INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Sistema de Conversação Autónoma para Marketing Multicanal

JOSÉ CARLOS MARQUES PAIVA
Junho de 2021

POLITÉCNICO
DO PORTO

Sistema de Conversação Autónoma para Marketing Multicanal

José Carlos Marques Paiva

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

Orientador: Prof. Paula Tavares
Supervisor: Prof. Ivo Pereira

Porto, 30 de junho de 2021

Dedicatória

A todos os envolvidos (orientadora, supervisor e amigos), namorada e pais, que neste mundo caótico, pandémico e confuso, dominado por um vírus interminável e intocável, me ajudaram a terminar mais uma etapa.

Um agradecimento também a todos os professores, que me avaliaram na primeira apresentação do documento, contribuindo construtivamente para o registo do processo de desenvolvimento do projeto.

Resumo

Atualmente, a tendência para utilização de plataformas *e-commerce* é substâncial, o que implica que haja mudanças consideráveis no comportamento dos consumidores [1]. A identificação do interesse dos utilizadores num determinado produto ou num segmento de produtos é fundamental, bem como a compreensão e análise estatística da procura dos mesmos. Neste contexto surge a E-goi, uma empresa já com alguns anos de experiência no setor de Marketing Digital, que oferece um conjunto de ferramentas diversificadas e úteis para a divulgação/reconhecimento de qualquer negócio quer seja de pequena dimensão - Pequena Média Empresa (PME) - ou de grande dimensão - empresas *Corporate*.

A contínua adesão de novos clientes à E-goi, bem como a manutenção dos que já trabalham com esta plataforma, alavanca uma necessidade enorme de ajuda. Para dar resposta a esta procura, existe um departamento intitulado por Serviço Ao Cliente (SAC) que, embora efetue um bom trabalho (conseguindo dar resposta a uma grande parte das dúvidas dos utilizadores), realiza tarefas muitas vezes repetitivas e rotineiras, que poderão ser executadas eficazmente através de, por exemplo, um sistema de respostas automático.

Tendo como objetivo manter este fluxo de conversa "humana" entre o cliente/empresa, foi decidido que uma boa resposta seria a criação de um Sistema de Conversação Autónomo, também conhecido como *chatbot*, de modo a substituir a "janela de pesquisa" já existente (denominada de Chatgoi). Este, deve ser capaz de, utilizando tecnologia NLP (*Natural Language Procesing*) identificar as questões dos clientes e responder com artigos de ajuda, que esclareçam as dúvidas dos mesmos. É igualmente importante acoplar todos os mecanismos de ajuda, já disponibilizados pela empresa, num único *widget* de suporte suficientemente versátil, desenvolvido na última versão de Angular (*framework* de apresentação da camada de dados) e num novo projeto à parte.

Mais do que um *chatbot*, este *widget* deve possibilitar a consulta de notificações enviadas pelo sistema, a visualização de *tickets* criados pelos utilizadores, um conjunto de ferramentas/serviços para prestação ao cliente, o conjunto de todas as páginas/vídeos/cursos para aprender a utilizar a ferramenta E-goi, uma comunidade para sugestão de ideias ou partilhas de tópicos e, caso se aplique, o acesso à informação do gestor da conta.

Palavras-chave: Chatbot, Marketing Digital, Assistente Virtual, Automação, Suporte

Abstract

Currently, the trend towards the use of e-commerce platforms is substantial, which implies considerable changes in consumer behaviors. It is essential to identify the large number of users that purchase a certain product, as well as to understand and analyze the demand for certain products, and this is where E-goi emerged. E-goi, a long-standing organization that operates in the Digital Marketing sector, offers a set of useful and diverse tools to publicize any business, whether small - *Small Medium Enterprise* (SME) - or large companies - Corporate.

The continuous entry of new customers on E-goi, as well as the maintenance of those who already work with this platform, raises a great need for help. To meet this demand, there is a department called SAC that, although doing a phenomenal job managing to answer most of the questions asked, performs often repetitive/routine tasks that could be automated through, for instance, an automatic response system.

To maintain this flow of a "human" customer/company conversation, it was assumed that a good answer would be to create an Autonomous Conversation System, also known as a Chatbot. This system should be capable of, using NLP (Natural Language Processing), identifying the questions made by the clients and answer accordingly with suggested articles. It is also important to join all the existant help mechanisms, already available and made by the company, in a centralized Help-Widget developed in the latest version of Angular and in a new distinguished project.

More than a Chatbot, this widget allows users to consult their notifications sent by the system, view all the tickets created, a set of tools to help with certain takss, all the pages to learn how to use E-goi, a community for suggesting ideas and, if applicable, access to the account manager information.

Keywords: Chatbot, Digital Marketing, Virtual Assistant, Automation, Support

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

AHP	*Analytic Hierarchy Process*.
AI	*Artificial Intelligence*.
API	*Application Programming Interface*.
BDD	*Behavior Driven Development*.
BLEU	*BiLingual Evaluation Understudy*.
CD	*Continuous Deployment*.
CEO	*Chief Executive Officer*.
CI	*Continuous Integration*.
CR	*Code Review*.
CRUD	*Create Read Update and Delete*.
CSS	*Cascading Style Sheets*.
CSV	*Comma-Separated Values*.
CTO	*Chief Technology Officer*.
FAQ	*Frequently Asked Questions*.
FFE	*Fuzzy Front End*.
FFNN	*Feed Forward Neural Network*.
FURPS	*Functionality Usability Reliability Performance Supportability*.
GB	*GigaByte*.
GUI	*Graphical User Interface*.
HITL	*Human In The Loop*.
HTML	*HyperText Markup Language*.
HTTP	*HyperText Transfer Protocol*.
JSON	*JavaScript Object Notation*.
LSTM	*Long Short Term Memory*.
LUIS	*Language Understanding Intelligent Service*.
MB	*MegaByte*.
MIT	*Massachusetts Institute of Technology*.
ML	*Machine Learning*.
MMS	*Multimedia Messaging Service*.
MR	*Merge Request*.
MVC	*Model View Controller*.

NCD	*New Concept Development*.
NLP	*Natural Language Processing*.
NLU	*Natural Language Understanding*.
ORM	*Object Relational Mapping*.
PHP	*Hypertext Preprocessor*.
PM	*Project Manager*.
PME	Pequena Média Empresa.
PO	*Project Owner*.
QA	*Quality Assurance*.
QEF	*Quality Evaluation Framework*.
QFD	*Quality Function Deployment*.
QnA	*Questions and Answers*.
REST	*Representational State Transfer*.
RNN	*Recurrent Neural Network*.
SAC	Serviço Ao Cliente.
SDK	*Software Development Kit*.
SEM	*Search Engine Marketing*.
SEO	*Search Engine Optimization*.
SME	*Small Medium Enterprise*.
SMS	*Short Message Service*.
SQL	*Structured Query Language*.
SWOT	*Strengths Weaknesses Opportunities Threats*.
TMDEI	Tese Mestrado do Departamento de Engenharia Informática.
UC	*Use Case*.
UI	*User Interface*.
UML	*Unified Modeling Language*.
UX	*User Experience*.
UXA	*User Experience and Assurance*.
WE	*Word Embedding*.
WIP	*Work In Progress*.
XSLX	*Excel*.

Capítulo 1

Introdução

Neste primeiro capítulo, é feita uma descrição do problema enquadrando-o no contexto em que se insere; são identificados, de forma resumida, todos os objetivos do projeto (principais e específicos ou menos relevantes); e é apresentada a metodologia/abordagem adotada para a resolução do problema.

No final, é detalhada a estrutura que foi utilizada para todo o documento, sintetizando cada um dos capítulos restantes, de forma resumida, contendo apenas os pontos essenciais.

1.1 Contexto

A presente dissertação aborda a realização de um projeto de desenvolvimento de software para a empresa E-goi, firma que atua no mercado de Marketing Digital. Surge num contexto académico, no âmbito da unidade curricular Tese Mestrado do Departamento de Engenharia Informática (TMDEI), do curso de Engenharia Informática ramo de Desenvolvimento de Software, para conclusão do grau académico de Mestrado.

Introduzindo o conceito de "Marketing Digital", este representa toda a divulgação de produtos ou serviços, através de dispositivos eletrónicos ou pela internet [2]. Pode ser também referido como *"Online Marketing"*, *"Internet Marketing"* ou *"Web Marketing"*. Ao contrário do que acontecia nos anos 1990 e 2000, em que o consumidor se deslocava à loja física, atualmente o consumidor utiliza cada vez mais dispositivos digitais para a realização de compras, daí o marketing digital se tornar fundamental para as empresas. Esta grande mudança no perfil dos clientes, faz com que as campanhas digitais sejam mais elaboradas e eficientes, sendo criadas com o objetivo de ir ao encontro das necessidades dos clientes e ao que estes procuram.

Existem muitas técnicas/táticas de promoção de campanhas, dependendo dos objetivos e potenciais ganhos de quem as promove. Segundo os avanços tecnológicos atuais, alguns exemplos são [2]: *Search Engine Optimization* (SEO), *Search Engine Marketing* (SEM), *Content Marketing*, *Affiliate Marketing*, *Marketing Automation*, *E-commerce Marketing*, *Inbound Marketing*, *Email Marketing*, entre muitas outras. Isto significa que, não são apenas dedicadas a canais de internet, como também a outros canais que fornecem meios digitais. A título de exemplo - os dispositivos móveis, que fornecem serviços como *Short Message Service* (SMS) e *Multimedia Messaging Service* (MMS).

A oportunidade, quer de criação de conteúdo direcionado a cada utilizador (de acordo com as suas necessidades), quer de criação de uma ferramenta que automatize todos os processos e estratégias de promoção digitais, possibilita o desenvolvimento de aplicações de software que ajam como intermediárias, facilitando a entregabilidade de mensagens publicitárias, menos

abrangentes e mais segmentadas. Neste contexto, nasce a E-goi - aplicação/plataforma intermediária entre o vendedor e o consumidor.

Apresentando a empresa, a E-goi é uma plataforma de automação de marketing multicanal, isto é, utiliza os canais de SMS, *SmartSMS*, *Email*, *Push*, *WebPush* e *Voz* como meio de comunicação/transmissão de publicidade através do envio de campanhas, *newsletters*, formulários, *landing pages*, entre muitas outras opções. Possibilita também a integração com redes sociais, blogs, Google *Analytics*[1], Google *Ads*[2], serviços de *e-commerce*; permite ainda a criação de *autobots* (fluxos de automação para otimização do processo de marketing), segmentação de listas de contactos (consoante o comportamento de cada cliente) e disponibilização de relatórios detalhados para cada campanha; responde também às necessidades multifacetadas dos clientes, através da disponibilização de diversos tipos de planos (*Corporate*, Pro, Base e *Social One*) e utilizando diversos idiomas, tanto no *website*, como nos artigos e cursos (em português, português do Brasil, espanhol e inglês).

1.2 Problema e Motivação

Segundo Miguel Gonçalves, *Chief Executive Officer* (CEO) da E-goi, uma das dificuldades existentes atualmente é a falta de auxílio a utilizadores que não conseguem realizar determinadas tarefas na plataforma, o que leva à criação de *tickets* de suporte ou abandono precoce da ferramenta (perda de potenciais clientes). Apesar de já existirem mecanismos de resposta a estas dificuldades, como uma base de conhecimento de pesquisa de artigos, um *blog* de ajuda e a disponibilidade de uma equipa dedicada a responder às dúvidas colocadas pelos utilizadores - Serviço Ao Cliente (SAC) - , não existe ainda uma capacidade de resposta suficientemente eficaz.

De um ponto de vista estatístico, apenas no ano de 2020, a plataforma teve um número de sessões superior a um milhão e meio, das quais foram criados mais de trezentos e dez mil pedidos de ajuda (*tickets*), cerca de quarenta mil precisaram de ajuda de suporte via *chat* (*live-chat* ou *Chatgoi* - mecanismo de pesquisa na base de conhecimento) - e mais de cinco mil chamadas foram direcionadas ao departamento do SAC (segundo estatísticas disponibilizadas pela organização).

O grande problema do mecanismo existente atualmente, é que serve apenas como uma pesquisa textual direcionada a uma *Application Programming Interface* (API), que devolve artigos consoante a frase colocada. Esta pesquisa não funciona de forma correta, pois muitas vezes não retorna nenhum artigo, ou não compreende aquilo que o utilizador questiona, levando à criação de novos *tickets*, diariamente, ou até mesmo ao abandono da plataforma. Esta falha conduz a que muitos utilizadores recorram erradamente à Comunidade para procurar respostas, uma vez que a "Comunidade" serve apenas como um fórum de partilha de sugestões e/ou ideias. Após experimentação e validação prática, verifica-se que o mecanismo de indexação da base de conhecimento para sugestão de artigos, encontra-se mal configurado não devolvendo os resultados mais úteis e não existe um processamento correto das frases enviadas (por exemplo, o retiro de carateres especiais).

Um problema secundário, está relacionado com o desenho da página, pois torna-se pouco claro o posicionamento dos mecanismos de ajuda. Por exemplo, o botão de ajuda descrito anteriormente, encontra-se no topo da página, o botão de contacto via *live-chat*, na lateral e

[1] https://analytics.google.com/
[2] https://ads.google.com/

o botão de contacto com a comunidade, situado num *link* no painel principal. Existem ainda os tutoriais da plataforma do Youtube e do Blog que não são devidamente apresentados, o que demonstra uma falta de organização dos elementos de ajuda.

Adicionalmente, verificou-se que é mantido um registo das pesquisas efetuadas, a utilidade das mesmas para o utilizador, as respostas apresentadas e se levou à abertura de um *ticket*. Este registo da utilidade é também um problema (não se verificando como fidedigno), porque não traduz o verdadeiro caso de pesquisa do utilizador e se os artigos devolvidos foram úteis ou não. A título de exemplo, se este efetuar uma pesquisa e não marcar a resposta como útil, é considerado que a sugestão de artigos não foi válida, o que pode não ser necessariamente verdade. De facto, o registo apresentado não mede eficazmente a informação relativa ao *Chatgoi*.

Existe ainda a falta de informação que é transmitida ao suporte da E-goi quando existe a criação de *tickets*, por parte dos clientes, uma vez que o SAC obtém apenas um conjunto de informações muito básicas do utilizador, que carecem de detalhes importantes sobre os mesmos.

A solução apresentada atualmente pela E-goi (*Chatgoi*) serve mais como uma barra de pesquisa de devolução de artigos, muito limitada, do que um mecanismo de suporte ao cliente eficaz e robusto, que devolva as respostas corretas e oriente o utilizador a resolver o problema em questão. Além disso, o conjunto de problemas apresentados aponta para uma necessidade premente de implementação de uma nova ferramenta e constitui também um grande interesse pessoal o desenvolvimento de um projeto nesta área.

1.3 Objetivos

Seguindo as boas práticas de engenharia informática, o intuito é desenvolver uma aplicação de suporte aos utilizadores da plataforma E-goi através da criação de um *widget* de ajuda, com um novo *chatbot* integrado (mecanismo de conversação).

Como tal, destacam-se como objetivos principais:

- investigação do sistema/solução atual, das dificuldades sentidas por quem utiliza a plataforma ou quer começar a utilizar ou sente dificuldades a realizar tarefas;
- iniciação de desenvolvimento de um projeto novo que apresente uma camada de *User Interface* (UI), com uma ajuda centralizada que tenha disponível todos os mecanismos de ajuda por parte da E-goi;
- integração deste novo projeto com os já existentes da E-goi, mantendo toda a lógica que existe atualmente: registo das questões colocadas, criação/consulta de tickets, ligação ao projeto da Comunidade (fórum de troca e partilha de ideias);
- integração com um mecanismo de conversação ou assistente virtual (*chatbot*), com recurso a uma ferramenta externa fácil de manter e que dê resposta às dúvidas dos clientes, mantendo um fluxo de conversa o mais humano possível;
- aproveitamento dos mecanismos de *Natural Language Processing* (NLP) da ferramenta para treino de um modelo utilizando os registos guardados até hoje (considera-se que este seja um bom ponto de partida, para o desenvolvimento do *bot*);
- expansão das fontes de procura de artigos, para além da base de conhecimento atual, como artigos do Blog e do Youtube.

Na plataforma E-goi, não existe forma de visualização das notificações enviadas pelo sistema (operações em curso, atualizações semanais, entre outras), porque estas surgem como um *popup* de curta duração, não podendo ser consultadas mais tarde. Por vezes, existe também uma necessidade dos clientes de serviços à medida. Neste âmbito, aparecem os objetivos secundários específicos, não menos importantes, que distinguem o *widget* de um simples *chatbot*, tais como:

- disponibilizar a consulta de notificações (com meio de pesquisa incorporado);
- possibilitar a filtragem de *tickets* por estado (aberto, fechado, apagado, etc);
- mostrar o histórico de conversas recentes com o *chatbot* (pelo menos as duas últimas);
- incluir serviços de ajuda extra ao cliente, com taxa associada (por exemplo, reposição de listas de contactos apagadas acidentalmente);
- oferecer um separador de aprendizagem de fácil e rápido acesso (acesso ao blog, artigos disponíveis, vídeos de consulta, etc);
- melhoria do projeto da Comunidade (visualmente e funcionalmente).

Por último, a integração da Comunidade, projeto existente já de grande escala e também um dos objetivos secundários, deve permitir que utilizadores sugiram alterações, criem *posts*, partilhem, comentem, votem e visualizem as ideias que foram implementadas, rejeitadas ou estão sob avaliação. Por motivos de *spam*, deve existir um moderador capaz de assegurar que todos os tópicos e comentários criados seguem as normas/código de conduta da E-goi.

1.4 Abordagem

Estabelecido o problema e quais os objetivos a cumprir, é fundamental esclarecer como serão abordados, considerando o contexto em que se insere o projeto, para poder proceder-se ao desenvolvimento do mesmo, de modo a apresentar uma solução.

A E-goi utiliza uma abordagem *Agile* [3] (não existindo o conceito de *sprints*[3]), apoiada por *weekly meetings*, em que cada um dos elementos refere o que realizou nessa semana, o que irá trabalhar durante a mesma e quais as dúvidas ou questões existentes (ajuda necessária). Utiliza também o Jira[4], ferramenta de gestão de projetos ágeis, para criação de *issues* relacionadas, neste caso, com um épico que é o *widget* de ajuda.

Existem 7 fases de desenvolvimento (passos para completar uma *issue*), organizadas num quadro Kanban, que serão abordadas em mais detalhe em 6.1, nomeadamente: *Backlog* > *For Dev* > *In Dev* > *For QA* > *In QA* > *In CR* > *Release*. As únicas fases que os *developers* devem gerir são, especificamente, a fase *For Dev* (para desenvolver), *In Dev* (em desenvolvimento) e *For QA* (para *Quality Assurance* (QA)), uma vez que o *Backlog* é gerido pelo *Project Manager* (PM) ou pelo *Project Owner* (PO) e as restantes pela equipa de QA e pelo *Lead Developer*.

Quanto à abordagem dos objetivos, após levantados os requisitos no capítulo 4, estes foram colocados no estado *For Dev*, com a respetiva estimativa de previsão de realização. Assim que é completado cada um dos objetivos, deve ser colocado na *issue* o tempo gasto (até

[3] Um sprint é um período curto e limitado em que uma equipa Scrum trabalha para completar uma tarefa.
[4] https://www.atlassian.com/software/jira

completar a tarefa), qual o trabalho realizado (*commits*, caso se aplique) e como deve ser testado. Assim que o objetivo atinge a fase final de *Release*, é colocado em produção.

1.5 Metodologia

Seguir-se-á como referência ao relatório, a metodologia descrita em [4]. Considera-se que o tema "*Design Science Methodology for Information Systems and Software Engineering*", se enquadra na metodologia que se pretende seguir, referindo diversos capítulos, que serão correlacionados com os capítulos apresentados ao longo da dissertação.

A ciência do design, no contexto de sistemas de engenharia de software, consiste em duas atividades fundamentais a saber: o desenho de um artefacto que constitua uma melhoria para os *stakeholders* e a investigação empírica do desempenho desses artefactos, num contexto particular. O artefacto que irá ser desenvolvido, inclui o desenho de uma aplicação centralizada de ajuda, de acordo com os objetivos definidos anteriormente.

Seguindo a estrutura definida na apresentação do autor da metodologia, em "*Outline of a thesis*" [5], no presente capítulo clarificam-se quais os problemas atuais e de que forma poderão ser resolvidos, de acordo com as necessidades dos stakeholders, dando resposta ao capítulo 1 - Motivação.

No capítulo 2, atende-se ao estado da arte de soluções/tecnologias semelhantes existentes no mercado, correspondendo ao capítulo 5 "*Survey of current solutions*". Aborda-se ainda uma análise científica na área dos *chatbots*, a sua forma de funcionamento e também uma comparação entre tecnologias atuais para o desenvolvimento dos mesmos.

Após estudar as soluções e tecnologias existentes, é feita uma análise de valor ao projeto, o que permite perceber a necessidade do mesmo, comprovando com mais exatidão a relevância do problema, dando reposta à parte 3 "*Problem Investigation*". Com recurso a vários modelos, alguns de análise estatística, avalia-se a importância da solução e a proposta realizada.

Após a análise anterior, avança-se para a estruturação dos requisitos em 4, correspondendo ao capítulo 4 "*Requirements for a solution*", listando as técnicas de obtenção dos mesmos e a definição de cada um de forma organizada (através de casos de uso).

Ainda no capítulo 4 e no capítulo seguinte, aborda-se a "*Chapter 6: My solution proposal*", apresentando, através de diagramas UML, uma análise e o desenho da solução. Faz também parte deste guia, o capítulo 6 - Implementação - que apresenta a solução em concreto e todo o processo de desenvolvimento da mesma, quais as ferramentas e a metodologia utilizados.

O capítulo 7 define a camada de testes, de forma a garantir que a solução satisfaz os requisitos e quais os efeitos que a mesma teve (representando o capítulo 7 e o 8 do guia da dissertação).

Para finalizar, "*Summary, answers to research questions, discussion, future work*" é apresentado na conclusão no ponto 8, resumindo o projeto e a solução concebidos e quais os *outcomes* obtidos, bem como o trabalho futuro a ser desenvolvido.

1.6 Estrutura do Documento

Com a finalidade de descrever o documento de forma sucinta, são apresentados os pontos fundamentais - capítulos - objetivamente, com uma breve descrição dos subcapítulos. Posto isto, destacam-se os capítulos na seguinte ordem:

1 - Introdução: descreve o contexto em que se insere o problema a solucionar, bem como os objetivos planeados para resolvê-lo. De seguida, é apresentada a metodologia que segue diretrizes e normas de software e a estrutura do documento;

2 - Estado da Arte: após realizada a introdução do problema e como abordá-lo, são analisados quais os conhecimentos existentes sobre o tema, que outras soluções semelhantes ou equivalentes existem no mercado e quais as tecnologias disponíveis para o desenvolvimento do projeto;

3 - Análise de Valor: avalia o valor do projeto a ser implementado, tendo em vista um produto que será desenvolvido tanto para o cliente, como para a organização, trazendo valor para ambas as partes. Além disso, aborda os tópicos de identificação e análise de oportunidades, geração e identificação de ideias e vários modelos como o *Analytic Hierarchy Process* (AHP), proposta de valor e *Quality Function Deployment* (QFD);

4 - Análise da solução: retrata os métodos seguidos para extração dos requisitos, os casos de uso estabelecidos a serem implementados e os requisitos não funcionais;

5 - Desenho da solução apresenta os artefactos de *software* propostos para resolução do problema e realização dos objetivos, através de diagramas *Unified Modeling Language* (UML) que apresentam diferentes formas de visualização do sistema;

6 - Implementação: aqui destacam-se todos os detalhes técnicos relacionados com o desenvolvimento de cada fase do projeto, assim como decisões tomadas quanto ao sistema e algoritmos utilizados;

7 - Experimentação e Avaliação: é feita uma avaliação da aplicação, de acordo com os requisitos iniciais e de forma a validar se os objetivos foram cumpridos, avaliando-os através de diversos testes;

8 - Conclusão: por último, apresentam-se as conclusões retiradas do processo percorrido até à entrega da solução final, quais as aprendizagens, problemas, obstáculos e trabalho futuro que poderá ser necessário empreender.

Capítulo 2

Estado da Arte

Este capítulo dá resposta à questão da necessidade do desenvolvimento deste projeto, bem como comparar a solução existente na plataforma, com outras soluções existentes no mercado. Além disso, serão mostradas quais as tecnologias possíveis para desenvolvimento deste projeto. Por fim, também serão apresentadas as vantagens, desvantagens e funcionalidades encontradas, respetivamente para cada caso de estudo.

Para que as figuras de demonstração dos exemplos apresentados não sejam lineares, isto é, não sejam sistematicamente apenas sobre conversas com os *chatbots* respetivos, outras figuras de exposição do funcionamento dos mesmos serão mostradas em alternativa.

2.1 Definições contextuais

Introduzindo os sub-capítulos seguintes, considerou-se uma boa abordagem apresentar algumas definições importantes e fundamentais, que devem ser previamente explicadas, para se perceber o contexto em que os *chatbots* se inserem e o seu modo de funcionamento. Os termos apresentados a seguir, serão mencionados diversas vezes ao longo deste documento e poderão ser consultados aqui em futuras referências (caso seja necessário), bem como no glossário poderão ser consultados os respetivos acrónimos.

