos *pods*. Durante este intervalo de inicialização, as solicitações são redirecionadas para os *pods* já existentes originando uma distribuição desigual de carga de trabalho. Nos testes com *ramp-ups* curtos, a carga é aplicada rapidamente o que pode significar que no momento em que os novos *pods* estão prontos para receber carga, parte significativa desta já foi processada pelos *pods* iniciais.

Desta forma, nos casos nos quais todos os *pods* criados estavam ativos e a carga foi distribuída de forma mais uniforme, é observável que à medida que o tempo médio de resposta diminui, o *throughput* aumenta. Este facto é um indicador positivo do desempenho do sistema, sugerindo que os recursos são utilizados de forma eficiente.

Este cenário é muito semelhante ao anterior pelo que a diferença nos valores observados nos dois pode ser atribuída à demora na inicialização dos novos *pods*. Esta demora causou uma distribuição desigual dos pedidos efetuados, especialmente em testes com o uso de *ramp-ups* curtos, impactando o tempo médio de resposta e o *throughput*.

5.3.5 Comparação dos 4 Cenários Implementados

A partir dos resultados obtidos relativos aos cenários implementados, para uma melhor perceção dos dados resultantes, foi criado um gráfico de barras. Desta forma é possível verificar toda a informação pretendida relativa ao tempo médio de resposta do sistema. Este gráfico encontra-se representado na Figura 28.

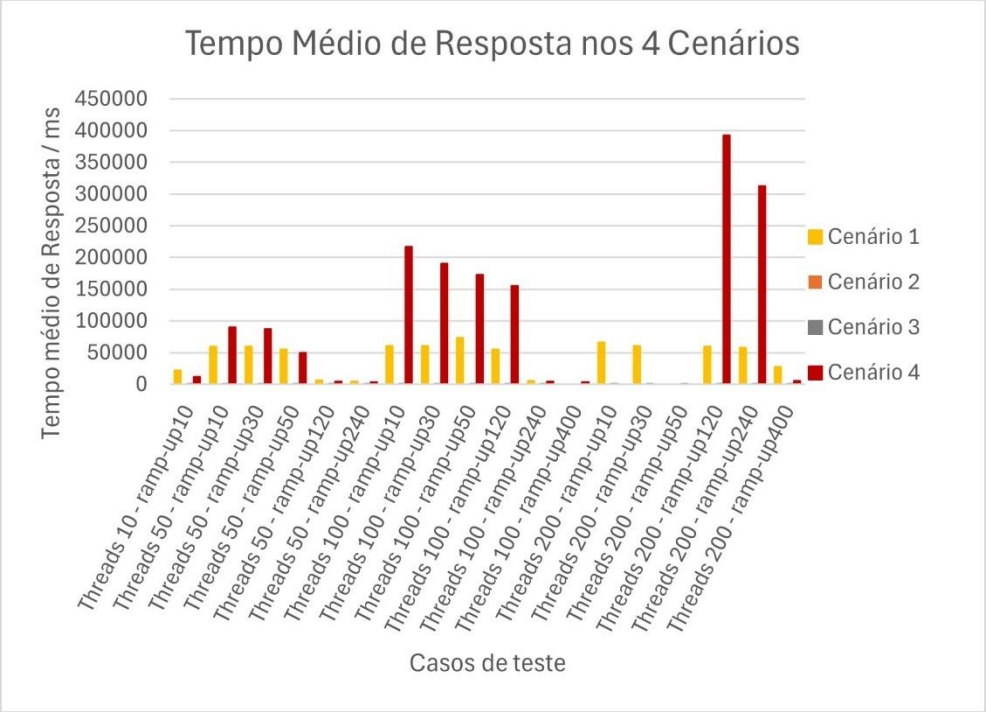

Figura 28: Comparação do Tempo Médio de Resposta nos 4 Cenários

Perante a figura é percetível que os valores obtidos associados ao tempo médio de resposta nos cenários 2 e 3 são os mais baixos e que por isso apresentam um melhor desempenho relativamente aos restantes cenários implementados.

Também é possível observar a diferença significativa dos valores do cenário 4, cuja explicação já foi referida na secção 835.3.4. É de referir que para este cenário não foram efetuados testes com *ramp-up* de 10, 30 e 50 para testes com *threads* a 200, no entanto é observado em todos os casos de teste que à medida que o valor de *ramp-up* aumenta o valor do tempo médio de resposta diminui e melhora o desempenho da aplicação.