Entity - variável que complementa o *Intent* (ex: "Quero marcar um voo às 10 da manhã" - "Data" seria uma entidade, pois representa uma variável que pode ser extraída do *Intent*) [6];

IFrame - *tag HyperText Markup Language* (HTML) que descreve o domínio de uma página HTML (normalmente externa), diferente do domínio da página atual [7]. Exemplo:

```
<iframe src="https://www.youtube.com/" title="Iframe do youtube"></iframe>
```

Intent - representa o que um utilizador quer realizar (ex: "Pretendo marcar um voo") [6];

Live-chat - conversa com uma pessoa "real", normalmente representado através de um agente de suporte, no contexto da plataforma;

NLU (AI) - através de inteligência artificial, o módulo *Natural Language Understanding* (NLU) processa as mensagens introduzidas pelo utilizador, classificando os *Intents* da frase, extraíndo as Entidades e realizando outras operações [8] (sujeito à interpretação de cada ferramenta a ser utilizada);

QnA (AI) - ao contrário do módulo NLU, o módulo *Questions and Answers* (QnA) procura identificar a pergunta do utilizador e retornar automaticamente uma resposta de um conjunto pré-definido de respostas [8];

Human In The Loop (HITL) - o módulo HITL descreve a capacidade de o suporte poder intervir na conversa entre o utilizador e o *bot*, interrompendo o fluxo entre ambos;

Quick action/reply - corresponde, normalmente, a botões com respostas pré-definidas pelo *chatbot* para que, rapidamente o utilizador possa escolher uma, em vez de escrever o que pretende. Por exemplo, quando são colocadas questões que visam obter uma resposta de "Sim" ou "Não" [9];

Software Development Kit (SDK) - conjunto de utilitários para desenvolvimento de aplicações de *software*, através de ambientes específicos. Poderá ter também: um compilador, um *debugger* e, por vezes, uma *framework* de *software* [10];

Webhook - subscrição a notificações/eventos *HyperText Transfer Protocol* (HTTP) em *real-time*, que é *triggered* ou ativada, quando acontecem mudanças a objetos específicos [11]. Isto faz com que não seja necessária a execução de *querys* ou pedidos constantes de informação, a uma *Application Programming Interface* (API);

Webview - traduz-se essencialmente numa página *web*, mas que é aberta dentro de um contexto específico, permitindo ter um maior controlo sobre a interface e funciona, especialmente bem, com dispositivos móveis [12].

2.2 Análise Científica

Com o intuito de explorar e alargar o conhecimento na área de *chatbots*, procedeu-se à leitura de vários artigos, que irão ser destacados ao longo deste subcapítulo. Descreve-se, numa pequena introdução, a importância do facto dos *chatbots* (atualmente), seguido da distinção das fases de desenvolvimento de um *chatbot*. O conhecimento obtido através destes artigos, permite assim ter um melhor critério de decisão comparativa no capítulo 2.4, uma vez também já esclarecidos alguns dos termos em 2.1.

2.2.1 Contexto dos Chatbots

Nesta nova era de tecnologia, os *chatbots* são o próximo grande domínio em termos de serviços de conversação [13]. Um *chatbot* representa uma pessoa virtual, com quem, efetivamente, qualquer ser humano possa conversar de forma verbal (como é o caso das tecnologias Amazon Alexa, Google Assistant ou Apple Siri) e/ou textual (como o caso dos exemplos que serão descritos nas secções 2.3 e 2.4). É crucial entender o modo como estas tecnologias operam e a sua necessidade para as organizações, referindo-se a análise atual e procedente apenas a *chatbots* textuais.

Um estudo de 2016 demonstra que, por causa de fracos serviços de atendimento ao cliente, empresas americanas perdem mais de 62 biliões de dólares americanos anualmente [14]. O facto é que, a criação de *chatbots* é cada vez mais comum e aplicada, como descrito pelo Facebook na conferência anual de 2018, ao ter anunciado que a sua plataforma foi utilizada para criação de mais de 300,000 *bots* [13]. Outro estudo publicado por uma empresa inglesa, Juniper, prevê que a criação de chatbots irá ajudar as firmas de negócio a poupar mais de 8 biliões de dólares americanos anuais, a partir de 2022 [15].

Assim sendo, existem duas fases de desenvolvimento de um *chatbot*: o *design* arquitetural (geração do *chatbot*) e a parte de implementação (integração do *chatbot*). A primeira pode ainda ser categorizada em quatro aspetos fundamentais, que são: domínio de conhecimento (aberto ou fechado), geração de respostas (pesquisa e retorno ou generativa),

processamento de texto (embutido em vetores ou alfabeto de latim) e *machine learning* (geralmente, redes neuronais) [16].

2.2.2 Design Arquitetural

Os *chatbots* podem ser aplicados em diversas áreas, desde o serviço ao cliente, assistência pessoal, simples *Frequently Asked Questions* (FAQ), até *bots* de propósito geral. Isto significa que a conceção e treino do modelo, de cada *chatbot* deve ser dependente da área em que atua e a que está a ser aplicado. Como descrito em 2.2.1, o domínio de conhecimento dos *bots* pode ser aberto, o que significa que deve abranger um conhecimento geral como notícias, entretenimento, desporto, etc. e também conhecimento básico humano. Caso o conhecimento seja específico a uma determinada área (psicologia, atendimento ao cliente, etc.), o domínio pode ser classificado como fechado.

Os métodos de geração/produção de respostas podem ser de pesquisa e retorno - seleção do melhor *output* para uma pequena lista de candidatos -, generativos - *output* flexível, baseado numa sequência de *inputs* e classificadores de treino - ou híbridos (incluem ambos). O melhor exemplo, para muitas literaturas e sistemas de geração de respostas híbridas é o Seq2Seq [17]. A arquitetura Seq2Seq, como descrito em [16], é baseada numa *framework* de codificação e descodificação, onde o codificador recebe um *input* e o descodificador produz um *output* relevante para o *input* (por exemplo, traduzir uma linguagem para outra). O sistema de codificação do "*Sequence-to-Sequence*" recebe uma frase como uma sequência de palavras, analisando cada palavra (uma por uma), enquanto gera a sua representação (uma palavra de cada vez) e, finalmente, gera a representação do *input* todo, para ser enviado para o módulo de descodificação. Depois, este módulo, decifra a representação do *input* e gera um *output* (uma palavra de cada vez), até que todas as representações sejam decifradas. Com esta capacidade, o sistema Seq2Seq apresenta bons resultados, pois foi implementado para:

> "*[...] Seq2Seq in ranking most similar output/response towards the input (retrieval) and also generate a totally flexible output in word or character based that is sequentially reflected the input (generative).*" ([16] - Lokman e Ameedeen, 2018)

Quanto ao processamento de texto, no que diz respeito ao domínio de *chatbots*, o *Word Embedding* (WE) é utilizado na maior parte dos sistemas que utilizam *Machine Learning* (ML) - para treino ou implementação *real-time*. O WE ou representação por vetores, consegue detetar uma relação semântica entre as palavras de um vocabulário específico (através de uma distribuição por hipóteses). Como a representação das palavras é traduzida em números reais, cálculos estatísticos e aritméticos são altamente possíveis, o que torna a sua utilização em modelos estatísticos de ML viável.

Para concluir, no contexto de arquiteturas modernas de *chatbots*, ML pode ser visto como uma tecnologia central, que é utilizada desde a preparação do *input* e processamento, até ao processamento de *output* e produção de resultados. O objetivo de ML nos *chatbots* é, em grande parte, gerar WE (utilizado no módulo de processamento de texto) na fase de preparação dos *inputs*, devido à grande utilização de métodos estatísticos em ML. Na fase de processamento dos *inputs*, o *Long Short Term Memory* (LSTM) é o modelo de ML mais utilizado em redes neuronais, que ao contrário de uma simples implementação em que a informação se move numa única direção (por exemplo, *Feed Forward Neural*

Network (FFNN)), durante um determinado intervalo de tempo, existe um uso recorrente da *Recurrent Neural Network* (RNN) em que o fluxo não é linear (ver a figura 2.1) [13].

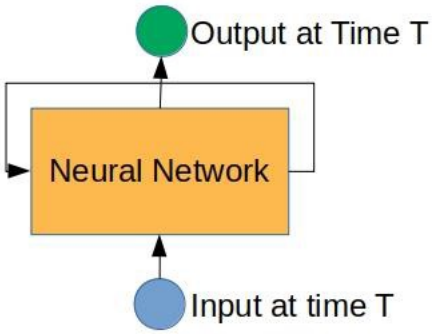


Figura 2.1: LSTM - Modelo RNN [13]

O modelo LSTM é utilizado para gerar uma representação de uma sequência de *inputs* (baseada na representação de cada palavra), normalmente utilizando o modelo de codificação Seq2Seq. O codificador é utilizado na fase de processamento dos *inputs* e, consequentemente, o descodificador é utilizado na fase de processamento dos *outputs* e na fase de produção. Em suma, qualquer *chatbot* com uma implementação Seq2Seq utiliza RNN para os módulos de processamento de valores recebidos e resultados obtidos.

2.2.3 Implementação/Conjuntos de Dados

Um modelo de ML é tão bom/útil quanto os seus conjuntos de dados, também conhecidos como *datasets*. Para classificadores de ML, um maior treino representa melhores parâmetros classificativos. Já no caso de WE, um dicionário pequeno resultará em representações escassas das palavras e menos vocabulário, afetando a representação das frases, o cálculo de *ranking* e a flexibilidade de geração do módulo de respostas.

Os *datasets* podem ter muitos tipos e estruturas. Como exemplo, no caso das cinco plataformas estudadas em [13], uma delas, SuperAgent (*closed-domain* - em 2.2.2), utilizou um conjunto de dados retirado diretamente do próprio *website* de *e-commerce* (informação dos produtos, questões e respostas, *reviews* dos clientes) e além disso, utilizou também 43 milhões de pares "pergunta-resposta" do Twitter[1] para treinar o módulo de "Conversa Fiada". Desses 43 milhões, apenas 5 milhões das perguntas mais frequentes não duplicadas (com menos de 15 palavras) foram utilizadas para treinar o modelo.

2.2.4 Métodos de Avaliação

Quanto aos métodos de avaliação dos sistemas de *chatbots*, poderão existir: avaliação humana, testes A/B, *BiLingual Evaluation Understudy* (BLEU) e análise de usabilidade. Estes métodos permitem avaliar a qualidade das respostas de um *chatbot* relativamente à percepção humana, tanto semanticamente, como sintaticamente. Relativamente ao primeiro método, a avaliação humana, é altamente costumizável e, à partida, definido através de escalas, por exemplo de 1-3 (bom, médio, muito bom). Os testes A/B representam testes entre duas variâncias do mesmo sistema. A avaliação BLEU é um método que pondera a qualidade do texto que foi traduzido por uma máquina [18]. E, por último, a análise de

[1] https://twitter.com/

usabilidade que se traduz em justificar o porquê de estar feito de uma determinada forma. A título de exemplo, a mesma ferramenta descrita no parágrafo anterior - SuperAgent -, utiliza este tipo de método para justificar a sua utilidade como extensão dos *web browsers* mais utilizados, como melhoria da experiência de compras *online* dos utilizadores.

2.3 Soluções e Abordagens Existentes

A primeira abordagem para resolução dos objetivos em 1.3, é observar o que existe no mercado, quais são as soluções oferecidas por outras empresas e organizações, e de que forma respondem aos clientes quando surgem dúvidas ou têm necessidade de auxílio na resolução dos seus problemas.

Posto isto, o primeiro passo foi estudar a solução já existente na E-goi, entender qual o estado em que se encontra, quais as falhas atuais, funcionalidades que poderão ser reutilizadas e a forma como opera os mecanismos de ajuda. O passo seguinte é então explorar outras ferramentas e seguir a mesma abordagem. Como tal, procedeu-se à analise destas plataformas através de criação de conta e interação com o *chatbot*, à leitura da documentação associada e à investigação de avaliações fornecidas por *websites* comparativos como é o caso da Capterra[2].

2.3.1 E-goi Chatbot (Chatgoi)

Atualmente, a E-goi suporta já um *live-chat* que permite comunicar com os agentes que integram a equipa de suporte. Analisando a figura 2.2, que representa esse mesmo mecanismo de conversa, é possível retirar diversas informações, tais como:

- opção de receber uma cópia da conversa para posterior consulta;
- pode ser terminado e recomeçado o *chat*, no mesmo estado, a qualquer momento, clicando no botão de "Fim do *chat*" ou clicando no botão de recomeçar (mecanismo de *start/stop conversation*);
- no fim da conversa, o sistema pede uma avaliação do serviço prestado pelo agente (sistema de avaliação/*rating*);
- regista, para cada mensagem, a hora do dia em que foi enviada;
- permite o anexo de ficheiros para uma melhor explicação do problema.

Para poder iniciar uma conversa com o suporte, é obrigatório o utilizador estar autenticado e, apesar de estar autenticado, precisa de dizer qual o seu *email* e o nome, a utilizar. Isto significa que, a E-goi não aproveita informações do utilizador (que já tem), para identificar com quem está a falar. Além disso, o *chat* não funciona fora da plataforma da E-goi, ou seja, é necessário o utilizador efetuar login para que possa usufruir do *chat*.

[2]https://www.capterra.pt/

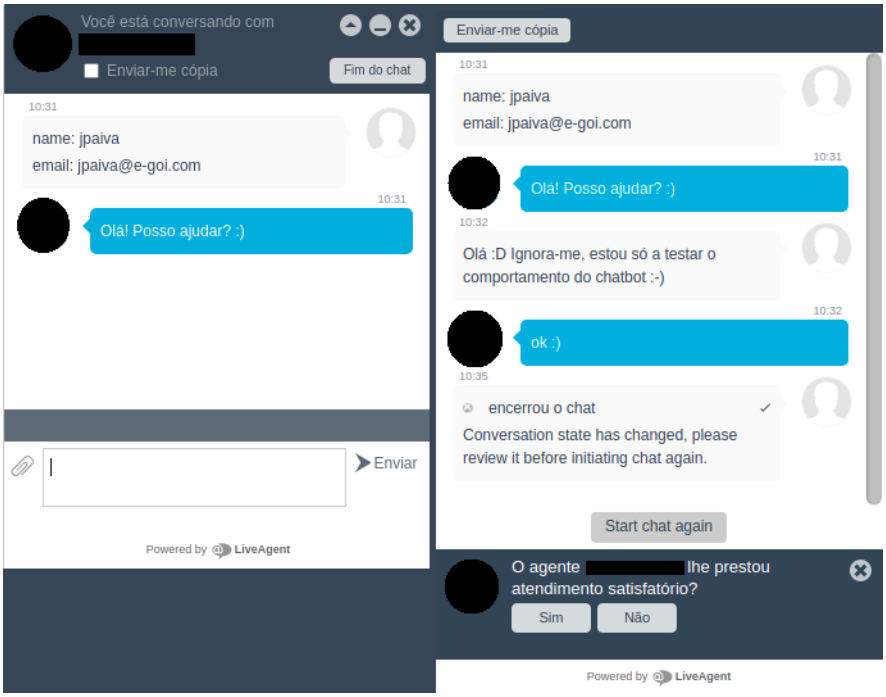


Figura 2.2: Exemplo do *live-chat* implementado na E-goi

Assim que o utilizador providencia essas informações, o sistema procura quais os agentes que estão disponíveis e atribui um ao utilizador que tenta iniciar o *chat*. De notar que, o *chat* apenas permite iniciar uma conversa com pessoas reais.

É importante referir que esta integração é feita através da plataforma LiveAgent[3] - serviço *web* de suporte multi canal (*chat*, voz) de atendimento ao cliente - por comunicação via API JavaScript, isto é, todo o design do botão e janela de conversa são desenvolvidos neste *website* e toda a lógica desta interação utilizadores/agentes é realizada através de um *script* ou *plugin* que é colocado na plataforma da E-goi. Este *script* interage diretamente com o LiveAgent, não tendo qualquer informação (do lado da E-goi) sobre como são processados os dados, qual a lógica de envio ou receção de mensagens, o *rating*, entre outras informações.

2.3.2 MongoDB Chatbot

Quanto a soluções fora do contexto E-goi, uma outra é a apresentada pelo MongoDB Cloud[4] - base de dados baseada na nuvem - que utiliza um serviço externo intitulado Intercom[5], empresa especializada em desenvolvimento de *chatbots*, para ajuda no esclarecimento de dúvidas por parte dos clientes.

A exibição do *chatbot* é realizada através de um *iframe*, isto é, a MongoDB injeta um *script* no código que permite aos utilizadores visualizar o *chatbot* (que aparenta ser deles), mas que na verdade executa código da Intercom. Esta prática é muito comum em diversos *sites* que, recorrem a serviços *third-party*/externos, normalmente bem desenvolvidos e especializados num processo ou produto e em mantê-lo.

[3] https://www.liveagent.com/
[4] https://www.mongodb.com/cloud
[5] https://www.intercom.com/

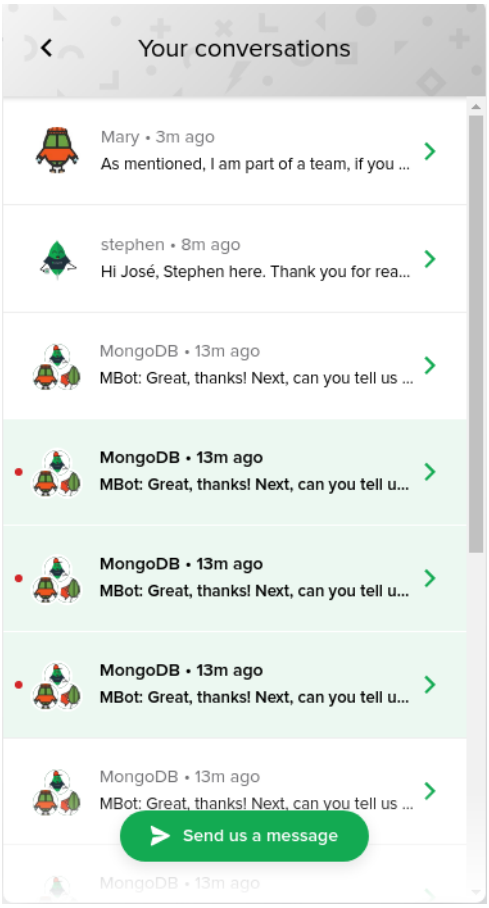


Figura 2.3: MongoDB - registo de todas as conversas

De forma a definir todos os pontos identificados para a aplicação de conversação analisada, a seguir encontram-se descritas quais as vantagens, desvantagens e funcionalidades consideradas como extra:

Vantagens

- funciona tanto para utilizadores autenticados como não autenticados, detetando automaticamente os dados de utilizadores autenticados;
- regista todas as conversas anteriores (permitindo continuar uma conversa inacabada) - histórico de conversas 2.3;
- permite efetuar o *download* da conversa (em formato de texto);
- suporta datas de envio das mensagens, o conceito de "visualização" (tal como uma conversa, por exemplo por Messenger[6] ou Whatsapp[7]) e conversas não abertas mostram uma notificação vermelha (representa uma mensagem não aberta);
- sugere *quick replies* à partida (Olá o que o traz aqui hoje? - "Eu preciso de ajuda com o produto; Quero falar com o departamento de vendas; Outro");

[6]https://www.messenger.com/
[7]https://web.whatsapp.com/

- envio de *email* com a conversa, quando é feita a troca de conversa com o *bot*, para conversa com o Serviço Ao Cliente (SAC) (esta troca é quase impercetível);
- permite ao utilizador, no fim da conversa, dar um *rating*, com *emojis* (ver 2.4);
- permite fazer *quick replies* fora do *chatbot* (apresentado na figura 2.5);

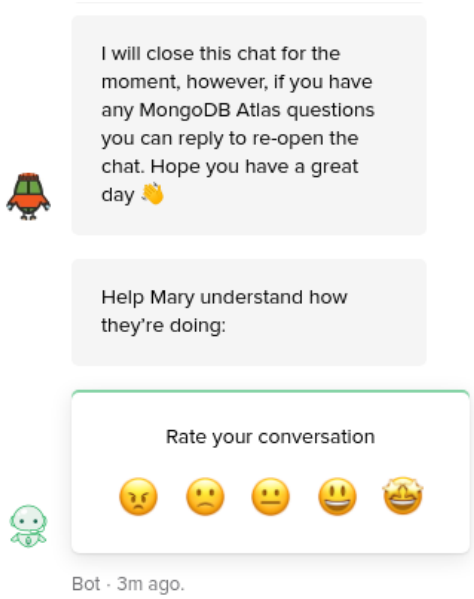


Figura 2.4: MongoDB - avaliação da conversa com o suporte

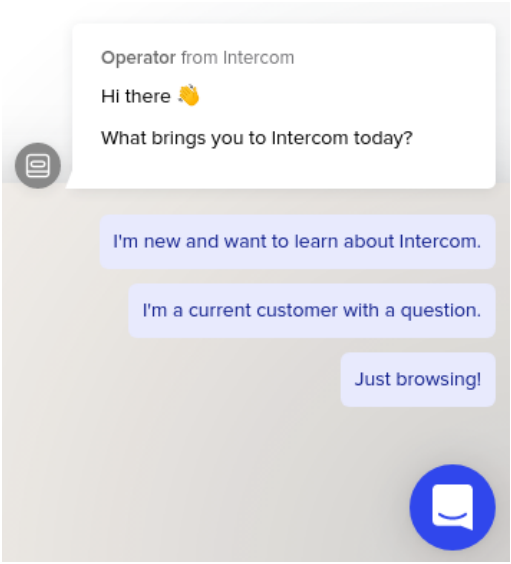


Figura 2.5: Intercom - sistema de notificações fora do chatbot

Desvantagens

- não permite arrastar o *chatbot*, isto é, está bloqueado/preso no canto do ecrã;
- quando é aberta uma conversa, são mostradas as primeiras mensagens, em vez das últimas;
- utiliza um serviço *3rd-party*, como serviço de *chatbot*. Este serviço é pago (a partir de 39$/mês) e é integrado no site através de um *iframe* (isto pode trazer problemas de privacidade e segurança, uma vez que são enviados os dados para a API de um serviço externo).

Funcionalidades Extra

- permite enviar *gifs*, *emojis* e anexos (não limita o envio de imagens);
- equipa do SAC dedicada ao *chatbot* (pelo menos 3 pessoas);
- o utilizador pode escolher terminar a conversa, a qualquer momento, com o suporte *live-chat*;
- contacto com alguém do suporte apenas é possível com algum tipo de autenticação (*email*);
- permite pesquisar artigos através de uma caixa de pesquisa integrada no *bot* (pesquisa nos artigos FAQ - separada da conversa com o *bot*);

- esconde a parte de envio de resposta, enquanto procura as respostas do servidor (para não permitir ao utilizador enviar mais mensagens, enquanto não obtiver resposta).

2.3.3 Zoom Chatbot

Sendo o Zoom[8] uma das grandes empresas de serviços, não só na *cloud*, como também aplicacional, para vídeo/aúdio conferências e *chat*, considerou-se um bom exemplo, uma vez que também tem embutido um serviço de *chatbot* no próprio *website*.

Este serviço de conversação, oferecido pela plataforma, utiliza o Ada[9], motor de automação de respostas aos clientes, com suporte de agentes integrado semelhante ao que apresenta o LiveAgent da E-goi, para resolução de dúvidas e/ou problemas, reduzindo o número de criação de *tickets* e o tempo de resolução de problemas. Está, tal como o serviço da Intercom em 2.3.2, integrado através de um *iframe* no *website* do Zoom.

O Ada representa uma plataforma "*No-Code*", funcionando através de uma comunicação por API, disponível para ser integrada em qualquer plataforma.

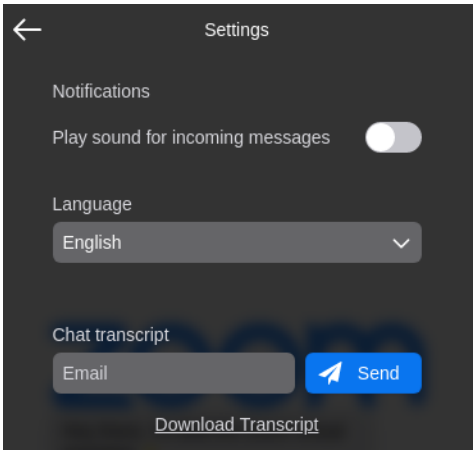


Figura 2.6: Zoom - definições disponíveis de configuração

Como estudo do funcionamento do Ada, foi seguido o mesmo processo de interação com o *chatbot* tal como o da Intercom (MongoDB), descrevendo-se de seguida quais as vantagens, desvantagens e funcionalidades extra encontradas:

Vantagens

- botão de configuração de definições 2.6, permitindo ativar/desativar o som das notificações, mudar a linguagem do *chat* (suporte a diversas línguas) e permite também efetuar o *download* da conversa;
- disponibiliza um leque de ações rápidas à partida 2.7 (ex: Como posso ajudar? - "Começar a utilizar o Zoom; Como me junto a uma reunião?; Câmara não funciona", etc);

[8] https://zoom.us/pt-pt/meetings.html
[9] https://info.ada.cx/zoom-automation-engine

- tem um sistema semelhante aos gostos nas mensagens de, por exemplo, o Messenger ou Microsoft Teams[10] marcando a resposta como "Sim" ou "Não" (só em determinados casos, como por exemplo: Foi útil a resposta?);
- deteção de *Artificial Intelligence* (AI), à medida que o utilizador insere palavras, com sugestões automáticas através do reconhecimento de texto 2.8;
- fila de espera para falar com agentes (por exemplo: "Estão 7 pessoas à sua frente, por favor aguarde...").

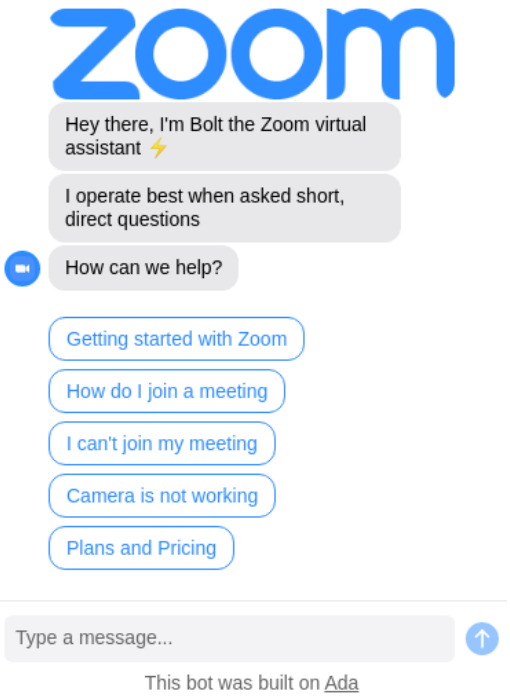


Figura 2.7: Zoom - ponto de entrada numa conversa

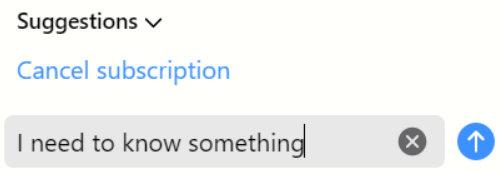


Figura 2.8: Zoom - sugestões à medida que o utilizador insere palavras

Desvantagens

- serviço pago (não disponível nos planos do *website*);
- não permite arrastar o ícone do *chatbot*, uma vez que está fixo à lateral, para utilizadores não autenticados, e numa janela de abertura, para autenticados;
- não mantém um histórico de conversas, apenas mantém a conversa atual;
- contacto com alguém do suporte, apenas possível com algum tipo de autenticação (*email* e *key* especial).

Funcionalidades Extra

- o *display* das mensagens é feito um a um, ou seja, não são enviadas todas de uma vez (o que transmite uma sensação de troca de mensagens com uma pessoa, *user friendly*);

[10] https://www.microsoft.com/en/microsoft-teams/group-chat-software

- quando o utilizador pretende ajuda, por exemplo, para se juntar a uma reunião são recebidas mensagens com um guia *step-by-step* 2.9, que, com o auxílio de imagens, mostra onde o utilizador terá que dirigir os cliques;
- o *chat* bloqueia o *input* quando chega ao fim do fluxo, dando a opção de o utilizador voltar ao início;
- o estado da conversa é guardado nas *cookies* de sessão.

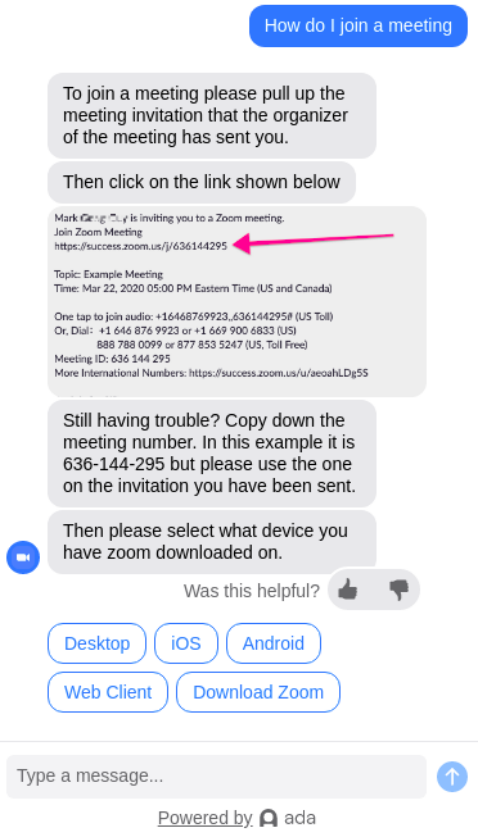


Figura 2.9: Zoom - "workflow" de utilização, após clique numa *quick action*

2.4 Tecnologias Disponíveis

Após analisadas as abordagens existentes no mercado em 2.3, é fundamental proceder-se à escolha de qual a tecnologia a utilizar no desenvolvimento no projeto. Para isso, foram exploradas várias alternativas, das quais apenas se selecionaram três, pelo critério de já terem sido utilizadas em projetos anteriores (quer pessoais, quer universitários) ao longo do ano de desenvolvimento deste projeto.

Portanto, esta secção descreve detalhadamente cada uma das tecnologias de desenvolvimento de *chatbots*, as suas vantagens e desvantagens, bem como funcionalidades consideradas extra e, finalmente, uma comparação entre elas. O intuito é perceber quais as tecnologias utilizadas, a forma de integração com outras plataformas (necessário para o *chatbot*), os canais de comunicação, o suporte de documentação/comunidade, entre outros tópicos.

2.4.1 Azure Bot Services

A Microsoft[11] suporta o desenvolvimento de *chatbots* através da Bot Framework[12], que pode ser integrado em qualquer *website*, tal como os descritos anteriormente através de, por exemplo, um *iframe*.

Através de outro serviço também desta empresa, intitulado *Language Understanding Intelligent Service* (LUIS)[13], é possível treinar modelos de inteligência artifical, definindo *Intents* que representam operações realizadas e entidades que constituem objetos do domínio. Após definidos para cada *Intent*, devem ser criadas frases que sejam relacionadas com aquela operação (ex: *Intent* - "ConsultaCriar"; Frase - "Boa tarde, queria marcar uma consulta").

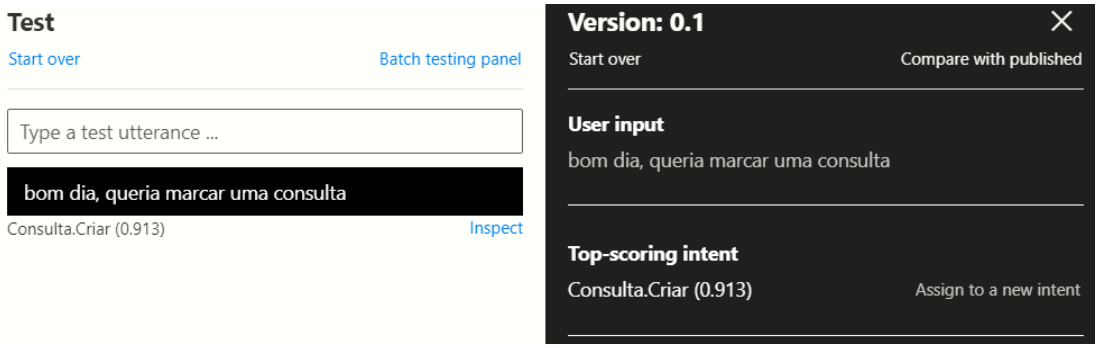


Figura 2.10: Azure - exemplo de identificação de um *Intent* de marcação de uma consulta

Posto isto, resta apenas treinar o modelo com o conjunto de frases previamente definidas, para que este se possa adaptar a diversos tipos de *inputs*. Adicionalmente, o serviço LUIS pode depois ser integrado através de uma comunicação *Representational State Transfer* (REST), precisando apenas da definição de um *webhook* que fica à escuta de novas mensagens recebidas.

Tal como referido anteriormente são apresentadas as vantagens, desvantagens e funcionalidades especiais encontradas:

Vantagens

- poderosa AI, com interface bem desenvolvida, que permite uma fácil definição de quais os *Intents* a criar e quais os *inputs* a definir (ver 2.11);
- geração de projetos-base utilizando a *framework* disponibilizada, como, por exemplo, a criação de um "*echo-bot*" (chatbot que responde com a mesma mensagem que é inserida pelo utilizador);
- existência de *Intents* já pré-definidos com identificações poderosas (ex: o LUIS consegue identificar para o *Intent* "Data" o dia, mês e ano);
- estatísticas detalhadas como: previsões por *Intent* (corretas, não claras e incorretas) e número de chamadas à API do LUIS;

[11] https://www.microsoft.com/
[12] https://azure.microsoft.com/en-us/services/bot-services/
[13] https://www.luis.ai/

- lógica de utilização do modelo - "Treino. Teste. Publicação." - simples de entender (1º treinar o modelo; 2º testá-lo; 3º publicar para produção/*staging*);
- suporte de definição de frases multi-linguísticas;
- suporte de envio de ficheiros, por defeito.

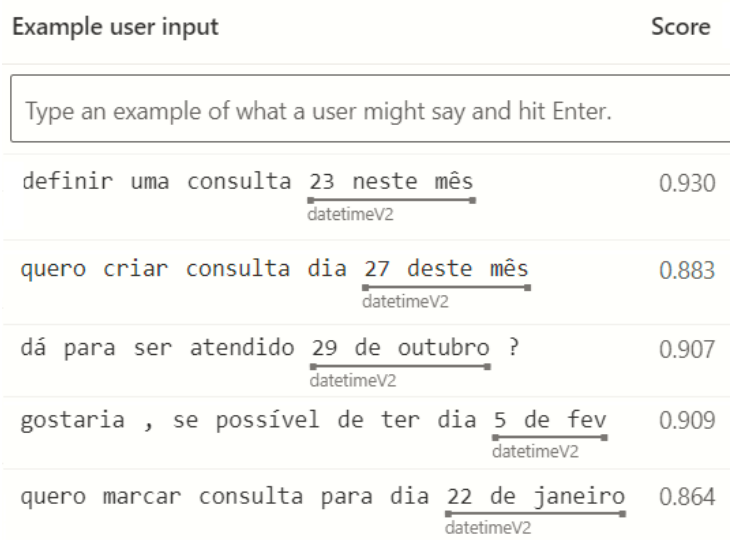


Figura 2.11: Azure - definição de frases e respetivos *scores*, depois de treinado o modelo

Desvantagens

- serviço pago a partir de 11€/mês, pelo serviço básico que inclui aplicações ilimitadas, 10*GigaByte* (GB) de espaço em disco, até 3 instâncias e computação dedicada;
- documentação extensível (mas ilegível e confusa), o que gera dificuldades no começo de utilização;
- para poder suportar funcionalidades AI, o projeto do *chatbot* tem que ser integrado com o LUIS;
- implementação de botões de *quick replies* suportada, mas só através do SDK da Microsoft.

Funcionalidades Extra

- é possível suportar o desenvolvimento através de um emulador, para efeitos de teste (serviço também oferecido pela Microsoft).

2.4.2 Messenger

Embora não tenha sido mencionado anteriormente, a plataforma de desenvolvimento do Messenger foi explorada, tal como a da Microsoft/Azure através do desenvolvimento de um projeto simples, para melhor compreensão do seu modo de funcionamento e a identificação de quais as suas funcionalidades e limitações.

O Messenger suporta também o desenvolvimento de chatbots para serem integrados em *websites*, com complemento de inteligência artificial, através da plataforma Wit.AI[14] (pertencente ao Facebook[15]) que, no contexto da solução experimental desenvolvida, não foi explorada (por defeito, a integração "básica" com a plataforma Wit.AI, pode ser habilitada no painel de controlo do Messenger).

Através da definição de *webhooks* é possível receber mensagens no modelo *JavaScript Object Notation* (JSON) que, dependendo do formato que é enviado para o utilizador, se torna variável. Caso a funcionalidade da Wit.AI esteja ativada, surge um campo extra, nomeado *Natural Language Processing* (NLP), nesta mensagem, se a identificação de uma *entity* for detetada para algum dos casos, como por exemplo: data/tempo, dinheiro/*currency*, números de telemóvel, *email*, temperatura, etc. No exemplo 2.12 é possível ver que, para a frase "vejo-te amanhã às 4 da manhã" [19], esta NLP consegue identificar a data e também qual o sentimento (neutro, no caso), embora apenas com uma confiança de sessenta e um por cento.

A comunicação destas mensagens para o utilizador é feita através da Graph API (API do Facebook dedicada à gestão de redirecionamento de mensagens), que suporta vários pedidos, todos eles descritos na documentação respetiva.

```
{...,
  "entities": {
    "wit$datetime:datetime": [
      {
        "id": "340464963587159",
        "name": "wit$datetime",
        "role": "datetime",
        "start": 8,
        "end": 23,
        "body": "tomorrow at 4pm",
        "confidence": 0.9704,
        "entities": [],
        "type": "value",
        "grain": "hour",
        "value": "2020-06-16T16:00:00.000-07:00",
        "values": [
          {
            "type": "value",
            "grain": "hour",
            "value": "2020-06-16T16:00:00.000-07:00"
          }
        ]
      }
    ]
  },
"traits": {
  "wit$sentiment": [
    {
      "id": "5ac2b50a-44e4-466e-9d49-bad6bd40092c",
      "value": "neutral",
      "confidence": 0.6162
    }
  ]
}
```


Figura 2.12: Messenger - pedido GET com identificação NLP da frase do utilizador [19]

Naturalmente que o exemplo indicado anteriormente, demonstra o pouco que este mecanismo de NLP consegue fazer, de modo que, a seguir, são apresentadas as vantagens, desvantagens e funções extra que foram encontradas na avaliação e experimentação desta plataforma:

Vantagens

[14] https://wit.ai/
[15] https://www.facebook.com/

- documentação simples, fácil de entender e configurar, com uma comunidade dedicada, pronta a ajudar;
- conjunto de botões, *templates*, *tags*, ações, entre outros (ver 2.13), disponibilizadas para integração direta (basta mudar a configuração do envio em JSON);
- separação, por *webhook*, de eventos, na receção das mensagens (reações - "*message reactions*", mensagens simples - "*messages*", etc);
- viabilização de integração com agentes (o utilizador comunica com uma página do Facebook, isto permite que, a qualquer momento, quem faz a gestão da página possa intervir);
- suporte, por defeito, do envio de *emojis*, ficheiros, imagens, *gifs*, etc;
- suporte multi-linguístico.

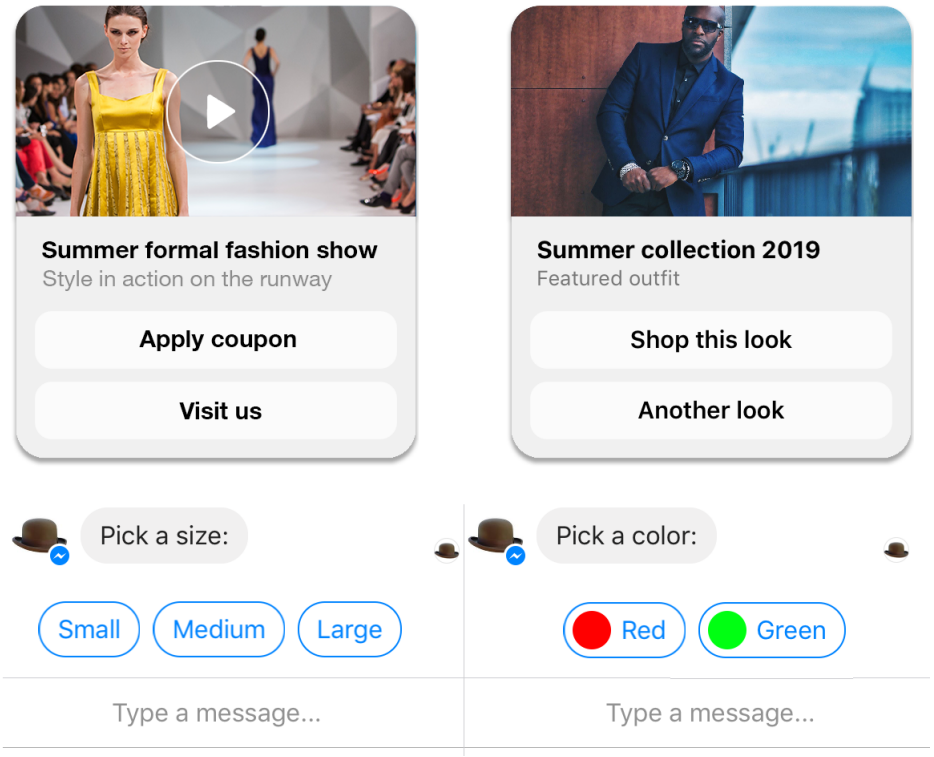


Figura 2.13: Messenger - exemplos de templates [20] e *quick reply actions* [9]

Desvantagens

- funcionalidades limitadas, às suportadas pelo Messenger (*templates*, botões, com exceção das *webviews* que são customizáveis);
- limitação de pedidos à Graph API (200 x o número de utilizadores, por hora);
- dificuldades em seguir o fluxo da conversa (pelo estudo feito não foi detetada nenhuma forma de identificação do estado da conversa).

Funcionalidades Extra

- consola *online* para testes à Graph API, integrada com a conta do Facebook;
- controlo de permissões no *dashboard* do Messenger (permissões de acesso ao *chatbot*, pedidos de mensagens, *roles*, etc);
- suporte de ambientes, tal como o Azure, de desenvolvimento, testes (*staging*) e produção;

2.4.3 Botpress

O Botpress[16] é uma ferramenta que se distingue de todas as outras, pelo desenvolvimento do código ser possível através de uma *user interface* (ver 2.14). Esta plataforma, concebida em NodeJS[17], foi desenhada para que, quem a utiliza, não tenha que recorrer a ficheiros de código, mas havendo sempre a possibilidade de ser realizado caso seja necessário. Utiliza ainda bases de dados SQLite e PostgreSQL para persistência dos dados e no *front-end* utiliza uma combinação de React[18], Redux[19] e Bootstrap, o que significa que o desenvolvimento é 100% Javascript.

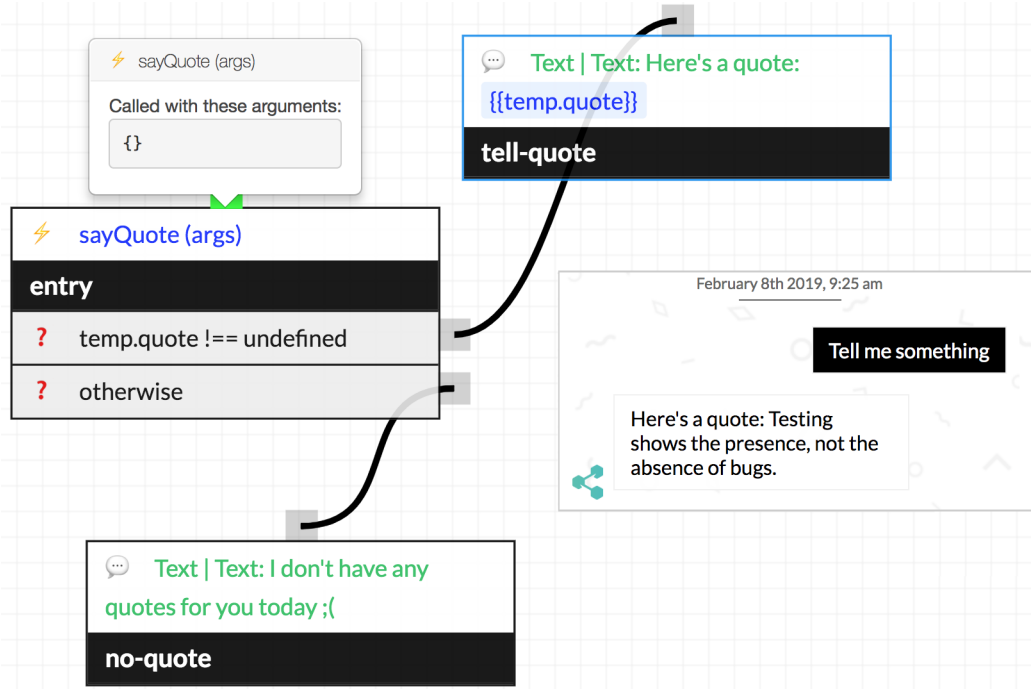


Figura 2.14: Botpress - exemplo de um fluxo básico em que o *bot* responde com uma frase aleatória [21]

Através de um fluxo/*flow* é possível definir nós/*nodes* ou *skills* (predefinidos), que variam entre: escolhas de opção múltipla, chamadas a API's, preenchimento de formulários, envio de *emails*, etc. Salvo estes, outros poderão ser definidos recorrendo a *actions* ou criando novos módulos [22] - código personalizado do lado do servidor, que é executado pelo *bot* como parte de um fluxo de conversa.

[16] https://botpress.com/
[17] https://nodejs.org/en/
[18] https://reactjs.org/
[19] https://redux.js.org/

Suplementarmente, esta ferramenta inclui também já um *debugger*, um emulador (para testes), módulos NLU e QnA, um *dashboard* de gestão administrativa e suporta múltiplos canais de mensagens. A comunicação pode ser feita através do *script* colocado no *site* a integrar ou através da Converse API (canal de comunicação REST desta ferramenta).

Esta plataforma, tal como as mencionadas anteriormente, foi devidamente explorada, pelo que foi possível retirar diversas conclusões relativamente às vantagens, desvantagens e funcionalidades extra:

Vantagens

- é uma plataforma *open source*, em constante desenvolvimento, com suporte da comunidade;
- suporte nativo de um sistema NLU nativo (offline - não requer conexão à internet) e QnA;
- alta escalabilidade e boa *Graphical User Interface* (GUI), o que permite que possa ser mantido até mesmo por *non-developers*;
- controlo dos dados é todo da organização que implementa o *chatbot*, isto é, o Botpress não tem acesso a nenhum tipo de dados.

Desvantagens

- limitado, tanto em termos de documentação, como em termos de funcionamento AI (modelos que funcionam para a generalidade dos casos e utilizadores, mas nem todos);
- poucos *skills* definidos à partida (ver 2.15);
- API de comunicação REST muito básica, apenas permitindo enviar mensagens num formato JSON específico;
- dificuldade na implementação de novos módulos customizados.

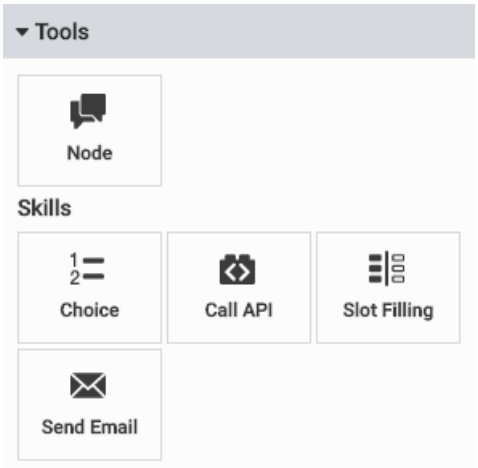


Figura 2.15: Botpress - *skills* disponíveis pela ferramenta [23]

Funcionalidades Extra

- página de análises estatísticas (utilizadores ativos, novos utilizadores, número de sessões, número de mensagens trocadas, etc);

- *debugger* e emulador, por defeito, já incluídos, bem como um *endpoint* de comunicação via API.

2.4.4 Comparação

Realizada uma análise detalhada de cada uma das ferramentas exploradas anteriormente, em 2.4, apresenta-se na tabela 2.1 a comparação entre as mesmas, para os pontos identificados como cruciais na escolha de uma plataforma de *design* e desenvolvimento de um *chatbot*.

	Azure Bot Services	**Messenger**	**Botpress**
Open source	Não	Não	Sim
Mecanismo NLP e QnA	Sim, *online*	Sim, *online*	Sim, *offline*
Interface Gráfica (Servidor)	Parcial (LUIS)	Não	Total
Suporte Comunidade / Documentação	Extensivo, mas difícil	Extensivo e fácil	Insuficiente
HITL	Sim	Sim	Sim
Suporte Multi-Linguístico	Sim	Sim	Sim, pago
Comunicação via REST API	Sim	Sim	Sim
Comunicação via JavaScript API	Sim	Sim	Sim
Analytics/Estatísticas	Sim	Sim	Sim

Tabela 2.1: Comparação entre as diferentes tecnologias estudadas

Das três plataformas, apenas o Botpress é uma ferramenta *open source*, tendo as restantes um custo associado e é a única que suporta um mecanismo de NLP *offline* gratuito, tendo as outras que processar os dados do utilizador na *cloud*. O Azure 2.4.1 contém uma boa interface gráfica para treino de modelos de ML, contudo distancia-se da *framework* de desenvolvimento do *chatbot*, por ser um serviço extra/à parte, enquanto que o Botpress integra diretamente este treino de modelos num único *dashboard* de desenvolvimento. Além disso, é a única ferramenta que suporta uma interface gráfica para desenvolvimento quase total do *bot*. A integração de *live-chat* é também fundamental, por ser um dos requisitos definidos nos objetivos em 1.3, sendo que, na tabela representada por HITL, todas possuem uma forma de interromper o fluxo do *chatbot* para responder aos utilizadores. O suporte multi-linguístico é essencial, uma vez que a E-goi suporta na sua plataforma quatro idiomas e, das diversas plataformas, todas parecem suportar esta interpretação, excepto o Botpress, que apresenta um custo extra para esta funcionalidade.

De notar que, pelo estudo realizado e tudo que foi abordado anteriormente, quase todas as plataformas analisadas contêm a maior parte dos mecanismos necessários para desenvolvimento de *chatbots* úteis e funcionais, colocando-se a dificuldade de implementação nas integrações necessárias e na interpretação da documentação destas *frameworks*.

Capítulo 3

Análise de Valor

Este capítulo responde à pergunta "qual o valor que este projeto traz para o cliente". Como descrito em [24], a definição "*costumer value*" é utilizada pela literatura de Marketing para descrever, tanto o valor que é entregue ao consumidor pelo fornecedor, como o valor que é entregue ao fornecedor pelo consumidor. É um processo sistemático, formal e organizado de analisar e avaliar um produto, em relação às necessidades do cliente [25].