O mesmo tipo de gráfico foi criado para a análise do *throughput* de todos os cenários implementados. A partir do mesmo, é possível identificar que os cenários 2 e 3 apresentam maiores valores de *throughput*. Desta forma é possível aferir que estes cenários possuem uma configuração mais eficaz para lidar com um número elevado de solicitações simultâneas.

Os valores mais baixos de *throughput* resultam do cenário 4. Isto pode indicar, tal como foi referido anteriormente, que apesar de possuir mais réplicas a ser utilizadas em comparação com os restantes cenários, a sua implementação pode estar a enfrentar problemas de inicialização dos *pods*.

Na Figura 29, o gráfico mencionado encontra-se exposto.

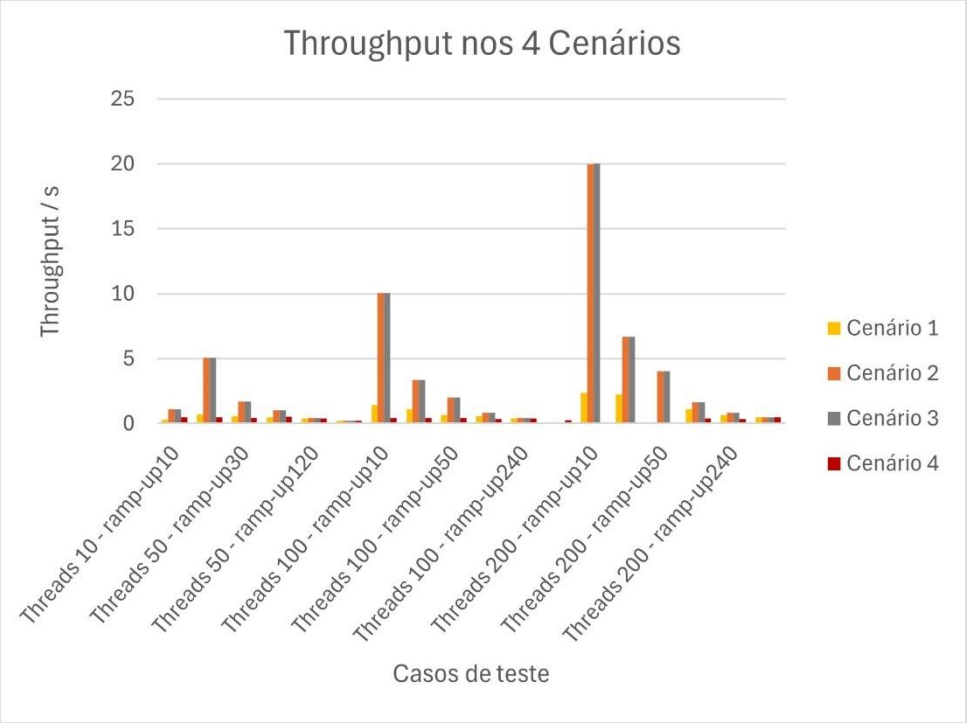

Figura 29: Comparação do *Throughput* nos 4 Cenários

6. Conclusão

Neste capítulo são descritos os objetivos atingidos face aos objetivos definidos no início do projeto. Seguidamente, é realizada uma reflexão sobre as limitações do trabalho desenvolvido e quais as melhorias e soluções que poderão enaltecer o valor do projeto.

6.1 Objetivos Concretizados

O estudo realizado focou-se na exploração de diferentes cenários de escalabilidade e automação com o objetivo de melhorar o desempenho, adaptabilidade dinâmica e eficiência operacional do sistema SWiPE. A partir da implementação e avaliação de quatro cenários distintos foi possível identificar configurações especificas capazes de atender com eficácia aos objetivos definidos.

Relativamente à melhoria do desempenho, pela observação dos resultados obtidos podemos concluir que os cenários 2 e 3, com a adição de novas instâncias dos componentes, apresentam os melhores resultados de desempenho. Este comportamento é evidenciado através da redução significativa dos tempos médios de resposta e do aumento do *throughput*, para além da redução de taxa de erro.

Ainda que não tenham sido efetuados testes ao sistema atual, comparando os resultados dos cenários 2 e 3 com o cenário 1, a melhoria do desempenho é notável. Assim, como no sistema atual apenas é executada uma instância de cada componente, é possível concluir que o desempenho do sistema SWiPE foi melhorado.