Começando pelo modelo *New Concept Development* (NCD), serão abordados os tópicos de identificação e descrição das oportunidades, com recurso a uma análise *Strengths Weaknesses Opportunities Threats* (SWOT), seguido da geração e identificação de ideias e, por fim, apresentado o modelo *Analytic Hierarchy Process* (AHP), que corresponde à seleção da ideia final. Finalmente será apresentada a proposta de valor e o modelo *Quality Function Deployment* (QFD).

3.1 Identificação e Análise da Oportunidade

O modelo NCD é um modelo relacional, que descreve um conjunto de elementos que fazem parte do *Fuzzy Front End* (FFE), procurando aumentar o valor, a quantidade e a probabilidade de sucesso de conceitos chave entrarem no ciclo de desenvolvimento e comercialização de um produto [26]. Os cinco elementos de atividade estão descritos na área interna do modelo e destacam-se por: identificação de oportunidade, análise de oportunidade, geração e enriquecimento de ideias, seleção de ideias e definição do conceito.

Na identificação de oportunidades existem várias técnicas de análise, por exemplo, observando a tendência das tecnologias, dos consumidores, dos competidores ou de mercado. Como descrito anteriormente, na secção 2.2, é possível observar uma tendência corporativa que advém, cada vez mais, da necessidade de um sistema que dê resposta aos clientes e tenha similaridades com um humano (seja empático com o cliente). Isto significa que, os artigos ou *Frequently Asked Questions* (FAQ), disponibilizados pelas organizações, estão aquém das necessidades dos consumidores, sendo fundamental um sistema extra, de preferência automático, para esclarecimento de dúvidas.

Surge aqui então uma oportunidade para a utilização desses sistemas, intitulados *chatbots*. Segundo o artigo [27], os *chatbots* podem ajudar a reduzir os custos de serviços ao cliente até 30% e, adicionalmente, uma pesquisa de centros de atendimento globais em [28], revelou que 56% das empresas no setor de multimédia e tecnologia planeiam investir em centros de serviço ao cliente *Artificial Intelligence* (AI). Complementarmente, na pesquisa referida, concluiu-se que um mau serviço ao cliente impede que, 3 em cada 5 clientes efetuem uma compra futura.

Ora, oferecendo a E-goi um plano gratuito de 30 dias, que procura inspirar os clientes e conquistar a sua confiança na plataforma, para que, mais tarde eles possam efetuar a compra de um plano (deixem de ser apenas contas *trial*) e se tornem em clientes satisfeitos com o serviço, é fundamental que a experiência inicial e a disponibilidade de esclarecimento para qualquer questão, estejam sempre disponíveis e sejam o máximo eficientes.

Por fim, o artigo [29] refere também que 80% das perguntas são "rotineiras", o que significa que os agentes perdem tempo a responder a perguntas que poderiam ser respondidas de forma automática (por um *chatbot*), libertando-os para maiores desafios. Este facto tem como consequência o aumento do custo da empresa ao investir em mais agentes ou clientes insatisfeitos por não haver disponibilidade de alguém que possa dar esclarecimentos.

Sucintamente, a falha identificada é a falta de um serviço disponível 24 horas por dia, 7 dias por semana, de resposta automática, com um *toolset* ou conjunto de ferramentas, que consigam esclarecer todas (ou quase todas) as dúvidas dos consumidores, bem como permitir orientá-los em qualquer tarefa que estes precisem de cumprir.

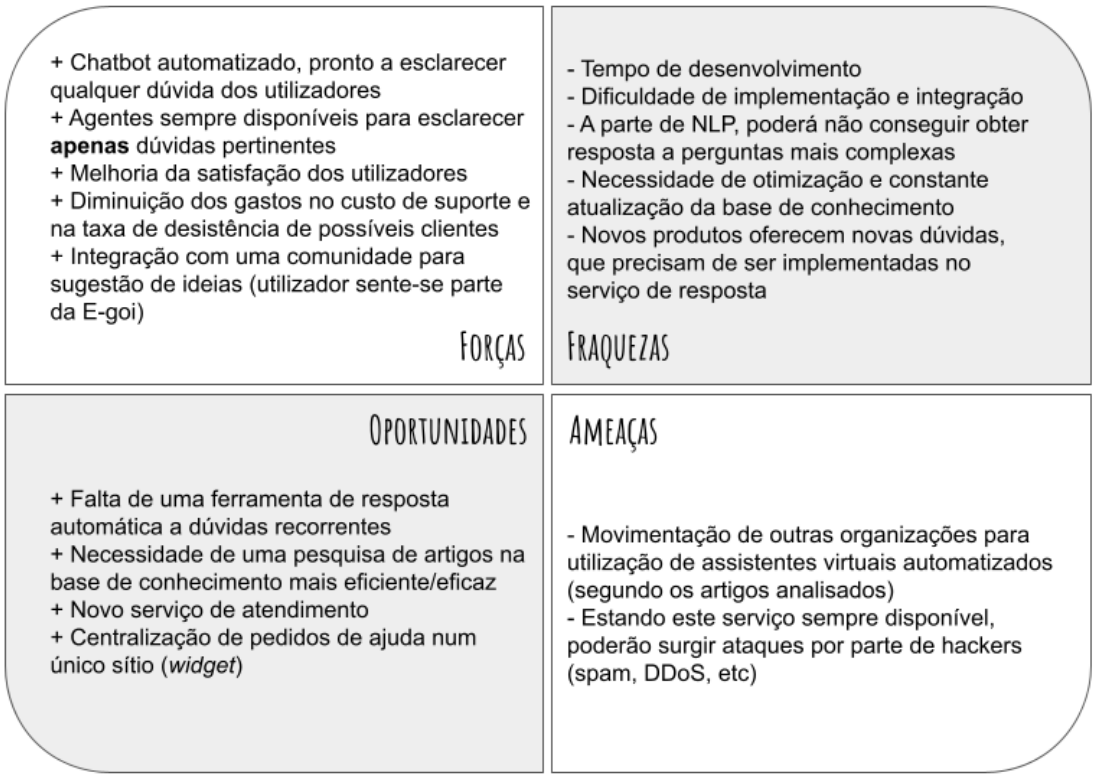


Figura 3.1: Análise SWOT - Forças, Fraquezas, Oportunidades e Ameaças

Considerando os artigos analisados, na figura 3.1 é possível verificar, através do modelo SWOT quais são as forças, fraquezas, oportunidades e ameaças, em relação ao projeto desenvolvido. Respetivamente, o primeiro campo define como pontos fortes: a automatização de respostas a dúvidas do utilizador, a alocação do tempo dos agentes para dúvidas mais explícitas, a melhoria da satisfação dos clientes, diminuição de custos no setor de suporte e da taxa de desistência, e por fim, integração dos utilizadores na plataforma (através da oportunidade de estes darem sugestões de melhoria no *widget*). De seguida, como pontos fracos: o custo temporal e a dificuldade de implementação, é sem dúvida, o maior problema (sendo um projeto quase novo e com uma dimensão abrangente de tecnologias), a falta de

resposta a perguntas complexas, a constante necessidade de otimização e acrescento de novas respostas para novos produtos. Quanto às oportunidades, é fundamental a existência de uma ferramenta centralizada, capaz de responder às dúvidas de utilizadores existentes (novos ou já com experiência) e a melhoria da base de conhecimento existente, acrescentando também a sugestão de artigos do Blog, vídeos do Youtube e cursos do Udemy. Concluindo, foram identificadas ameaças, como descrito nos parágrafos anteriores, da movimentação de outras plataformas para utilização de mecanismos de *Machine Learning* (ML) e a possibilidade de existência de ataques por parte de *hackers*, uma vez que, o *chatbot* poderá ter conhecimento de informações sensíveis sobre o utilizador.

3.2 Geração e Enriquecimento de Ideias

O processo de geração de uma ideia concreta começa pelo nascimento, seguido do desenvolvimento e, por fim, a maturação do conceito..., sendo esta geração portanto, evolucionária. Como descrito em [26], "*Ideas are built up, torn down, combined, reshaped, modified and upgraded.*", o que significa que pode, desde que é concebida a ideia, ser reestruturada múltiplas vezes ou até mesmo destruída, antes de atingir a forma final.

Existem diversas formas e metodologias para o nascimento de ideias, sendo as sessões de *brain-storming* um dos processos formais e, possivelmente, mais conhecidos. Como este, existe também: uma cultura organizacional que permita aos trabalhadores investir tempo em novas ideias (testando-as); as necessidades do mercado ou negócio terem que continuamente seguir o avanço natural das tecnologias; a rotação de trabalho/equipas para estimular a troca de conhecimentos, entre outros. Já no contexto da E-goi, o *widget-help* surge "por necessidade", isto é, tornou-se indispensável colocar, num único lugar, todos os elementos de ajuda e de serviço, que estariam até agora dispersos pela plataforma. O surgimento da ideia de um novo menu (que seria mais condensado e removeria a opção de ajuda), ao mesmo tempo do desenvolvimento desta ideia, veio afinar também ainda mais a necessidade deste serviço. Contudo, a ideia de um serviço centralizado de ajuda não ficou por aqui, sendo continuamente alterado em diversas sessões de *brain-storming*, até ser atingida a forma final, segundo o *Chief Technology Officer* (CTO) da empresa.

3.3 Seleção de Ideias

3.3.1 Avaliação AHP

Um dos métodos mais utilizado, no apoio à tomada de decisão e de ajuda na realização de decisões multi-critério discretas, é o método AHP, porque permite a utilização, tanto de critérios qualitativos, como quantitativos, no processo de avaliação de alternativas. Esta secção descreve a utilização desse método, segundo o artigo [30], do qual foram utilizadas diversas tabelas/figuras e seguidas as fases de desenvolvimento, até ser alcançada a solução ideal.

Tendo em conta a síntese comparativa realizada na tabela 2.1, apresentada no capítulo 2, foram selecionados três critérios que se distinguem nas ferramentas analisadas, para serem utilizados na metodologia AHP. Estes critérios foram escolhidos pelo facto de terem sido obtidos resultados diferentes na comparação das três ferramentas e serem considerados de

grande importância para a tecnologia a utilizar no desenvolvimento do projeto. Todas as tabelas apresentadas a seguir foram estruturadas com recurso à ferramenta Excel[1].

Fase 1 - Construção da árvore hierárquica de decisão

Nesta fase é definido o problema e um diagrama hierárquico (ver 3.2), que se divide em três partes: o objetivo, escolha da tecnologia para desenvolvimento do projeto; os critérios, suporte da ferramenta através de uma comunidade ou documentação bem estruturada, uma *User Interface* (UI) de desenvolvimento de código para posteriormente poder ser utilizada por utilizadores *non-developers* e a ferramenta ser *open source*; por fim, as alternativas, que refletem o estado da arte, nomeadamente a secção 2.4.

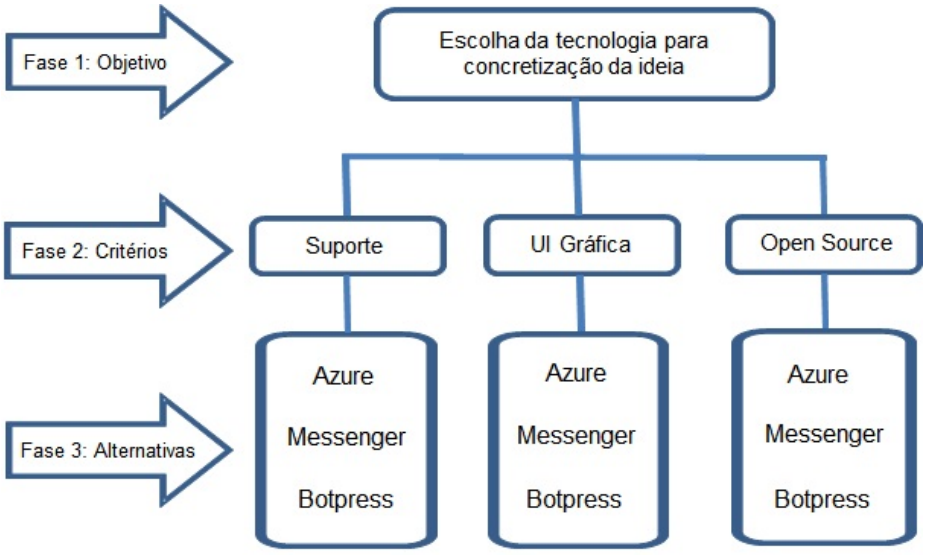


Figura 3.2: Árvore hierárquica de decisão

Fase 2 - Comparação das alternativas e critérios

De seguida, apresenta-se uma matriz que compara os elementos da hierarquia apresentada anteriormente, tendo em conta a tabela da "Escala Fundamental" em 3.1. Esta tabela indica o valor, numa escala de valores de 1 até 9, de comparação entre dois critérios ou duas alternativas.

[1] https://office.live.com/start/Excel.aspx

Nível de importância	Definição	Explicação
1	Igual importância	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 à outra
9	Importância absoluta	A evidência favorece uma atividade em relação à outra com o mais alto grau de certeza
2,4,6,8	Valores intermediários	Quando se procura uma condição de compromisso entre duas definições

Tabela 3.1: Escala fundamental - Níveis de importância de comparações [30]

Como tal, consideraram-se as seguintes premissas:

a) A UI gráfica é fracamente mais importante que o Suporte;

b) O *Open Source* é fortemente mais importante que o Suporte;

c) O *Open Source* é fracamente mais importante que a UI.

	Suporte	UI Gráfica	*Open Source*
Suporte	1	$\frac{1}{3}$	$\frac{1}{5}$
UI Gráfica	3	1	$\frac{1}{3}$
Open Source	5	3	1

$$A = \begin{bmatrix} 1 & \frac{1}{3} & \frac{1}{5} \\ 3 & 1 & \frac{1}{5} \\ 5 & 3 & 1 \end{bmatrix}$$

Tabela 3.2: Tabela/Matriz de comparação dos critérios selecionados

Construiu-se a tabela/matriz em 3.2, utilizando os valores obtidos através das premissas e da Escala Fundamental, que permite estabelecer uma comparação entre os três critérios selecionados.

Fase 3 - Prioridade relativa de cada critério

A terceira fase consiste em estabelecer a prioridade relativa de cada critério, normalizando a matriz da 2ª fase, através da divisão de cada um dos valores das células pelo respetivo total da sua coluna. Depois, realizando a média de cada linha, é possível obter o valor da prioridade relativa ou vetor próprio - identificação da importância de cada critério (ver 3.3).

	Suporte	UI Gráfica	*Open Source*
Suporte	0.111(1)	0.0769	0.1304
UI Gráfica	0.333(3)	0.2308	0.2174
Open Source	0.555(5)	0.6923	0.6522

$$VP = \begin{bmatrix} 0.11 \\ 0.26 \\ 0.63 \end{bmatrix}$$

Tabela 3.3: Matriz normalizada e Vetor da Prioridade/Vetor de Pesos (VP) Relativa de cada critério

É possível verificar que o critério *Open Source* aparece em primeiro lugar (valor de 0.63), seguido do critério UI Gráfica (0.26) e, por fim, o critério de Suporte em último lugar (com um valor de 0.11).

Fase 4 - Avaliar a consistência das prioridades relativas

De forma a validar a consistência dos resultados apresentados nas fases anteriores, é calculada a Razão de Consistência. Para tal, são utilizadas três fórmulas, das quais: $A \times VP = h_{max} \times VP$ permite calcular o Valor Próprio (h_{max}), $IC = (h_{max} - n)/(n-1)$ serve para calcular o Índice de Consistência (IC) e $RC = IC/IR$ permite calcular a Razão de Consistência (RC) dividindo o IC pelo Índice Aleatório (IR) - valor calculado para matrizes quadradas de ordem n pelo Laboratório Nacional de Oak Ridge, EUA. Todos os cálculos realizados podem ser consultados no ficheiro em anexo (ver anexo A).

Obtido o valor h_{max} através da 1ª fórmula, foi calculado o Índice de Consistência pela segunda fórmula, o que permitiu realizar a fórmula final:

$$RC = \frac{IC}{IR} <=> RC = \frac{0.0194}{0.58} <=> RC = 0.03 < 0.1$$

Sendo a Razão de Consistência (RC), abaixo de 10% significa que as paridades são confiáveis e não aleatórias.

Fase 5 - Construção da matriz de comparação paritária para cada critério

Novamente foi realizado o procedimento da Fase 2 e 3, para calcular os vetores prioridade, mas tendo em conta agora as alternativas possíveis e para cada critério. Para cada critério, foram consideradas várias premissas que podem ser consultadas no anexo A. Com os vetores prioridade foi construída a matriz de comparação em 3.4.

	Suporte	UI Gráfica	*Open Source*
Azure	0.0978	0.2316	0.1428
Messenger	0.7151	0.0718	0.1428
Botpress	0.1871	0.6965	0.7143

$$PC = \begin{bmatrix} 0.11 \\ 0.26 \\ 0.63 \end{bmatrix}$$

Tabela 3.4: Matriz Prioridade e Peso dos Critérios (PC) calculado em 3.3

Fase 6 - Obter a prioridade composta para as alternativas

Multiplicando a Matriz Prioridade pelo Peso dos Critérios, é possível obter as prioridades de cada alternativa Azure, Messenger e Botpress, respetivamente:

$$Prioridade = \begin{bmatrix} 0.16 \\ 0.18 \\ 0.65 \end{bmatrix}$$

Fase 7 - Escolha da alternativa

Observando a matriz final obtida na fase anterior, é possível concluir que a alternativa "Botpress" aparece como a mais indicada para o desenvolvimento deste projeto, em função dos critérios definidos e das suas respetivas importâncias.

3.3.2 Modelo QFD

Como descrito em [31], o QFD é um sistema para desenho de um produto ou serviço, tendo em vista as necessidades do cliente, envolvendo todos os membros de uma organização, que oferecem ou fornecem esse produto/serviço. Cada termo desta sigla pode ser descrito como:

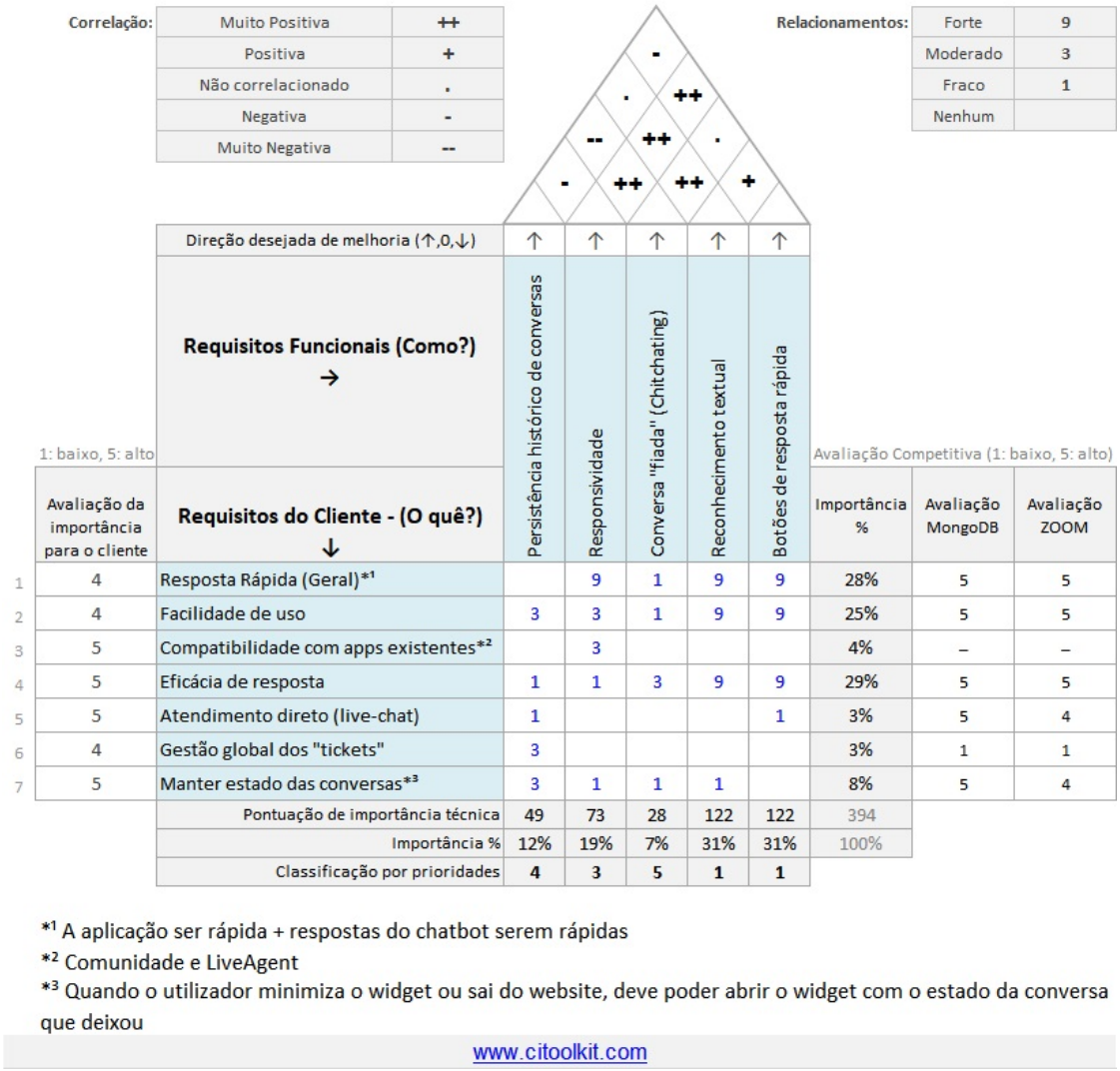
- Qualidade - ir de encontro às necessidades do cliente;
- Função - "O Que Deve Ser Feito", focado na atenção;
- Implantação/*Deployment* - "Quem O Vai Fazer, Quando".

O intuito deste modelo é permitir uma maior e mais rápida qualidade dos produtos no mercado, a um preço mais baixo, seguindo um *customer-driven design* (orientado pelas necessidades dos consumidores) e disponibilizar um mecanismo de consulta para futuras melhorias de processos ou *design*. As vantagens deste sistema são diversas, pois permitem que: as organizações possam melhorar os processos de desenvolvimento, exista um menor número de mudanças posteriores, surjam menos problemas em fases iniciais, a reputação da qualidade seja levada a sério, a definição do produto esteja documentada e, no geral, exista uma melhoria do negócio.

Posto isto, na figura 3.3 é apresentado este modelo, onde são descritas as necessidades do cliente (do lado esquerdo), levantados nas discussões realizadas com o CTO, bem como o grau de importância de cada um dos requisitos, em cima estão descritos os requisitos funcionais do problema e qual a direção de melhoria (aumentar ou diminuir). Os valores inseridos nas avaliações das plataformas da MongoDB e do Zoom, foram atribuídos considerando o estudo realizado no capítulo 2.

De notar que o modelo descrito na figura anterior, seguiu uma abordagem similar à utilizada numa aula do *Massachusetts Institute of Technology* (MIT)[2] e, ao invés de serem utilizados os símbolos tradicionais, foram utilizadas as respetivas escalas para os relacionamentos e avaliações (1-5 e 0-9, respetivamente).

[2] https://www.youtube.com/watch?v=u9bvzE5Qhjk



Correlação:	
Muito Positiva	++
Positiva	+
Não correlacionado	.
Negativa	-
Muito Negativa	--

Relacionamentos:	
Forte	9
Moderado	3
Fraco	1
Nenhum	

	Avaliação da importância para o cliente	Requisitos do Cliente - (O quê?) ↓	Persistência histórico de conversas	Responsividade	Conversa "fiada" (Chitchating)	Reconhecimento textual	Botões de resposta rápida	Importância %	Avaliação MongoDB	Avaliação ZOOM
	1: baixo, 5: alto	Requisitos Funcionais (Como?) →						Avaliação Competitiva (1: baixo, 5: alto)		
		Direção desejada de melhoria (↑,0,↓)	↑	↑	↑	↑	↑			
1	4	Resposta Rápida (Geral)*1		9	1	9	9	28%	5	5
2	4	Facilidade de uso	3	3	1	9	9	25%	5	5
3	5	Compatibilidade com apps existentes*2		3				4%	–	–
4	5	Eficácia de resposta	1	1	3	9	9	29%	5	5
5	5	Atendimento direto (live-chat)	1				1	3%	5	4
6	4	Gestão global dos "tickets"	3					3%	1	1
7	5	Manter estado das conversas*3	3	1	1	1		8%	5	4
		Pontuação de importância técnica	49	73	28	122	122	394		
		Importância %	12%	19%	7%	31%	31%	100%		
		Classificação por prioridades	4	3	5	1	1			

*1 A aplicação ser rápida + respostas do chatbot serem rápidas
*2 Comunidade e LiveAgent
*3 Quando o utilizador minimiza o widget ou sai do website, deve poder abrir o widget com o estado da conversa que deixou

www.citoolkit.com

Figura 3.3: Modelo QFD aplicado aos requisitos especificados

Os resultados obtidos mostram que o "Reconhecimento textual" e os "Botões de resposta rápida" tem uma igual e maior importância, de 31%, em relação aos outros requisitos funcionais. Por outro lado, a importância da "Eficácia de resposta" tem um peso de 29% em 100%, ou seja, representa mais de $\frac{1}{4}$ da importância em relação ao total, para o cliente, estando a "Rapidez da aplicação", logo a seguir com um valor de 28%.

3.4 Definição do Conceito

O último passo do modelo NCD [26], deve ser visto como um *"gate document"* para o investimento neste projeto, isto é, deve apresentar os fatores/critérios, para as partes interessadas no mesmo (será descrito em maior pormenor em 3.5). Tendo em consideração as análises efetuadas anteriormente, através dos vários modelos, procede-se à definição do conceito com base nos seguintes critérios:

- os objetivos resumem-se em - criar um chatbot capaz de devolver artigos úteis para responder às dúvidas dos clientes (FAQ, YouTube, Udemy e blog); possibilitar o contacto do suporte da E-goi (e a criação/ alteração/listagem de tickets); permitir ver as notificações recebidas; dar acesso a serviços customizados (recuperação de listas de contactos, entre outros); e integrar o utilizador na comunidade para partilha de ideias/sugestões;
- enquadramento do projeto nas necessidades da organização, pela impossibilidade de resposta ao volume atual de questões e solitações de ajuda dos clientes atuais, bem como pelo aumento substancial diário de novos;
- no contexto mundial atual, estando o *e-commerce* em grande crescimento e tendo os clientes enormes necessidades de estratégias de venda dos produtos, a E-goi tem como obrigatoriedade fornecer-lhes um serviço com toda a ajuda possível (sentido de oportunidade para um projeto de *chatbot*);
- sendo todas as tecnologias consideradas para o desenvolvimento do projeto maioritariamente *open-source*, os únicos custos previstos seriam em *hardware* e num *software developer*, para desenvolvimento e manutenção do mesmo;
- não sendo elevados os custos de conceção e implementação de ideia, considera-se uma mais valia o presente projeto; além disso, prevê-se que o horizonte temporal de desenvolvimento total do projeto, esteja entre seis a doze meses.

3.5 Proposta de Valor

A proposta de valor é descrita em [32] pela definição de como itens de valor (produtos ou serviços), bem como serviços complementares que adicionam valor, são oferecidos e apresentados para satisfazer as necessidades dos clientes. Destaca-se também o parágrafo seguinte que refere:

> *"[...] the VALUE PROPOSITION element is an overall view of a firm's bundle of products and services that together represent value for a specific CUSTOMER SEGMENT. It describes the way a firm differentiates itself from its competitors and is the reason why customers buy from a certain firm and not from another."*
> ([32] - Osterwalder e Pigneur, 2003)

Esta definição é muito importante, pois explica que a Proposta de Valor oferece um ponto de vista dos produtos e serviços de uma empresa, que apresentam valor para um segmento específico de clientes. Destaca também a importância da diferenciação das empresas que disponibilizam os produtos certos aos clientes certos, o que é fundamental em ambientes altamente competitivos.

A Proposta de Valor pode ser respondida através de uma série de perguntas, tais como: "Qual é o produto?", "Para quem é o produto?", "A quem traz valor?", "Que valor o produto oferece?" e "Porque é que este produto é único?". Para dar resposta a todas estas questões, optou-se por apresentar o modelo Canvas de Osterwalder, que contém a Proposta de Valor (do lado esquerdo) e o Perfil do Cliente (do lado direito), na figura 3.4, com recurso ao *website* Strategyzer[3].

[3] https://www.strategyzer.com/

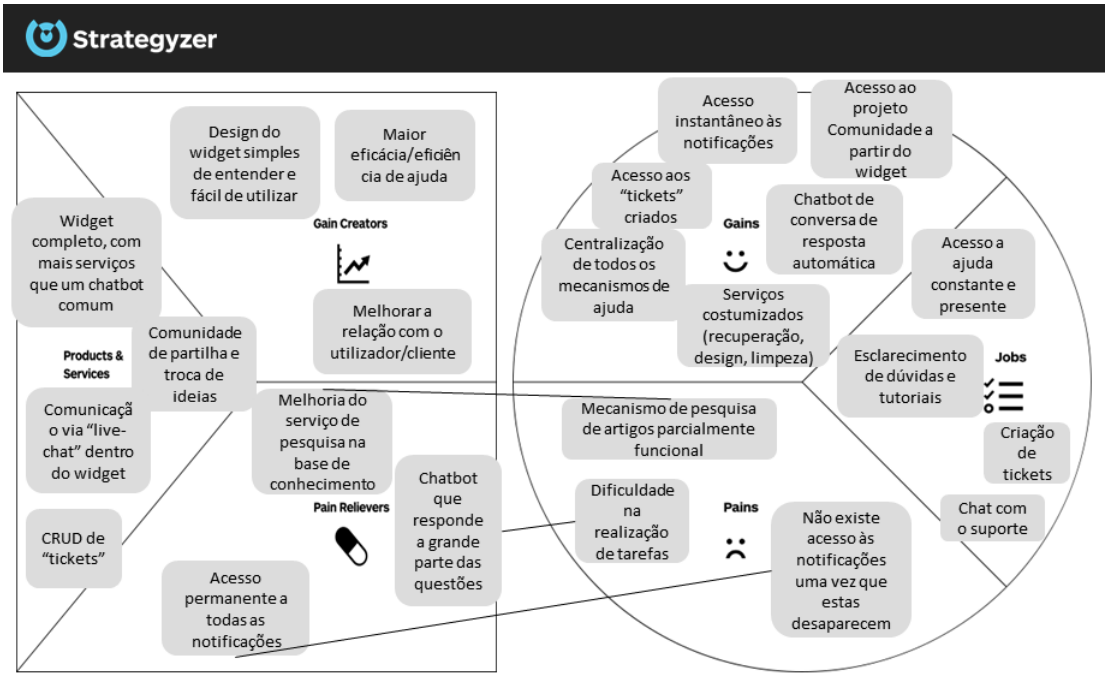


Figura 3.4: Proposta de Valor com o Perfil do Cliente [33]

Como referido em [33], os *Pain Relievers* descrevem como exatamente os produtos ou serviços aliviam as "dores" específicas dos clientes, os *Gain Creators* descrevem como os produtos ou serviços criam valor para os clientes e os *Products & Services* são apenas uma simples lista dos produtos oferecidos. Do lado direito, os *Gains* demonstram os resultados e benefícios que os clientes desejam, os *Jobs* representam o trabalho que os clientes tentam cumprir ou resolver e as *Pains* descrevem o que incomoda ou impede os clientes de tentar cumprir os seus *Jobs*.

Capítulo 4

Análise da Solução

Neste capítulo, será identificado o problema, as pessoas que pretendem uma solução para o mesmo, a natureza da solução desejada e de que forma será realizada a comunicação entre a equipa de software e os *stakeholders*. Serão apresentados todos os requisitos definidos, quem utilizará a solução desenvolvida, quais os benefícios desta solução, o ambiente em que será desenvolvida, bem como o modo de funcionamento atual dos componentes existentes.

4.1 Técnicas de Elicitação

Existem diversas técnicas de levantamento de requisitos descritas em [34], das quais se destacam: as entrevistas ou questionários, *workshops*, através de observação, análise de documentos, análise de interfaces gráficas e até análise das interfaces de sistema.

Como este projeto já se encontrava numa fase inicial de desenvolvimento, embora que suspensa durante um grande período de tempo, no momento de início do projeto foi entrevistado o *developer* que trabalhou neste projeto, ao longo de quatro meses. Através dessa reunião/entrevista, foi possível retirar e realizar uma troca de ideias, sobre quais os requisitos que teriam sido desenvolvidos, quais os que não teriam sido cumpridos e quais faltavam cumprir, bem como perceber a interpretação dada ao problema. Inclusive, foi ainda sucintamente dada uma breve explicação das tecnologias utilizadas, o modo como foram utilizadas, e discutidas ideias desenvolvimento futuro.

Embora a técnica de observação dos utilizadores consista em avaliar como estes realizam as suas tarefas, geralmente através de observação direta, como a E-goi possui grandes quantidades de informação sobre o comportamento dos utilizadores, nomeadamente estatísticas mensais sobre criações de conta, desistências, utilização do suporte, criação de *tickets*, entre muitas outras, optou-se por analisar esta informação, para avaliação de novos requisitos e alteração dos existentes. Por motivos de confidencialidade, não serão divulgadas mais informações, acerca destas estatísticas, mas pretende-se que fique clara a ideia de que existe um conjunto relativamente grande de informação útil e rica.

Além de ambas as técnicas mencionadas nos parágrafos anteriores, foi também realizada a análise de documentação existente, bem como a análise das interfaces gráficas e de sistemas existentes na E-goi, em comparação com a desenvolvida, estudando as soluções oferecidas de modo a descobrir os requisitos funcionais e do utilizador.

4.2 Elaboração dos Requisitos

Após reunidos todos os conhecimentos do projeto desenvolvido e retiradas informações através do comportamento dos utilizadores na elicitação dos requisitos, é importante definir concretamente os casos de uso (requisitos funcionais) tendo em conta os cenários que descrevem como os utilizadores irão interagir com o sistema.

Como tal, são apresentados os diferentes requisitos, uma breve descrição e um diagrama *Unified Modeling Language* (UML) - linguagem gráfica para especificação, documentação e visualização de sistemas de *software* [35] -, na lista em 4.2.1 e os requisitos não funcionais em 4.2.2

4.2.1 Requisitos Funcionais

Identificam-se os seguintes casos de uso ou requisitos funcionais:

UC-1) **Integração com o projeto da Comunidade**

O utilizador acede ao *widget*, dirigindo-se ao separador da Comunidade. O sistema mostra ao utilizador a Comunidade dentro do separador.

UC-2) **Integração com o LiveAgent/Botpress (Criação de Tickets)**

O utilizador acede ao *widget*, iniciando a conversa com o chatbot, seguindo o fluxo da conversa... até o sistema sugerir a criação de um *ticket*. O utilizador seleciona a opção, indicando qual é o problema e anexando uma imagem/documento (opcional). O sistema guarda o *ticket* e devolve a identificação para posterior consulta.

UC-3) **Listar/Consultar tickets**

O utilizador acede ao *widget*, dirigindo-se ao separador de *tickets*. O sistema mostra uma lista de todos os *tickets* do utilizador, com o respetivo estado (não respondido/-respondido/fechado).

UC-4) **Listar/Consultar notificações**

O utilizador acede ao *widget*, dirigindo-se ao separador de notificações. O sistema mostra um número limitado de notificações, para não sobrecarregar a página, ordenadas por data (mais recentes primeiro). O utilizador escolhe ver mais (opcional). O sistema mostra notificações mais antigas. O utilizador seleciona uma notificação. O sistema mostra toda a informação da notificação escolhida.

UC-5) **Integração com o LiveAgent (Continuação do fluxo do Ticket)**

O utilizador acede ao *widget*, dirigindo-se ao separador de *tickets*. O sistema mostra uma lista de todos os *tickets*. O utilizador seleciona um *ticket*. O sistema mostra a descrição do problema e a imagem (opcional) submetidos através do fluxo de conversa. O utilizador vê a resposta do suporte e submete uma nova mensagem. (Quando o agente responder, mais tarde, o sistema mostra a mensagem)

UC-6) **Integração com o LiveAgent (Live-chat)**

O utilizador acede ao *widget*, inicia conversa com o *chatbot*, seguindo o fluxo da conversa... até o sistema sugerir a opção de contactar o suporte (via *chat*). O utilizador seleciona a opção e inicia a conversa com um agente. Terminada a conversa,

o sistema pede ao utilizador uma avaliação do agente. O utilizador avalia (opcional) o agente. O sistema continua o fluxo da conversa.

UC-7) **Distinção entre notificações lidas/por ler**

O utilizador acede ao *widget*, dirigindo-se ao separador de *notificações*. O sistema mostra a lista de notificações distinguindo se a notificação já foi lida. O utilizador clica numa notificação marcada como "por ler". O sistema mostra a informação da notificação e atualiza a notificação para "lida".

UC-8) **Número de notificações/tickets não abertos**

O utilizador acede à página da E-goi. O sistema mostra o *widget* minimizado com o total resultante da soma de notificações por ler e *tickets* por responder. O utilizador abre o *widget*. O sistema mostra os dois separadores (notificações e *tickets*) com o respetivo número individual "por abrir/ler".

UC-9) **Filtro de pesquisa de Tickets**

O utilizador acede ao *widget*, dirigindo-se ao separador de *tickets*. O sistema mostra uma lista de todos os *tickets* do utilizador, com o respetivo estado (não respondido/-respondido/fechado) e uma barra de pesquisa. O utilizador seleciona o filtro pretendido (ex: não respondidos). O sistema mostra apenas os *tickets* com o(s) filtro(s) escolhido(s).

UC-10) **Integração com o Botpress (Fluxo de conversa/Troca de mensagens)**

O utilizador acede ao *widget*, dirigindo-se ao separador de início de uma nova conversa. O sistema envia uma mensagem personalizada de início de conversa (ex: Olá "nome de utilizador"). O utilizador continua o fluxo da conversa efetuando uma questão. O sistema devolve a resposta com artigos de resposta. O utilizador devolve feedback sobre a utilidade da resposta. O sistema processa a resposta e responde de acordo. O utilizador volta efetuar nova questão ou abandona o fluxo de conversa.

UC-11) **Persistência do estado de uma conversa**

O utilizador acede ao *widget*, dirigindo-se ao separador de início de uma nova conversa. O sistema envia uma mensagem de início de conversa. O utilizador continua o fluxo, abandonando a conversa numa das seguintes situações: minimiza o *widget*, termina a sessão, termina a sessão do *browser* ou desliga a internet; mais tarde resume ou abre novamente o *widget*. O sistema mostra a conversa no ponto exato em que foi deixado.

UC-12) **Consulta do histórico de conversas**

O utilizador acede ao *widget*, dirigindo-se ao separado de histórico de conversas. O sistema mostra um número limitado de conversas do utilizador. O utilizador seleciona uma conversa. O sistema mostra a troca de mensagens entre o utilizador e o *chatbot*.

UC-13) **Identificação NLP no Botpress**

O utilizador acede ao *widget*, dirigindo-se ao separador de início de uma nova conversa ou selecionando uma já existente. O sistema envia uma mensagem de início de conversa (caso seja uma nova conversa). O utilizador efetua uma questão ou escreve algo. O sistema deteta o que é pretendido e devolve a resposta correta.

UC-14) **Implementar a possibilidade de anexar ficheiros**

O utilizador acede ao *widget*, dirigindo-se ao separador de início de uma nova conversa ou selecionando uma já existente. O sistema envia uma mensagem de início de conversa (caso seja uma nova conversa). O utilizador seleciona o botão de anexar ficheiros. O sistema abre a janela de seleção de um ficheiro com os formatos de (PNG, JPG ou JPEG). O utilizador anexa um ficheiro, e é enviado pelo *chat* (caso o ficheiro esteja acima de um *threshold* é rejeitado com uma mensagem de erro). O sistema responde de acordo.

UC-15) **Implementar o envio de emojis**

O utilizador acede ao *widget*, dirigindo-se ao separador de início de uma nova conversa ou selecionando uma já existente. O sistema envia uma mensagem de início de conversa (caso seja uma nova conversa). O utilizador seleciona o botão de *emojis*. O sistema abre a janela de seleção de *emojis*. O utilizador seleciona um *emoji* e é enviado. O sistema responde de acordo.

UC-16) **Melhoria da pesquisa de artigos existente**

O utilizador acede ao *widget*, dirigindo-se ao separador de início de uma nova conversa ou selecionando uma já existente. O sistema envia uma mensagem de início de conversa (caso seja uma nova conversa). O utilizador efetua uma questão. O sistema pesquisa nos artigos das FAQ's, Blog, Youtube ou Udemy e sugere alguns dos resultados obtidos de acordo com a questão colocada.

UC-17) **Disponibilizar o widget para utilizadores não autenticados**

O utilizador dirige-se à página de login da E-goi e clica no botão do *widget*. O sistema abre o *widget* mostrando apenas a funcionalidade de chatbot, limitando todas as ações de contacto com o suporte humano.

UC-18) **Integração do Widget com os projetos da E-goi como Web Component**

O utilizador dirige-se a qualquer uma das páginas da E-goi. O sistema mostra o ícone do *widget* no canto da página.

UC-19) **Implementar persistência das pesquisas efetuadas**

O utilizador acede ao *widget*, dirigindo-se ao separador de início de uma nova conversa ou selecionando uma já existente. O sistema envia uma mensagem de início de conversa (caso seja uma nova conversa). O utilizador coloca dúvidas. O sistema guardar internamente a questão colocada (para mais tarde ser avaliada) e responde.

UC-20) **Implementar sistema de alarmística**

O utilizador acede ao *widget*, dirigindo-se ao separador de início de uma nova conversa ou selecionando uma já existente. O sistema envia uma mensagem de início de conversa (caso seja uma nova conversa). O utilizador envia uma mensagem. O sistema não consegue aceder ao mecanismo de conversação autónoma e gera um alerta (ex: envio de um email) a notificar que algo está errado com o servidor.

UC-21) **Implementar limite de mensagens nas conversas**

O utilizador acede ao *widget*, dirigindo-se ao separador de início de uma nova conversa ou selecionando uma já existente. O sistema disponibiliza apenas as mensagens mais

recentes. O utilizador efetua o *scroll* para cima. O sistema mostra mensagens mais antigas.

UC-22) **Implementar traduções no Widget-Help**

O utilizador acede ao *widget*. O sistema deteta a linguagem do utilizador e é apresentando na língua que está associada à conta. O utilizador dirige-se ao separador de início de uma nova conversa, efetuando uma pesquisa. O sistema responde e processa os dados na linguagem do utilizador.

UC-23) **Integração do projeto Admin com o Botpress (Consulta de pesquisas)**

O utilizador (da E-goi) acede ao painel de administração. O sistema mostra os separadores disponíveis. O utilizador dirige-se ao separador relativo ao Widget-Help. O sistema carrega as pesquisas efetuadas pelos clientes da E-goi (com a informação associada a cada uma).

4.2.2 Requisitos Não Funcionais

Seguindo o modelo *Functionality Usability Reliability Performance Supportability* (FURPS), publicado em 1994, modelo de qualidade de software criado por Robert B. Grady em [36], é possível descrever um conjunto de critérios (Funcionalidade, Usabilidade, Fiabilidade, Desempenho e Suportabilidade) que permitem avaliar um produto de software ou um processo de desenvolvimento de *software*. Através destas métricas, é possível medir o sucesso de um projeto, classificando o sistema da forma como ele deve funcionar.

A tabela 4.1 expõe as cinco métricas descritas no parágrafo anterior e acrescenta ainda os requisitos de design, de implementação, de interface e físicos (modelo FURPS+).

A classificação das métricas descritas no modelo criado por Grady, podem ainda ser elaboradas como em [37]. A métrica de Funcionalidade apresenta a capacidade, compatibilidade, portabilidade e segurança que o sistema deve ter. A Usabilidade descreve os fatores humanos, estéticos, de consistência, responsividade e documentação. A Fiabilidade é a robustez/disponibilidade do sistema. O desempenho demonstra os requisitos que o sistema deve apresentar em termos de velocidade, eficiência e escalabilidade. Por fim, no modelo FURPS, a Suportabilidade apresenta a modularidade, adaptação e flexibilidade de mudança do sistema. Quanto aos restantes, representam as restrições relativamente a padrões de *design* e processos de desenvolvimento de *software*, restrições de construção do sistema e limites de recursos, restrições das funcionalidades inerentes às interfaces e utilização de módulos externos, restrições físicas impostas pelo *hardware* utilizado para implantação do sistema, respetivamente.

Métrica	Descrição
Funcionalidade	Todas as funcionalidades existentes de comunicação com o suporte devem ser mantidas (Avaliação da conversa - Rating -, Ligar para um agente) através do desenvolvimento do UC-6; O *upload* de ficheiros deve ser limitado a imagens e com um máximo de 10 MB; A aplicação deve ser segura, seguindo as boas práticas de desenvolvimento de *software*.
Usabilidade	Deve ser responsiva, seguindo as boas práticas de *Material Design*; A aplicação deve ser intuitiva e fácil de entender e memorizar; A aplicação deve ter um número baixo de falhas ou garantir que falha o menos possível.
Fiabilidade	- *Não existentes*
Desempenho	- *Não existentes*
Suportabilidade	Deve estar disponível para funcionar em qualquer *browser* (Chrome, Mozilla, Internet Explorer,...); Suporte multí-linguístico (português, português-brasil, inglês e espanhol); A configuração do fluxo do *chatbot* deve ser bem documentada e facilmente entendível; A aplicação deve ser testada em vários níveis (*white-box testing* e *black-box testing*); O projeto deve ser fácilmente alterável, caso seja necessário no futuro.
Requisitos de design	Devem ser seguidas as interfaces providenciadas pela empresa.
Requisitos de implementação	Integração com a tecnologia Botpress; Integração com a tecnologia LiveAgent; Integração com a tecnologia ElasticSearch; Utilização da *framework* Angular[1] (versão 9) e as bibliotecas Angular Material e NgRx[2].
Requisitos de interface	- *Não existentes*
Requisitos físicos	- *Não existentes*

Tabela 4.1: Modelo FURPS+ aplicado ao sistema de Conversação Autónoma

Capítulo 5

Desenho da Solução

Para que se obtenha um produto funcional, além da análise dos requisitos é importante também realizar um planeamento faseado, definindo como é que a solução será desenvolvida e qual a estrutura/arquitetura a seguir. De forma a elaborar o desenho da solução, apresentam-se neste capítulo os vários artefactos do sistema a desenvolver (servindo como base para a implementação), tendo em conta os modelos C4 [38] e o modelo 4+1 de vista arquitetural [39].

5.1 Vista Lógica

Martin Fowler, autor conhecido na área de arquitetura de software, define o Modelo de Domínio como um conjunto de objetos web interconectados, onde cada objeto representa, de certa forma, algo individual, tão grande como uma organização ou tão pequeno como uma única linha num formulário de preenchimento [40]. Inserir um Modelo de Domínio num contexto aplicacional envolve injetar toda uma nova camada de objetos que modelam a área de negócio/*business layer* em que se está a trabalhar.

É fundamental que, no processo de criação de um Modelo de Domínio, sejam encontrados os objetos que representam os dados utilizados no negócio e os objetos que capturam as regras do negócio. Como tal, e como forma de complementar os requisitos identificados em 4.2, construiu-se o Modelo de Domínio representado na figura 5.1.

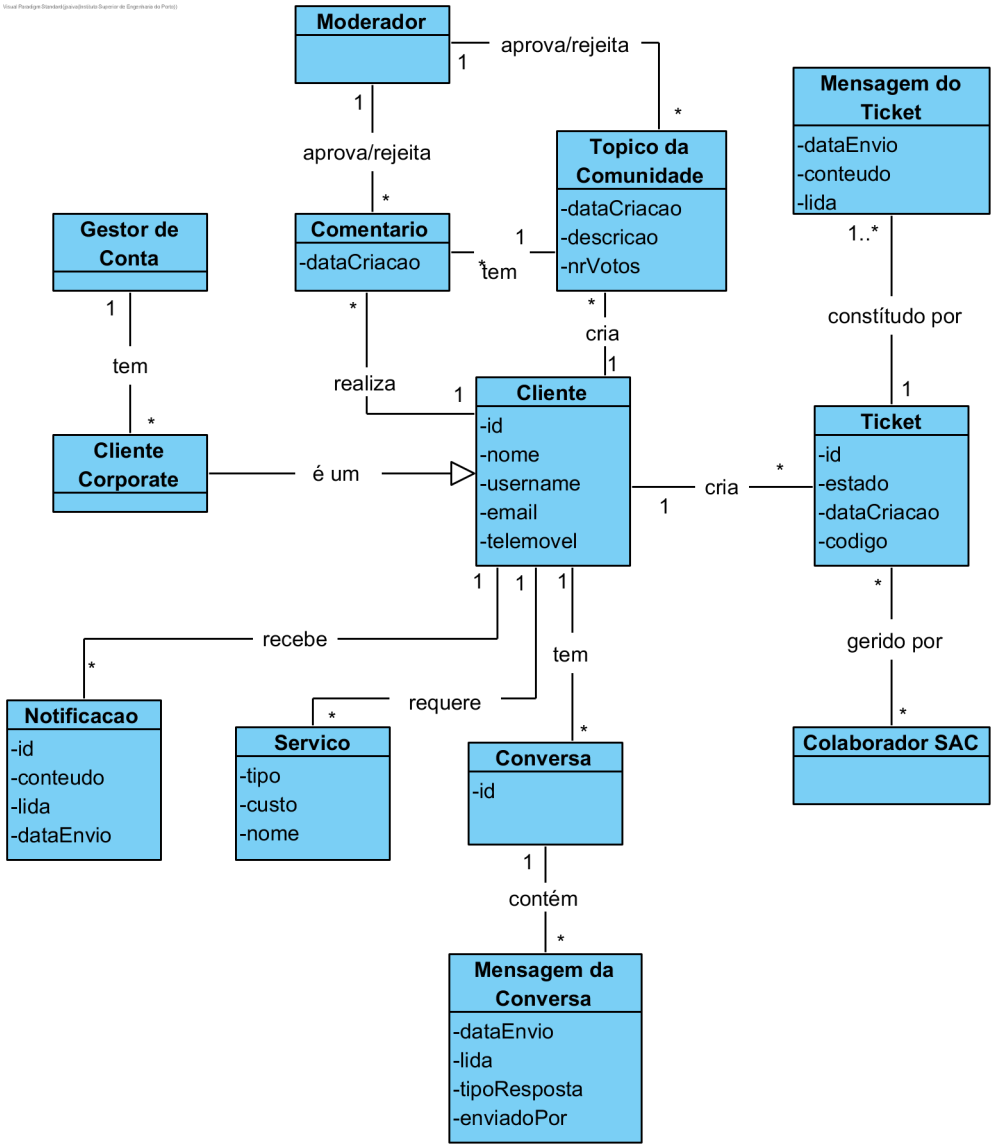


Figura 5.1: Modelo de Domínio

5.2 Vista Física

Esta secção descreve todos os projetos necessários para a realização do projeto principal (Widget-Help), os *containers* na camada física e as conexões entre eles. É apresentada uma descrição de cada um deles, e qual o seu objetivo, de forma a possibilitar o desenvolvimento do projeto desta dissertação.