Relativamente à adaptabilidade dinâmica, o cenário 4 integrou a automação dos recursos do SWiPE através de um ambiente Kubernetes, o que demonstrou ter uma capacidade superior relativamente aos cenários anteriores e consequentemente ao sistema atual. Kubernetes, com o seu modelo de orquestração e pela definição de ficheiros de configuração, realiza a gestão automática da escala de *pods* de acordo com as variações de carga de trabalho do sistema. Durante os testes realizados, mesmo com variações significativas na carga de trabalho, o

ambiente Kubernetes conseguiu manter a taxa de erros nula. O uso deste ambiente assegura a escalabilidade do sistema de forma eficiente sem a intervenção manual.

Por outro lado, a automação proporcionada pelo Kubernetes no cenário 4 também resultou na melhoria significativa na eficiência operacional. A capacidade de adicionar e remover recursos, balanceamento de carga e recuperação de falhas promovidas pelo último cenário implementado, minimiza a necessidade de intervenções manuais.

6.2 Limitações e Trabalho Futuro

Apesar dos avanços alcançados, o estudo realizado identificou algumas limitações principalmente na implementação do último cenário. O desafio observado foi a latência na inicialização de novos *pods* em resposta a aumentos na carga de trabalho.

Em cenários de *ramp-up* curto, a diferença de tempo entre a inicialização dos *pods* e o momento em que estes se encontram ativos e disponíveis para as cargas impactou negativamente o desempenho do sistema. Esta limitação sugere a necessidade de melhorias na configuração da inicialização dos *pods* com o intuito de mitigar este atraso.

A introdução de Kubernetes e a configuração de réplicas de base de dados aumentou a complexidade geral do sistema, o que requer um nível de conhecimento mais técnico e pode introduzir novos pontos de falha.

Com base nas limitações mencionadas, é retido que se deve explorar mais a área da otimização da inicialização de *pods*. Ou seja, devem ser explorar técnicas avançadas capazes de reduzir a latência na inicialização de novos *pods*. Além disso também pode ser efetuado um estudo tendo em conta o histórico do uso da aplicação SWiPE de modo a antecipar os picos de carga e iniciar a criação de *pods* previamente.

Para além disso, também pode ser realizado o aprimoramento da distribuição de carga nas réplicas de base de dados. Poderá ser realizada uma investigação acerca da distribuição das cargas de leitura e escrita entre réplicas de MongoDB. Pode ser incluída a utilização de técnicas de *sharding* para distribuir a carga de uma forma mais granular.

Referências

Al-Dhuraibi, Y. *et al.* (2017) 'Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER', *IEEE International Conference on Cloud Computing, CLOUD*, 2017-June, pp. 472–479. doi: 10.1109/CLOUD.2017.67.

Al-Said Ahmad, A. and Andras, P. (2019) 'Scalability analysis comparisons of cloud-based software services', *Journal of Cloud Computing*. Journal of Cloud Computing, 8(1). doi: 10.1186/s13677-019-0134-y.

Angular (2024). Available at: https://angular.dev/overview.

Autoscaling Workloads | Kubernetes (2024). Available at: https://kubernetes.io/docs/concepts/workloads/autoscaling/#scaling-workloads-vertically.

Burns, B. *et al.* (2016) 'Borg, omega, and kubernetes', *Communications of the ACM*, 59(5), pp. 50–57. doi: 10.1145/2890784.

Buyya, R. *et al.* (2009) 'Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility', *Future Generation Computer Systems*. Elsevier B.V., 25(6), pp. 599–616. doi: 10.1016/j.future.2008.12.001.

Cassol, I. and Arévalo, G. (2018) 'A methodology to infer and refactor an object-oriented model from C applications', *Software - Practice and Experience*, 48(3), pp. 550–577. doi: 10.1002/spe.2549.

Docker (2024). Available at: https://www.docker.com/.

Dogani, J., Namvar, R. and Khunjush, F. (2023) 'Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey', *Computer Communications*, pp. 120–150. doi: 10.1016/j.comcom.2023.06.010.

Dragoni, N. *et al.* (2017) 'Microservices: Yesterday, today, and tomorrow', *Present and Ulterior Software Engineering*, (January), pp. 195–216. doi: 10.1007/978-3-319-67425-4_12.

Express.js (2024). Available at: https://expressjs.com/.

HAProxy vs NGINX: Performance & Benchmark — RapidSeedbox (2024). Available at: https://www.rapidseedbox.com/blog/haproxy-vs-nginx#05.

Herbst, N. *et al.* (2018) 'Quantifying cloud performance and dependability: Taxonomy, metric design, and emerging challenges', *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 3(4). doi: 10.1145/3236332.