Traduzem-se, essencialmente, em projetos individuais (internos e externos) que foram interligados, em camadas, sendo que a camada de apresentação, isto é, aquilo que o utilizador vê, é apenas através do projeto Widget-Help.

5.2.1 Descrição dos Containers

De modo a exibir os diversos projetos da E-goi, descreve-se de forma breve cada um dos mesmos:

Widget-Help

Começando pelo projeto que segue os modelos apresentados em 5.6, o Widget-Help é um projeto de *front-end*, isto é, foi desenvolvido em Angular 9, com recurso à biblioteca NgRx - tecnologia que permite fazer a gestão do estado de uma aplicação de forma reativa, baseando-se na biblioteca Redux[1].

Este projeto apresenta a fachada de interação com os projetos descritos nas subsecções seguintes. É responsável por fazer a gestão dos estados do *widget*, abertura/término, abandono de conversas, comunicação com o projeto do Botpress via *Application Programming Interface* (API), comunicação com o projeto do LiveAgent e Comunidade via Javascript API - *IFrame* - e, por fim, com os Services via API.

Comunidade

A Comunidade, projeto já muito antigo na E-goi, desenvolvido na linguagem de *Hypertext Preprocessor* (PHP)[2] e algum Javascript, é um monolítico que contém as camadas de negócio, acesso de dados e interface de utilizador, agrupadas num único sítio. Este projeto exibe um fórum que permite aos utilizadores criar publicações com sugestões de ideias/tópicos para mais tarde serem implementados pela E-goi, uma vez aprovadas/os pelos gestores e caso se enquadrem na plataforma. Possibilita também que os utilizadores comentem, votem em sugestões comuns e sigam o estado de uma ideia.

O intuito deste projeto é envolver qualquer pessoa que utilize a E-goi, dando a oportunidade de existir uma mudança em funções existentes/novas, que possam aproximar-se mais das necessidades de quem as utiliza diariamente e tenha preferência que funcionem doutra forma.

LiveAgent

Serviço de terceiros de suporte multi canal de apoio ao cliente, que contém toda a lógica de suporte a plataformas via *chat*, chamada ou *tickets*. Esta plataforma é configurável através de um painel num domínio privado, desde o botão de início de conversa, até ao próprio *chat* de interação com o departamento de Serviço Ao Cliente (SAC) da E-goi.

Como mencionado nos requisitos de implementação em 4.2.2, é fundamental migrar toda a lógica descrita no parágrafo anterior, mantendo todas as funcionalidades, através de um *iframe* ou alternativamente através de API.

Botpress

Projeto onde está alojada toda a lógica conversacional, identificação *Natural Language Processing* (NLP), pesquisa na base de conhecimento, fluxo de conversa, etc. Este projeto tem uma interface de desenvolvimento que, apesar de ser muito limitada pode ser extendida. Através de código JavaScript é possível realizar tarefas customizadas/extra que sejam necessárias. Esta tecnologia é *open source* e pode ser utilizada correndo um servidor através de um executável, desenvolvido em NodeJS.

Por defeito, tem uma API *out-of-the-box* muito simples, que permite efetuar pedidos *HyperText Transfer Protocol* (HTTP) *Representational State Transfer* (REST), para troca de mensagens com o *bot*, iniciando o fluxo de conversação.

Admin

[1] https://redux.js.org/
[2] https://www.php.net/

É um projeto interno, desenvolvido em PHP, maioritariamente utilizado para consulta de listagens, como por exemplo, das pesquisas efetuadas no Chatgoi (antigo mecanismo de pesquisa), informações sobre os utilizadores e as traduções existentes feitas ou necessárias fazer (por motivos de privacidade, apenas são indicadas as funcionalidades necessárias/úteis ao desenvolvimento do projeto). De notar, que este projeto contém muito mais do que apenas estas três.

Services

Servidor de grande escala, que suporta uma grande quantidade de pedidos disponibilizados através de uma API. É um projeto desenvolvido em PHP, seguindo a estrutura definida pela Zend Framework[3], utiliza o padrão *Model View Controller* (MVC), sem utilizar a "View" (uma vez que existem outros projetos com a interface definida). Comunica não apenas com uma, mas várias bases de dados *Structured Query Language* (SQL), acedendo à camada de dados, de forma a garantir a segurança e controlo de acesso aos mesmos.

ElasticSearch

O ElasticSearch é um motor de pesquisa *multitenant* (arquitetura de software em que uma única instância de software corre num servidor), criado a partir da biblioteca Apache Lucene [4], e permite pesquisar documentos de forma eficiente. Disponibiliza diversas API's para o *Create Read Update and Delete* (CRUD) de documentos, gestão dos *indexes* (índices de separação de documentos) e respetivos *templates*, pesquisa de resultados através de *querys*, entre muitas outras funcionalidades.

Essencialmente, traduz-se numa base de dados que replica os artigos registados no *helpdesk* (LiveAgent), através de um *cron job* (ver secção 5.3 parte do Services). Assim, estes são devolvidos ao utilizador de forma rápida, através da utilização do projeto Services, que integra todos os pedidos ao ElasticSearch.

5.2.2 Diagrama de Implantação (Nível 2)

Um diagrama de implantação serve como um modelo da arquitetura física de um sistema e demonstra as relações entre os componentes de software e hardware em que o sistema opera [41]. Tipicamente este tipo de diagramas é utilizado para mostrar o conjunto de nós de um sistema distribuído, os artefactos de cada nó e os componentes ou outros elementos que os artefactos implementam.

Tendo-se estabelecido previamente quais os projetos necessários para a implementação do sistema, descrevendo-os brevemente, torna-se fundamental apresentar qual a relação entre eles e de que forma estes comunicam, através do diagrama em *Unified Modeling Language* (UML) representado na figura 5.2. Este diagrama apresenta o sistema numa perspetiva de vista de alto nível, ou seja, de forma mais abstrata, focando-se no sistema como um todo, sendo apresentado nos capítulos seguintes uma vista de mais baixo nível.

[3] https://framework.zend.com/
[4] https://lucene.apache.org/

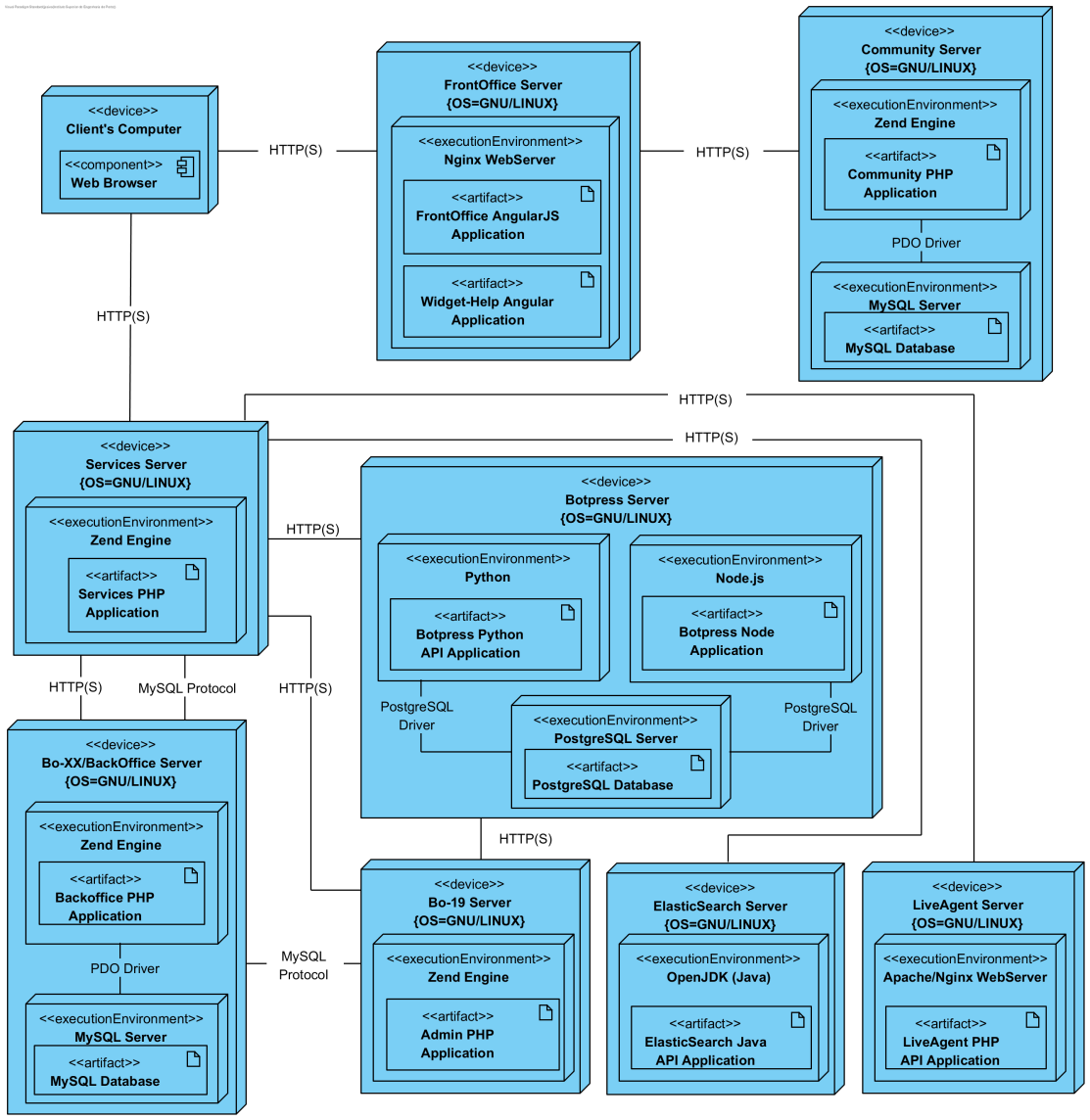


Figura 5.2: Diagrama de implantação (Nível 2)

Antes de mais, é fundamental apresentar as considerações tomadas na apresentação deste diagrama:

- todas as aplicações e bases de dados, consideram-se como artefactos do sistema, estando por isso apresentadas com o estereótipo "*«artifact»*";
- os nós apresentados com "*«executionEnvironment»*" representam qual o ambiente de execução dos artefactos presentes nesse servidor;
- todos os servidores correm sob o sistema operativo GNU/Linux, isto é, correm em Unix e a comunicação entre todos os servidores, incluindo a comunicação realizada com o cliente, é feita tipicamente através do protocolo HTTP(S) REST;
- embora esteja apresentada a comunicação entre o dispositivo do cliente e o servidor do Services, quem processa estes pedidos são as aplicações que estão no FrontOffice. Foi apresentado desta forma, porque os ficheiros que estão no FrontOffice, correm, de

facto, na máquina do cliente, isto é, são servidos estaticamente (correm no *browser* do cliente);

- todos os servidores são proprietários da E-goi;
- por fim, é importante também mencionar que, seria, de todo, impossível representar todos os servidores da E-goi num único diagrama e, por esse facto, apresentam-se apenas os que são fulcrais ao desenvolvimento deste projeto.

Começando por descrever as ligações ao Services, é possível observar que é o nó que processa todos os pedidos das aplicações do FrontOffice (é a ponte de ligação entre o Widget-Help e o Botpress), Backoffice e todos os outros servidores, com exceção da comunidade. Embora a base de dados esteja noutro servidor (no Backoffice), este servidor comunica diretamente com a mesma, através do protocolo MySQL (conexão direta via TCP/IP).

O servidor do Botpress, como descrito anteriormente, apenas tem uma API muito simples para comunicação com o *chatbot*. Como tal, torna-se fundamental o desenvolvimento de uma nova API, que permita aceder à base de dados desta aplicação, para que possam ser apresentadas, no projeto do Admin, as pesquisas efetuadas pelos utilizadores, os resultados das respostas obtidas e a utilidade da informação devolvida. O nó da esquerda representa portanto, essa mesma API, que apenas interage com a base de dados e devolve informação da mesma.

O FrontOffice contém as duas grandes aplicações da camada de apresentação ao cliente: a fundamental a ser desenvolvida que é a do Widget-Help e uma outra, intitulada de "FrontOffice Application", que apresentará o Widget como um *iframe* (não será apresentada a segunda, uma vez que serve apenas o propósito de apresentar o Widget-Help). A aplicação da comunidade era apresentada através da aplicação do FrontOffice, sendo que agora é mostrada através do Widget-Help. O cliente visualiza-a portanto, através da interface do Widget-Help, também como um *iframe*.

O Bo-XX/BackOffice representa uma instância de um servidor, isto é, um servidor que tem uma aplicação e uma base de dados própria, e que é replicado inúmeras vezes, por razões de escalabilidade e *fail-safe*. Apenas se apresenta um por questões de simplicidade, e para mostrar onde está localizada a base de dados utilizada pelo Services e pelo Bo-19.

Quanto ao Bo-19, este servidor é recente e apresenta o projeto interno do Admin, que lê diretamente, tal como o Services, da instância da base de dados do Backoffice através do protocolo MySQL. Está ligado ao servidor do Botpress uma vez que acede também, de forma direta, à API de consulta da informação de base de dados do mesmo.

Os servidores do ElasticSearch e o LiveAgent, são ambos servidores que contêm uma API cada, que é integrada diretamente no Services, para consulta de artigos e gestão de ajuda, respetivamente. Para reforçar a ideia, o Services serve como uma ponte, entre o LiveAgent e o ElasticSearch (busca de artigos do LiveAgent e inserção no ElasticSearch), e também entre o Widget-Help e o LiveAgent (criação e listagem de tickets, chamadas ao suporte, etc). Uma vez que o Widget-Help não pode relacionar-se diretamente com este serviço, por motivos de segurança (chaves API), utiliza-se o Services como *server-in-the-middle*.

5.3 Vista de Desenvolvimento

Os diagramas de componentes demonstram a estrutura de um sistema de software, descrevendo os componentes de software, as suas interfaces e as suas dependências [42]. Com a

secção anterior pretendeu-se demonstrar uma vista de alto nível do sistema. Agora, com o diagrama da figura 5.3 e diagramas subsequentes, o objetivo passa por interiorizar a forma como estão e devem ser estruturados os projetos a utilizar, apresentando portanto um diagrama de nível 3 da vista de desenvolvimento.

Alternativamente, este diagrama poderá ser visto como de nível 2 do modelo C4, uma vez que, ao invés de ser mostrado apenas um *container* indívidual com os componentes específicos, optou-se por englobar, por exemplo, todos os *models* num único *component*, todos os *services* num único *component*, etc. Isto porque, o modo de funcionamento para qualquer caso de uso, segue por norma o fluxo apresentado.

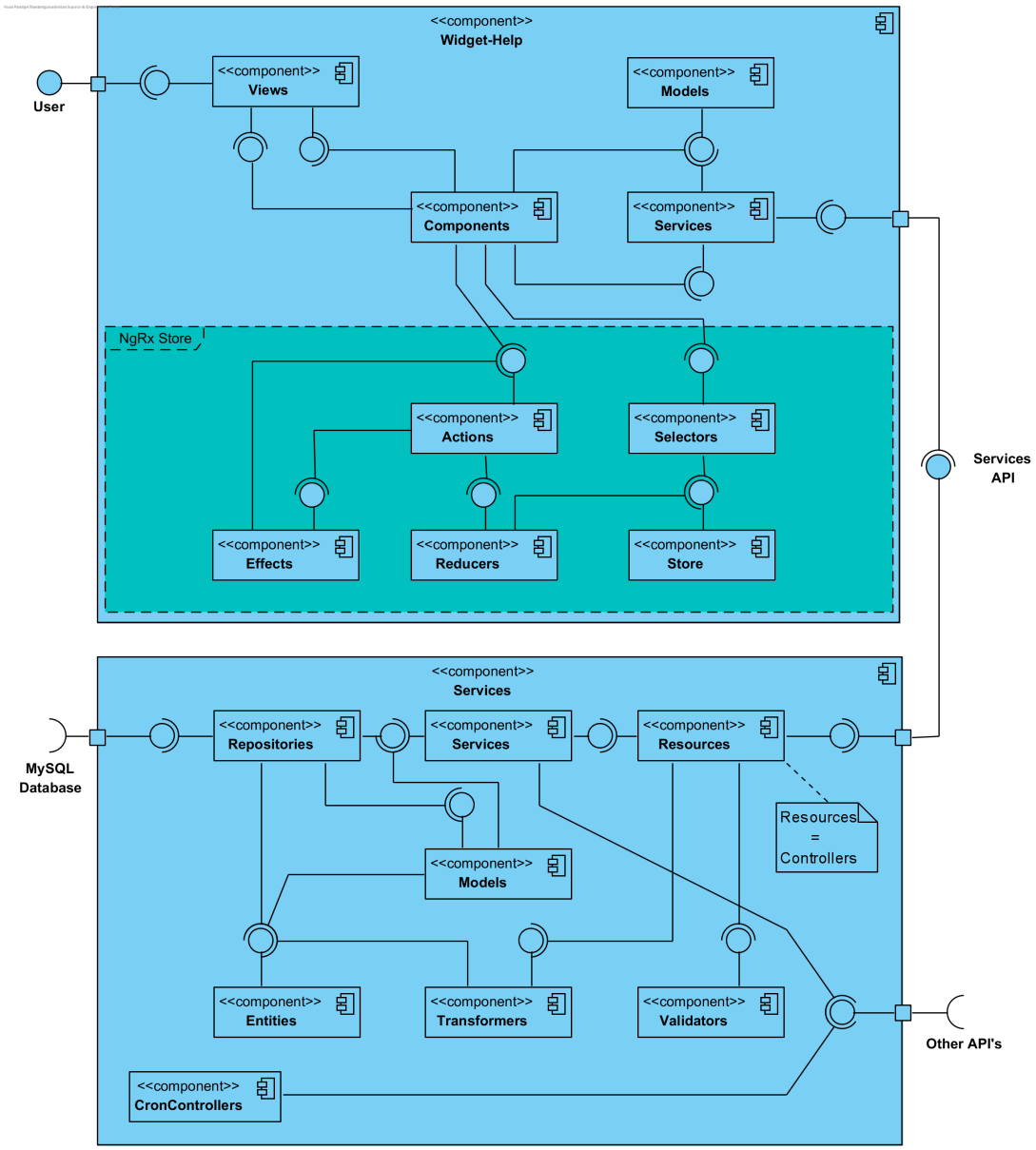


Figura 5.3: Diagrama de componentes de interação entre o Widget-Help e Services (Nível 3)

Os dois grandes componentes presentes no sistema apresentado na imagem 5.3 são, de cima para baixo, o projeto do Widget-Help e o projeto do Services, respetivamente. Estes

comunicam entre eles, através da interface (do lado direito), fornecida pelo Services e podem ser considerados o *Front-End* e o *Back-End* da aplicação final. Os restantes projetos, à exceção da Comunidade que segue uma estrutura semelhante ao Services e é um projeto desenvolvido pela E-goi, não serão apresentados na vista de componentes (por serem tecnologias externas).

O componente de cima utiliza a *framework* de Angular, o que implica o uso de um modelo arquitetural baseado em componentes. Estes podem ser vistos como controladores e a sua função é processar os dados e efetuar o *display* na *View*. A *View* por sua vez pode aceder aos dados do controlador e decidir como os deve apresentar, sendo esta comunicação bi-direcional. Existe também o conceito de *"Model"*, onde são definidas as classes deste sistema e os serviços onde são colocados os pedidos a serviços externos a este projeto. Os componentes comunicam com estes serviços para efetuar pedidos e atualizar os modelos.

A parte inferior do diagrama do Widget-Help com o título NgRx Store[5], introduz um novo princípio baseado em *Redux*[6]. A NgRx ou "estado reactivo para Angular", pode ser melhor compreendida através das seguintes alíneas:

- A *Store* é um "contentor" responsável por guardar o estado global de uma aplicação (*State*) e que é acessível em qualquer parte (através dos *Selectors*) - funções utilizadas para "selecionar" partes do estado;
- Sempre que há uma mudança no *State*, os *Reducers* são responsáveis por pegar no estado atual e na última ação, e atualizá-lo ou criar um novo estado;
- As *actions* são eventos únicos que são enviados a partir dos componentes;
- Por fim, os *effects* são efeitos colaterais sobre os quais, os componentes, não precisam de ter conhecimento (normalmente são serviços de longo processamento que observam todas as ações da *Store*).

Quanto ao componente representado abaixo (o projeto Services), segue uma implementação MVC, segundo a Zend Framework[7] (sendo que a *View* é apresentada através do Swagger).

Os *Resources* são o ponto de entrada de todos os pedidos direcionados a esta API e extendem um controlador abstrato que, define um conjunto de métodos CRUD, que retornam "Método não permitido" caso o *Resource* não contenha a sua própria implementação do método a ser chamado. Interagem com os *Validators* e *Transformers*, para validar o *input* e transformar os dados caso seja necessário, respetivamente.

Os *Validators* produzem um resultado na forma de *boolean*, garantindo que os dados são ou não válidos. Se o *input* não for correto, podem ainda devolver informação sobre quais dos campos não estão de acordo com os requisitos.

Quanto aos *Transformers*, estes são objetos ou *callbacks*, que têm o conhecimento de como mostrar os dados. Para evitar a duplicação de código e otimizar a transformação de dados que ocorre sempre da mesma forma, esta classe contém o método *transform* que pode ser invocado a partir dos *Resources*.

Os *Services* podem ser acedidos através de uma *Factory* que quando invocada, instância o *Repository* passando-o como parâmetro na instanciação do *Service* e retornando-o para o

[5] https://ngrx.io/guide/store
[6] https://redux.js.org/
[7] https://framework.zend.com/manual/1.12/en/introduction.overview.html

Resource, para ser utilizado. Assim, os *Repositories* são diretamente injetados na instância do serviço e permitem aos *Resources* uma comunicação *Object Relational Mapping* (ORM) com a base de dados, apenas através do serviço.

As *Entities* representam, como seria de esperar, as entidades da base de dados, com cada campo associado a uma coluna e com os respetivos gets e sets. Já os *Models*, apresentam métodos que efetuam operações CRUD, através de pedidos HTTP, a outras aplicações da E-goi.

Por fim, os *CronControllers* são controladores especiais, porque são especificamente realizados com métodos que podem ser corridos através da linha de comandos (*cron tasks*), sendo normalmente utilizados para realizar operações automatizadas, por exemplo diariamente.

De notar que existem mais padrões/classes utilizados neste projeto, mas que não serão apresentados por não serem relevantes ao desenvolvimento da aplicação final e por motivos de simplicidade dos diagramas apresentados.

5.4 Vista de Processo

A vista de processo apresenta os processos do sistema, como eles comunicam e o seu modo de execução. Para tal, utilizam-se diagramas de sequência que descrevem os casos de uso ou requisitos funcionais mais complexos e relevantes, apresentados no capítulo 4.

Esta secção pretende abordar processualmente os níveis 2 e 3 do modelo C4, começando por uma apresentação mais abstrata do sistema (nível 2), entrando de seguida em detalhe (nível 3), em alguns dos *Use Case* (UC) apresentados.

5.4.1 Apresentação do UC-10

A figura 5.4 expressa o modo como um cliente, no UC-10 (descrito brevemente em 4.2.1), interage com o sistema quando pretende efetuar uma pesquisa. Existem cinco *"life-lines"* respetivas aos projetos/componentes apresentados anteriormente, que mostram os pedidos HTTP e *SQL* que devem existir por trás do Widget-Help. Do lado direito do diagrama é possível também observar que, está presente o UC19 - Persistência das pesquisas efetuadas.

Sempre que o utilizador efetua o envio de uma mensagem, esta é processada pelo Services primeiro e só depois pelo Botpress, para garantir que é fidedigna (garantindo também que os dados do cliente são mais tarde acessíveis, para registo na base de dados). Depois de enviada ao Botpress, utilizando o *id* de utilizador como identificador da conversa, se o tipo de mensagem for uma questão, esta é enviada ao Services para efetuar a pesquisa na base de conhecimento (ElasticSearch), retornando os artigos relacionados com a pergunta realizada.

As duas notas apresentadas no diagrama visam descrever o modo como os dados são guardados na base de dados do Botpress, utilizando um *id* de evento (único) gerado por esta ferramenta, que mais tarde pode ser utilizado para modificar os valores que foram guardados. O propósito do armazenamento destes dados é para que, mais tarde no UC-23 (Consulta de pesquisas), possam ser vistas as pesquisas efetuadas, os resultados retornados e a utilidade dos mesmos (a fim de efetuar melhorias no *chatbot*).

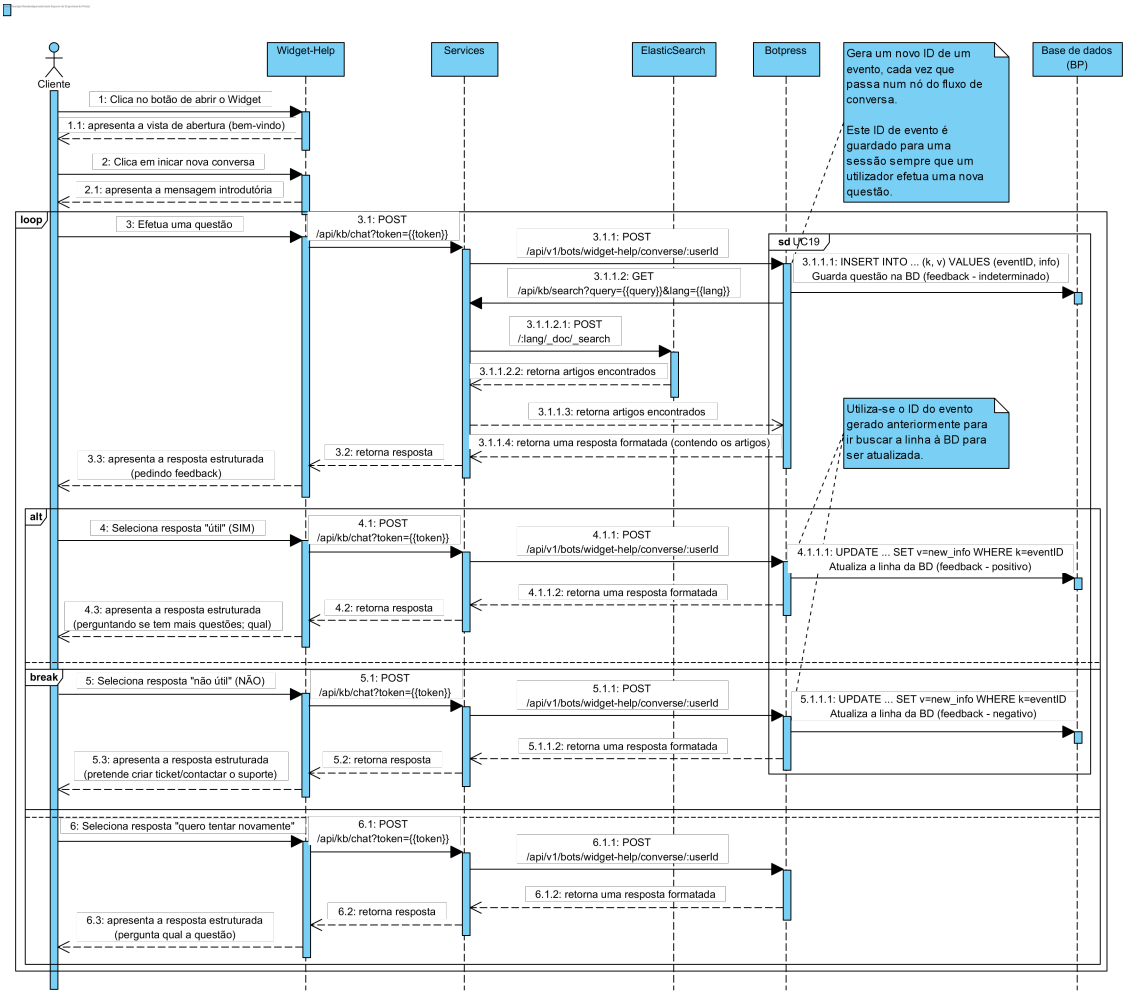


Figura 5.4: Diagrama de sequência do UC-10 (Nível 2)

O fluxo de questões efetuadas pelo utilizador é envolvido num loop, em que o utilizador efetua uma questão e são lhe dadas 3 opções de resposta (*quick-replies*), para a pergunta "Foram úteis os artigos", respetivamente: "Sim", "Não" e "Quero tentar novamente". No caso da primeira e última resposta, são devolvidos novos resultados de resposta a questão efetuada. Caso a resposta seja "Não", então é dada a opção ao utilizador de contactar o suporte (UC-6) ou criar um *ticket* (UC-2).

5.4.2 Apresentação do UC-20

Sendo este *chatbot* de ajuda, um mecanismo que, à partida, deve funcionar a qualquer momento e atempadamente, considera-se a síntese do UC-20 como uma das mais importantes. Este caso de uso traduz-se essencialmente, em alertar os adminstradores de sistemas de que existem falhas de comunicação entre os *containers* e deve existir uma reparação assim que possível (ver figura 5.5), para que tudo volte a funcionar normalmente.

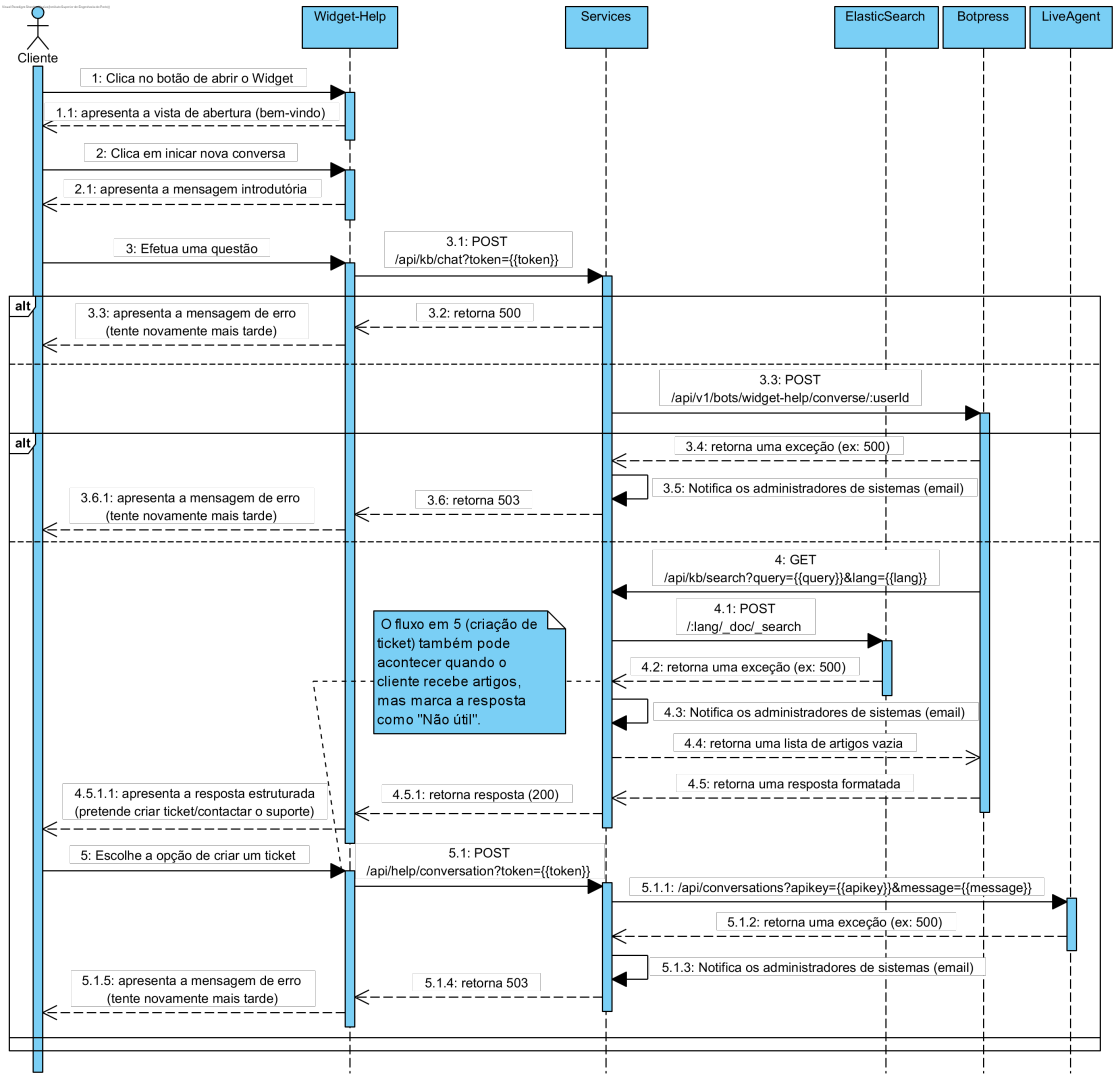


Figura 5.5: Diagrama de sequência do UC-20 (Nível 2)

O processo exposto é semelhante ao da figura 5.4, no sentido em que o cliente começa por se dirigir à parte de comunicação com o *chatbot*, colocando uma questão. De seguida, são apresentadas várias alternativas do que pode suceder, caso alguma das outras *life-lines* se encontre indisponível.

Na primeira alternativa, sendo esta a mais direta possível e o pior caso (pois é o componente que processa todos os pedidos), se o servidor do Services se encontrar indisponível, na grande parte dos casos é retornado um erro 500 (erro interno de sistema) e apresentado ao cliente uma mensagem de "tente novamente mais tarde".

Na parte inferior (segunda alternativa), é subdivida em duas, originando-se o problema no servidor do Botpress ou ElasticSearch. Caso algum destes esteja impossibilitado de dar resposta ou o formato da resposta tenha mudado (por serem serviços externos, poderá surgir este último caso), é alertada a equipa de sistemas.

Na subdivisão da segunda alternativa existe ainda a possibilidade de o utilizador: receber uma lista de artigos vazia (porque não foram encontrados ou o servidor do ElasticSearch não se encontrar disponível) ou então receber uma lista de artigos não relevantes à pesquisa

efetuada. Se esta ocorrência levar à criação de um *ticket*, mas o serviço do LiveAgent se encontrar indisponível é também lançado um alerta para a equipa de sistemas.

De notar que, a opção de criação de uma chamada ao suporte (UC-6) não foi incluída, porque a forma de comunicação com o LiveAgent não é realizada através do Services, mas sim diretamente pela JavaScript API disponibilizada por esse serviço. Ou seja, não atravessando o *endpoint* do Services, não há possibilidade de ser gerado um alerta, apenas uma mensagem de aviso ao utilizador para tentar mais tarde.

5.4.3 Apresentação do UC-23

Um dos grandes requisitos da E-goi é também poder observar as pesquisas feitas pelos clientes, para que o *chatbot* possa ser atualizado/melhorado, os artigos possam ser revistos/alterados/eliminados e as fontes aprimoradas. Tudo isto apenas é possível se existir alguma listagem das pesquisas para análise da equipa de *User Experience* (UX).

Na figura 5.6 são apresentadas as possibilidades do UC-23 que, além da descrição apresentada em 4.2.1, deve permitir ainda: ver a informação específica a uma pesquisa, filtrar a listagem segundo vários filtros (ex: *idCliente*, *idUtilizador*, país) e ainda possibilitar a exportação dos dados para um ficheiro *Comma-Separated Values* (CSV). Como estas funcionalidades entram mais no detalhe de implementação (nível 3) não serão apresentadas.

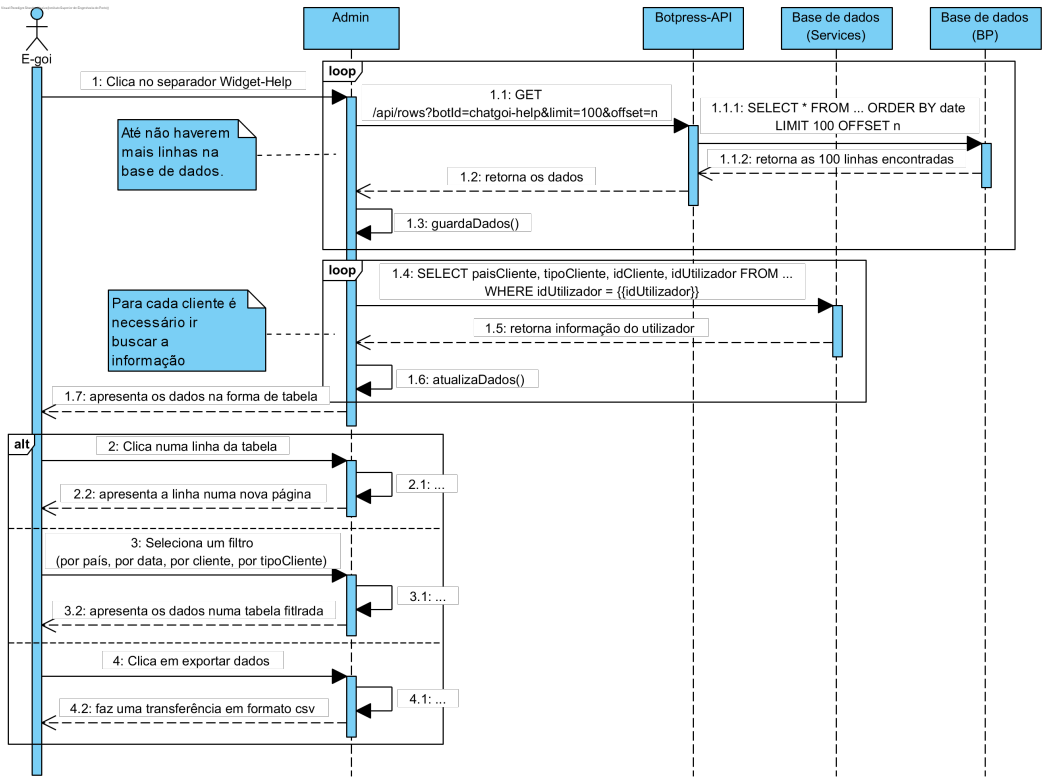


Figura 5.6: Diagrama de sequência do UC-23 (Nível 2)

Quando alguém da E-goi acede à página do separador do Wiget-Help, no projeto do Admin (com as permissões certas), são feitos sucessivos pedidos à Botpress-API (projeto criado para dar suporte aos pedidos do Admin). Como poderão existir muitos resultados, limita-se o pedido a apenas 100 resultados (número variável) de cada vez, sendo feitos vários

pedidos até não haver mais linhas a retornar. Assim que todos os dados forem carregados em memória, são atualizados com mais informação relativa aos clientes, que não consta na base de dados do Botpress, mas sim na do Services.

No fim, é apresentada sob a forma de tabela todas as pesquisas efetuadas, identificando qual o utilizador, o país de origem, a data da pesquisa, os artigos devolvidos, a utilidade da resposta (dada pelo utilizador), etc.

5.4.4 Apresentação do UC-5

O UC-3 e UC-4 são dois casos de uso semelhantes, pois têm o intuito de fazer uma listagem, de *tickets* e de notificações, respetivamente. Isto pode ser feito num único pedido à API do Services, havendo apenas distinção na forma como são listados e no que acontece, pós-clique num dos elementos da listagem, no caso dos *tickets* (através do UC-5).

De tal modo, apresenta-se na figura 5.7 o UC-5, englobando o UC-3 e o UC-2 (criação de *tickets* pelo *chatbot*), sendo que o UC-2 segue um fluxo semelhante ao que foi apresentado na figura 5.5 (no fundo da imagem), embora no caso em que não ocorre uma exceção, o *id* do *ticket* criado é devolvido na mensagem.

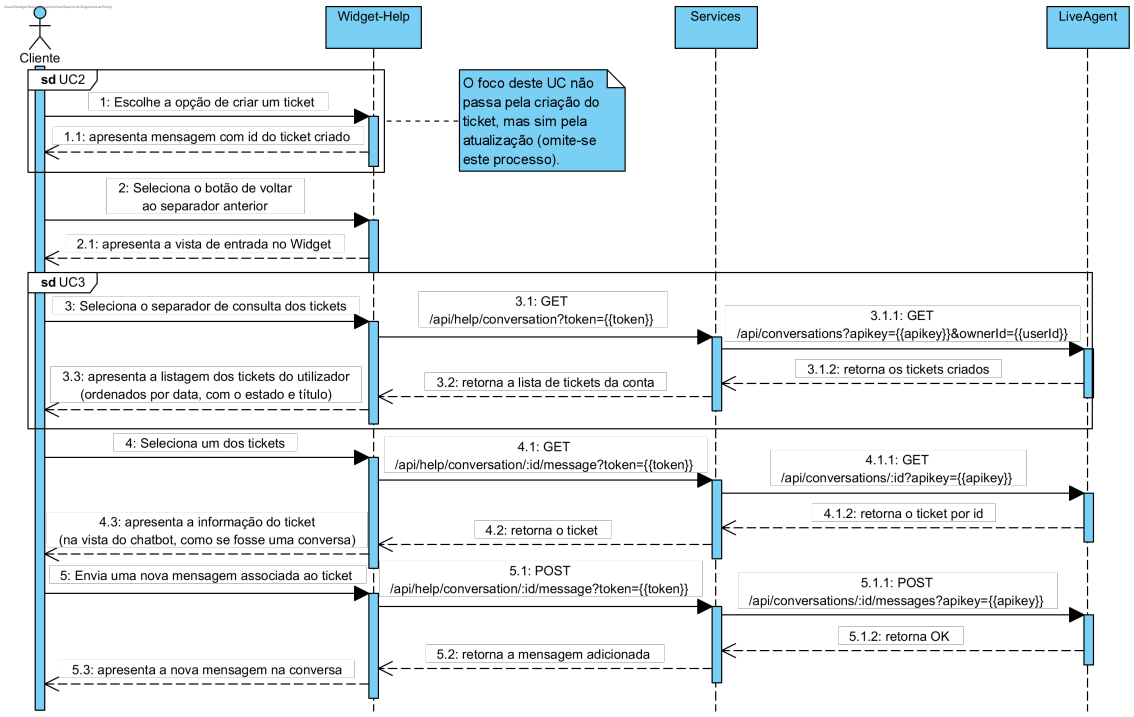


Figura 5.7: Diagrama de sequência do UC-5 (Nível 2)

Assim que o utilizador tem *tickets* criados, estes são apresentados ordenadamente por data, no separador de consulta dos mesmos (UC-3), com o estado associado (ex: Aberto/Fechado/Respondido...), parte do assunto (um *preview*), o *ID* e a data, também da última mensagem. Selecionando um, é realizado um pedido ao Services para devolver a informação relativa ao *ID* do *ticket* escolhido, sendo apresentado na mesma vista em que foi criado, isto é, como se fosse uma conversa entra o utilizador e o *chatbot* (só que neste caso, o suporte).

Nesta vista de conversa, o utilizador pode contactar diretamente o suporte, em vez de fazer questões ao *chatbot*, sendo realizados pedidos POST ao services, que se refere à API do LiveAgent para atualizar o mesmo.

5.5 Apresentação do UC-4

Com o intuito de apresentar uma vista de mais baixo nível (nível 3 do modelo C4), selecionou-se o caso de uso 4, porque, como este tipo de vista apresenta (mais detalhadamente) o modo de funcionamento do Widget-Help, os casos de uso anteriores teriam uma enorme quantidade de pedidos a apresentar. Para sintetizar a implementação e o modo de funcionamento da biblioteca NgRx (em conjunto com Angular), o diagrama da figura 5.8 demonstra a forma como são guardados/acedidos/atualizados os dados na *Store* (que deve ser semelhante para os casos de uso relativos a este projeto) e engloba o caso de uso UC-7 (*update* de notificações como "lidas").

As notificações apenas são carregadas quando o cliente entra na E-goi, logo a seguir a autenticá-lo. Uma vez autenticado, é disparada a ação *LoadWidget*, que desencadeia o efeito de buscar as notificações da conta realizando um pedido GET à API do Services. Ainda dentro do mesmo efeito é disparada a ação de carregar as notificações para a *Store* - *LoadNotifications*.

Quando o cliente acede ao cartão das notificações, o componente acede à *Store* através de um *Selector* (getNotifications). Como este método retorna um *Observable* (porque é assíncrono) é necessário subscrever ao mesmo, para quando os dados forem retornados serem processados. Assim que os dados são carregados para o componente, as notificações são apresentadas na *View*.

Depois, no UC-7, quando o utilizador seleciona uma notificação, caso esta ainda não tenha sido marcada como lida, o componente faz um pedido ao serviço (que por sua vez faz um pedido ao *backend*) para que a mesma seja atualizada (na base de dados). Contudo, isto não é suficiente, pois não é atualizada a *Store*. Então, o componente dispara, de seguida, uma ação (*UpdateNotification*) que atualiza o estado da notificação respetiva, guardada na *Store*.

Alternativamente, em vez de ser o *Component* a invocar o método do serviço (diretamente) e depois realizar a atualização da *Store* através duma *Action*, poderia ser criado um novo *Effect* que, após o cliente abrir uma notificação, aquando da atualização da *Store* invoca o *Service* (ver figura 5.9). Em suma, em vez de ser o componente a chamar o serviço, passaria a ser um efeito colateral da atualização da notificação na *Store*.

De notar que a figura 5.9 apenas apresenta a parte do loop da figura 5.8, sendo que o resto (a parte superior do diagrama) se mantém igual.

5.6 Mockup da Interface Gráfica

Através do trabalho realizado pelos *designers* da E-goi, existia já uma interface gráfica visual (não implementada), isto é, um esboço do que era pretendido como apresentação da solução, como guia do projeto a ser desenvolvido, elaborada com recurso às ferramentas Sketch[8] e

[8] https://www.sketch.com/

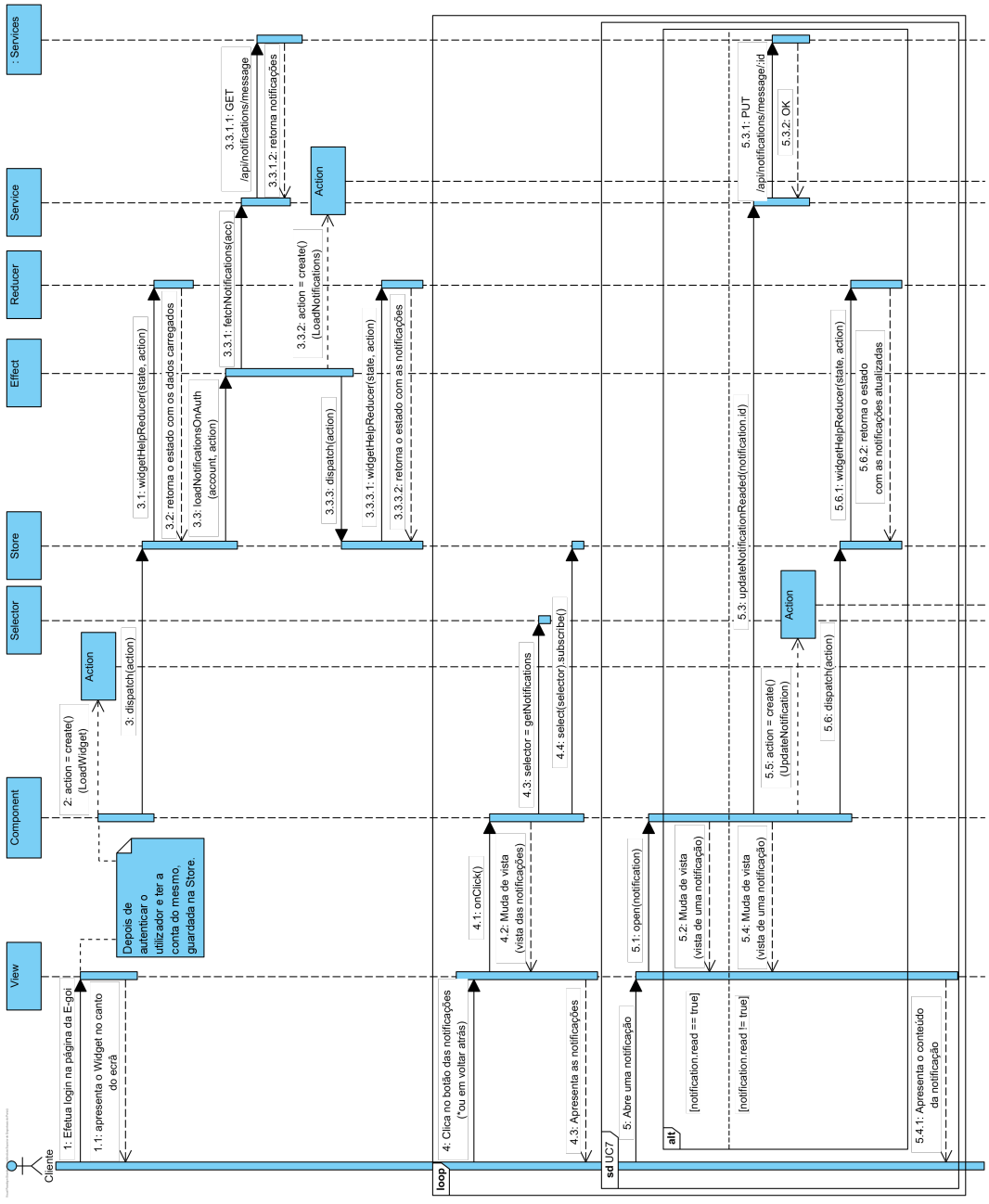


Figura 5.8: Diagrama de sequência do UC-4 (Nível 3)

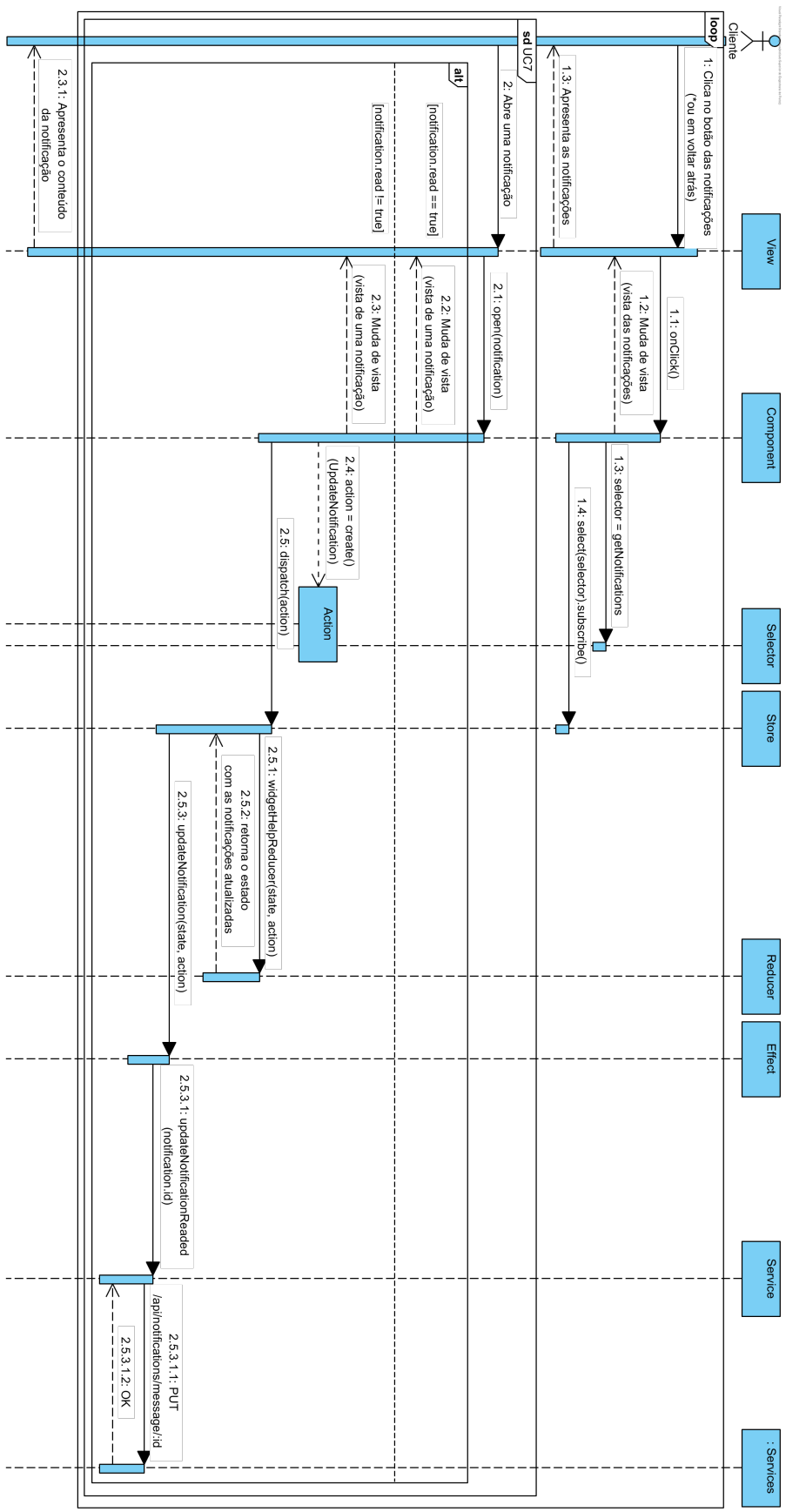


Figura 5.9: Alternativa ao diagrama de sequência do UC-4 (Nível 3)

InVision[9] - *toolkit* para desenho e criação de produtos digitais e, ferramenta para entrega e partilha visual e interativa desses mesmos desenhos, respetivamente.