Holmes, S. (2013) *Getting MEAN with Mongo, Express, Angular and Node*.

Horizontal Pod Autoscaling | Kubernetes (2024). Available at: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

Hussain, S. M., Bhatti, S. N. and Rasool, M. F. U. (2017) 'Legacy system and ways of its evolution', in *International Conference on Communication Technologies, ComTech 2017*, pp. 56–59. doi: 10.1109/COMTECH.2017.8065750.

Jovanovikj, I. *et al.* (2019) 'Modular Construction of Context-Specific Test Case Migration Methods', in *International Conference on Model-Driven Engineering and Software Development*, pp. 534–541. doi: 10.5220/0007690205340541.

Kashyap, S. and Singh, A. (2023) 'Prediction-based scheduling techniques for cloud data center's workload: a systematic review', *Cluster Computing*. Springer US, 26(5), pp. 3209–3235. doi: 10.1007/s10586-023-04024-8.

Kubernetes (2024). Available at: https://kubernetes.io/.

Lehrig, S., Eikerling, H. and Becker, S. (2017) 'Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics', in *QoSA 2017 - Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, Part of CompArch 2017*, pp. 83–92. doi: 10.1145/2737182.2737185.

Miraz, M. H., Ali, M. and Excell, P. S. (2021) 'Adaptive user interfaces and universal usability through plasticity of user interface design', *Computer Science Review*, p. 100363. doi: 10.1016/j.cosrev.2021.100363.

Mishra, S. K., Sahoo, B. and Parida, P. P. (2020) 'Load balancing in cloud computing: A big picture', *Journal of King Saud University - Computer and Information Sciences*, pp. 149–158. doi: 10.1016/j.jksuci.2018.01.003.
Moharir, M. *et al.* (2020) 'A Study and Comparison of Various Types of Load Balancers', *2020 5th IEEE International Conference on Recent Advances and Innovations in Engineering, ICRAIE 2020 - Proceeding*, 2020. doi: 10.1109/ICRAIE51050.2020.9358333.
MongoDB (2024). Available at: https://www.mongodb.com/pt-br/resources/basics/databases/document-databases.
Mulcahy, J. J. and Huang, S. (2017) 'Reengineering autonomic components in legacy software systems: A case study', *11th Annual IEEE International Systems Conference, SysCon 2017 - Proceedings*. IEEE, pp. 1–7. doi: 10.1109/SYSCON.2017.7934780.
Newman, S. (2020) *Monolith to Microservices : Evolutionary Patterns to Transform your Monolith*. Available at: http://oreilly.com/catalog/errata.csp?isbn=9781492047841.
Node.js (2024). Available at: https://nodejs.org/pt/.
Nygard, M. (2007) *What readers are saying about Release It !*, *North*. Available at: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Release+It!+Design+and+Deploy+Production-Ready+Software#0.
Page, M. J. *et al.* (2021) 'The PRISMA 2020 statement: An updated guideline for reporting systematic reviews', *The BMJ*, 372. doi: 10.1136/bmj.n71.
React (2024). Available at: https://react.dev/learn.
Richardson, C. (2019) *Microservices Patterns: With examples in Java*.
Da Silva, C. E., Yan De Lima Justino, · and Adachi, · Eiji (2022) 'SPReaD: service-oriented process for reengineering and DevOps Developing microservices for a Brazilian state department of taxation', *Service Oriented Computing and Applications*, 16, pp. 1–16. doi: 10.1007/s11761-021-00329-x.
da Silva Pinheiro, T. F. *et al.* (2023) 'A performance modeling framework for microservices-based cloud infrastructures', *Journal of Supercomputing*, 79(7), pp. 7762–7803. doi: 10.1007/s11227-022-04967-6.
Soni, D. and Kumar, N. (2022) 'Machine learning techniques in emerging cloud computing integrated paradigms: A survey and taxonomy', *Journal of Network and Computer Applications*. Elsevier Ltd, 205(March), p. 103419. doi: 10.1016/j.jnca.2022.103419.
Vaidya, O. S. and Kumar, S. (2006) 'Analytic hierarchy process: An overview of applications', *European Journal of Operational Research*, 169(1), pp. 1–29. doi: 10.1016/j.ejor.2004.04.028.
Zeng, R. *et al.* (2022) 'Performance optimization for cloud computing systems in the microservice era: state-of-the-art and research opportunities', *Frontiers of Computer Science*, 16(6). doi: 10.1007/s11704-020-0072-3.