Esta apresentação visual é nada mais que um esboço/guia, para facilitar o desenvolvimento, descrevendo o que é pretendido e a estrutura que deve ser seguida.

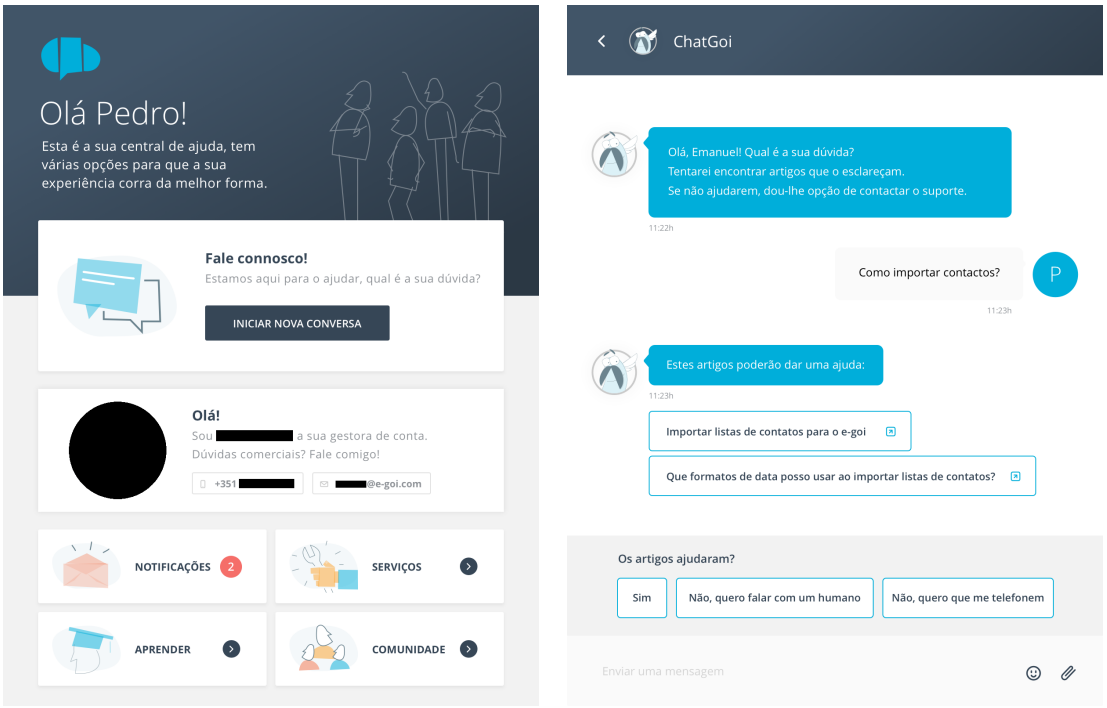


Figura 5.10: Visão inicial gráfica para o Widget-Help, incluindo a página de conversa com o chatbot

A visão inicial, na figura 5.10 do lado esquerdo, apresenta à partida uma quantidade de funcionalidades (pelo menos cinco) que distinguem o *widget-help* de todos as outras plataformas que foram analisadas no capítulo 2, por ser muito mais do que um simples *chatbot*. Além do que está representado na figura, existe também ainda a possibilidade de haver um histórico de pelo menos as duas conversas mais recentes entre o utilizador e o *chatbot*. Do lado direito é representado apenas um fluxo de conversa exemplar entre o utilizador e o sistema.

[9] https://www.invisionapp.com/

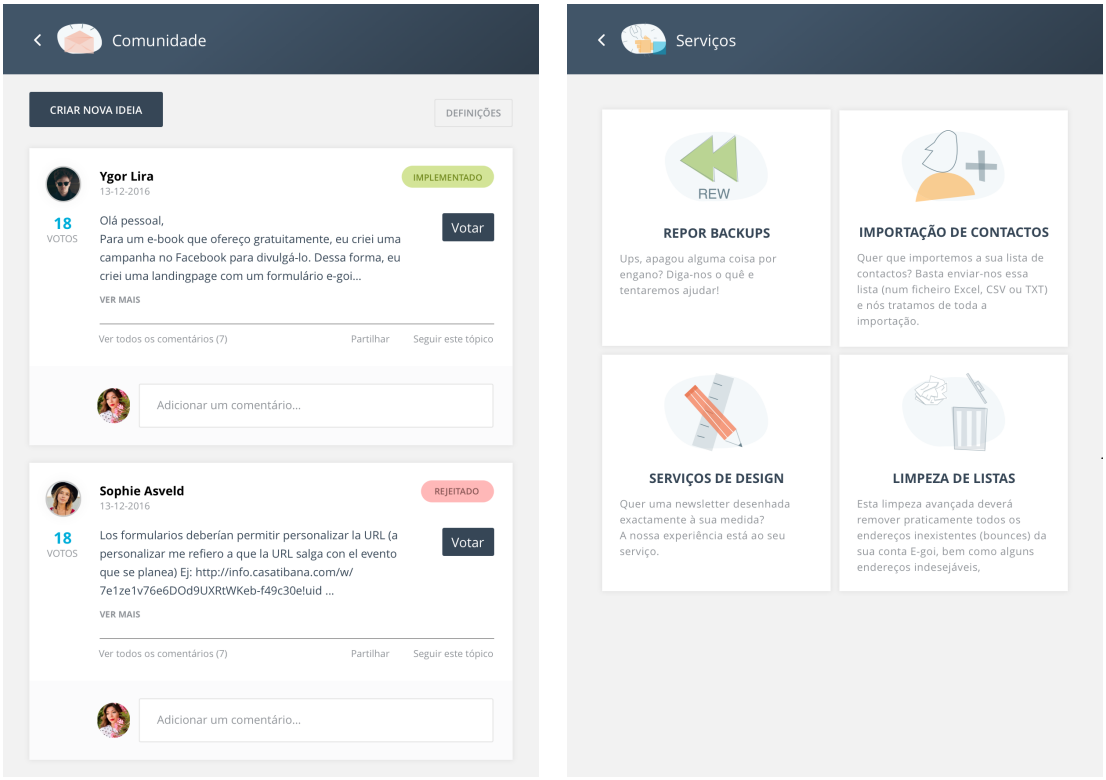


Figura 5.11: Visão da comunidade e do separador serviços

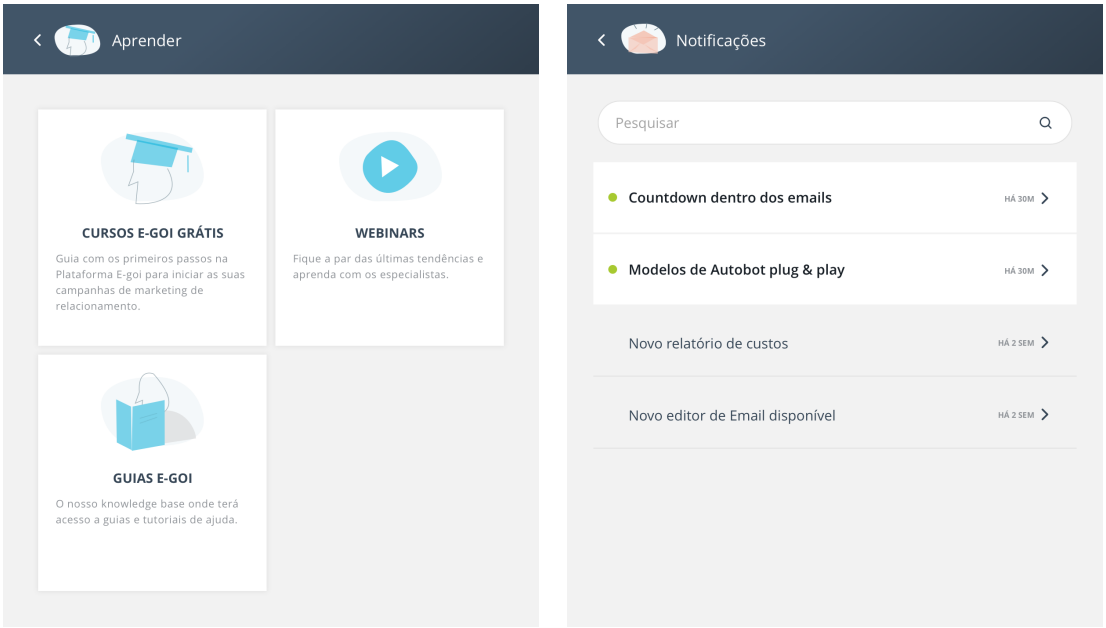


Figura 5.12: Visão do separador de aprendizagem e do separador de notificações

Como é possível observar nas figuras 5.11 e 5.12, todos os componentes do sistema foram pensados à priori, apresentando-se, respetivamente, os separadores da comunidade, serviços, aprendizagem e notificações.

O separador da comunidade integra o projeto da Comunidade, para que os utilizadores possam sugerir ideias, tópicos ou sugestões, podendo votar nelas, sendo que, as ideias mais votadas aparecem em primeiro lugar. Este fórum é controlado por um moderador, em que cada comentário ou mesmo publicação, é submetido a uma revisão. Após aprovado/rejeitado é mudado o estado público do mesmo para que todos possam tomar conhecimento dos tópicos implementados.

O dos serviços apresenta várias solicitações que o cliente pode efetuar, por uma taxa associada. São intitulados de serviços à medida e cada um contém um formulário parametrizado, que após o preenchimento é enviado para o suporte da E-goi, a fim de ser avaliado.

O separador de aprendizagem contém um conjunto de cartões, com links de ajuda e o das notificações apresenta uma listagem com todos os avisos enviados ao utilizador, a data de envio e o estado de abertura (se foi vista ou não).

Capítulo 6

Implementação

Após toda a análise e planeamento, torna-se fundamental iniciar o processo de implementação de algo concreto, que será depois avaliado e mantido futuramente. Este capítulo começa por descrever o contexto organizacional, de desenvolvimento do projeto e os/as procedimentos/metodologias, e o processo de desenvolvimento do projeto Widget-Help, Botpress e restantes projetos, entrando no detalhe de implementação, o último nível do modelo C4 (nível 4 - código).

6.1 Ferramentas e Metodologias de Desenvolvimento

A E-goi adota uma estrutura hierárquica horizontal, dividindo a organização em vários departamentos, um deles identificado por diversas vezes, anteriormente, como o departamento do Serviço Ao Cliente (SAC) e outro (relevante para esta dissertação), que é o de Engenharia do Produto. Este último contém várias equipas especializadas/dedicadas ao desenvolvimento de um único produto, como por exemplo, Integrações, *Mobile*, *Web*, *Email*, entre muitas outras. Do conjunto de equipas que a E-goi apresenta, dado que o contexto do problema deste projeto se insere em ajudar os utilizadores e na oferta de ajuda, a equipa atribuída foi a de *User Experience and Assurance* (UXA) - talvez mais conhecida no contexto de outras empresas como equipa de *Quality Assurance* (QA).

A maior parte das equipas do departamento, utiliza várias ferramentas, importantes e comuns, como: Git[1] - ferramenta *open source* gratuita de controlo de versões -, GitLab[2] - plataforma de desenvolvimento de operações e manutenção de código para *Continuous Integration* (CI)/*Continuous Deployment* (CD) -, Jenkins[3] - ferramenta *open source* para suportar *builds*, *deploys* e automações de quaisquer projetos - e, finalmente o Jira[4] - *website* para gestão ágil de projetos permitindo executar o planeamento e controlo de tarefas.

No caso específico deste projeto e, seguindo as boas práticas de desenvolvimento de software, utilizou-se o Jira que disponibiliza uma quadro Kanban, para especificação dos requisitos funcionais descritos em 4.2.1. Este quadro é uma ferramenta de manutenção de projetos ágeis, que ajuda a visualizar as tarefas/*issues* em progresso, as que estão por concluir e as que já foram realizadas, atribuindo um valor/peso de importância a cada uma [43].

Como referido em 1.4, a abordagem utilizada pela equipa de UXA é a Agile, apoiada por reuniões semanais de discussão do que foi feito, irá ser desenvolvido e questões adicionais,

[1] https://git-scm.com/
[2] https://about.gitlab.com/
[3] https://www.jenkins.io/
[4] https://www.atlassian.com/software/jira

utilizando o Jira para efetuar os registos temporais dos requisitos referidos no parágrafo anterior.

Quanto ao GitLab, todos os projetos mencionados em 5.3 têm um repositório associado, com exceção do ElasticSearch e LiveAgent, e foram clonados para a máquina de desenvolvimento para poderem ser alterados.

A ferramenta de controlo de versões Git foi utilizada para o desenvolvimento do projeto Widget-Help, subdividindo-se em duas *branches*, um ramo de desenvolvimento (*development*) e outro onde seria colocado o código de produção (*master*). Todos os restantes projetos utilizam quatro *main branches*: a de desenvolvimento (tarefa/*issue* seria o nome da branch), de correção de bugs (*bug tracking*), de entrega (*staging*) e de produção (*master*). O fluxo seguido é: a implementação da nova funcionalidade é realizada no respetivo ramo, sendo de seguida feito um/a *merge*/junção desse ramo com o de *staging* para ser testado assim que possível; uma vez testado, é revisto por um *lead developer* para ser colocado em produção (de notar que no caso do Widget-Help todo o projeto será revisto antes de ser colocado em produção, tendo já sido realizado um *code review* inicial).

Quando o código é colocado na *branch* de *staging* (remotamente no repositório), a *pipeline* do Jenkins instala as dependências necessárias, compila o código, executa os testes e, no fim, realiza uma análise ao código através da ferramenta SonarQube[5], sendo gerado um relatório para pós-análise. Este processo é executado para um servidor de testes (*staging*), para que os operadores de QA possam verificar se o código funciona em todas, ou quase todas, as circunstâncias. Todo este processo pode ser descrito através da figura 6.1.

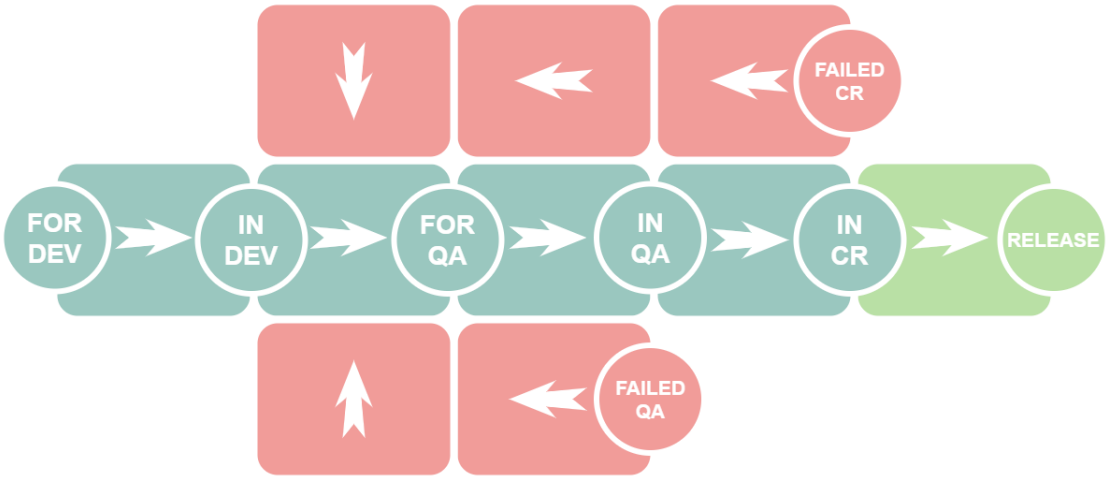


Figura 6.1: Passos do Jira para *release* para resolução de um Caso de Uso

Caso o requisito não cumpra os testes realizados em QA, é submetido para reavaliação pelo *developer* e caso o código, que cumpre o requisito, esteja mal estruturado ou contenha falhas de segurança ou seja pouco eficiente, na fase de *Code Review* (CR), deve ser também revisto novamente pelo *developer*, iniciando novamente todo o processo pela fase de desenvolvimento. Assim que atravessa a fase de CR, os *Lead developers* de cada equipa aprovam os *merge requests* para a *branch* de *release*, com a finalidade de colocar o código na máquina de produção.

[5]https://www.sonarqube.org/

De notar que, na fase de desenvolvimento, o *developer* deve registar o *work log*/tempos de trabalho na *issue* respetiva, junto com os *commits* efetuados. Além disso, deve ainda abrir um *Merge Request* (MR) para a *branch* de *release*, com a *label Work In Progress* (WIP) que deve ser removida assim que este for aprovado.

6.2 Widget-Help

No capítulo de Design em 5, apresentou-se este projeto como dividido em duas partes: a parte de Angular e a de NgRx. Seguindo a mesma ideologia, pretende-se neste subcapítulo refletir, aquilo que foi desenvolvido, sintetizando a explanação nessas duas partes fundamentais, começando pela tecnologia de Angular.

6.2.1 Tecnologia Angular

Seguindo as boas práticas de Angular, o projeto foi estruturado de forma modular, o que significa que existem vários módulos, designamente 4, que são: o módulo de raíz *app module*, o das vistas *view module*, o dos cartões *cards module* e o comum *common module*. Cada módulo contém um conjunto de componentes associados e que podem ser utilizados noutros módulos. Todos os serviços, modelos, componentes de NgRx e utils, estão na raíz do projeto e, portanto, pertencem ao *app module*.

O módulo *app* declara 3 componentes: *AppComponent*, *HelpWidgetComponent* e o *ToggleButtonComponent*. Os dois últimos são utilizados pelo primeiro e podem ser vistos como, o *widget* em si e o botão que o abre/fecha, nessa ordem.

Quanto aos restantes: o *common* contém os *pipes* (espécie de funções que são utilizadas em *HyperText Markup Language* (HTML) com o caratere "|") e componentes/vistas comuns a todos os outros módulos; o das *views* contém os componentes/vistas que apresentam os quadrados, com a imagem e o título (ver 5.6); e o último, o dos cards, representa as vistas dos cartões, quando as *views* são abertas.

Como demonstração apresenta-se a resolução ou parte da resolução do UC-3, relativo à listagem dos tickets do cliente, exibindo primeiro o componente, depois a vista e por fim o serviço associado.

Components

É através de código TypeScript que o Angular executa a lógica de processamento de dados. A figura 6.2 mostra o componente dos *tickets*, escrito em TypeScript, que é essencialmente uma classe com um decorator "*@Component*". Contém ainda um construtor onde são injetados os serviços e a *store*, que é utilizada, neste caso, para ir buscar os dados da conta e dos *tickets* no método *ngOnInit()*.

Este método é um método especial de angular que é invocado sempre que o *selector <hw-tickets>* é utilizado, sendo o ponto de entrada para qualquer *component*. É importante ainda mencionar que as variáveis declaradas dentro da classe, como é o caso do *array* de *tickets*, são acessíveis em qualquer parte da *view*.

Os restantes componentes funcionam da mesma forma e são semelhantes a esta estrutura.

```
15  @Component({
16    selector: 'hw-tickets',
17    templateUrl: './tickets.component.html',
18    styleUrls: ['./tickets.component.scss']
19  })
20  @HWView(viewNames.tickets)
21  export class TicketsComponent implements OnInit {
22    imageUrl = `${environment.widget_url}/assets/images/main-cards/widget-tickets.svg`;
23    account: Account;
24    tickets: Ticket[] = [];
25    searchFilter = '';
26    readonly replyType = ChatReplyType;
27    emptyStateMessage = this.translateService.instant('TR/HelpWidget/Views/Tickets/EmptyState');
28
29    constructor(private store: Store<WidgetHelpState>,
30      private ticketsService: TicketsService,
31      private translateService: TranslateService) {
32    }
33
34    ngOnInit(): void {
35      this.store.select(getAccountState).subscribe(account => this.account = Account.from(account));
36      this.store.select(getTickets)
37        .subscribe(tickets => this.tickets = tickets.map(t => Ticket.from(t)));
38    }
```

Figura 6.2: Excerto do componente dos *ticket component*

Views

Declarada no componente através do *templateUrl*, uma *view* está normalmente associada a um único *component*. A fig. 6.3 demonstra de que forma os *tickets* são apresentados, sendo que na linha 23 é associada a função *open()* declara no componente apresentado anteriomente, que abre o componente com o *selector "hw-ticket-card"* ou seja a vista do *ticket* selecionado.

```
1   <hw-view topBarText="{{'TR/HelpWidget/Views/Tickets/Title' | translate}}"
2     topBarImage="{{imageUrl}}">
3
4     <!-- Empty state -->
5     <div class="hw-centered hw-text-center" *ngIf="!tickets; else hasTickets">
6       <img src="{{imageUrl}}">
7       <p [innerHTML]="emptyStateMessage" ></p>
8     </div>
9
10    <ng-template #hasTickets>
11    <!-- Search bar -->
12    <mat-form-field appearance="fill" class="hw-ticket-search">
13      <mat-label>{{'TR/HelpWidget/Global/Search' | translate}}</mat-label>
14      <label>
15        <input matInput (keyup)="updateSearchFilter($event)">
16        <mat-icon>search</mat-icon>
17      </label>
18    </mat-form-field>
19
20    <!-- Tickets -->
21    <div class="hw-ticket-container">
22      <ng-container *ngFor="let ticket of tickets | filter : searchFilter : titleFromTicket">
23        <hw-ticket-card [read]="!ticket.open" [ticket]="ticket" (click)="open(ticket)">
24        </hw-ticket-card>
25      </ng-container>
26    </div>
27
28    </ng-template>
29  </hw-view>
```

Figura 6.3: *View* completa dos *tickets*

A função *updateSearchFilter()*, deteta sempre que o utilizador clicar numa tecla do teclado, filtrando os tickets apresentados na linha 22.

Services

Para terminar, os serviços são também classes, que podem ter o *decorator "@Injectable()"* para poderem ser passadas nos construtores dos componentes. A classe da figura 6.4 exibe o método *fetchTickets()*, que efetua um pedido GET ao services, ao *endpoint "/help/conversation"*, obtendo todos os tickets e ordenando-os do mais recente para o mais antigo (linha 32).

Utiliza também a função *map()* para converter o objeto *JavaScript Object Notation* (JSON) obtido num model do projeto (*Ticket* - linha 23) extraindo os campos para a propriedade correta.

```
11  @Injectable()
12  export class TicketsService {
13    static readonly initialLimit = 7;
14    static readonly conversationLimit = 20;
15    static readonly messageLimit = 160;
16    static readonly step = 5;
17    constructor(private http: HttpClient) {
18    }
19
20    fetchTickets(): Observable<Ticket[]> {
21      return this.http.get('/help/conversation').pipe(map((resp: any) => {
22        const tickets: Ticket[] = resp.items.map(item =>
23          Ticket.from({
24            id: item.conversationid,
25            title: item.code,
26            content: item.preview,
27            status: item.status,
28            time: this.parseCreatedDate(item.datecreated),
29            open: item.status !== 'R'
30          })
31        );
32        return tickets.sort((a, b) => b.time.getTime() - a.time.getTime());
33      }));
34    }
```

Figura 6.4: Excerto do serviço dos *tickets*

6.2.2 Tecnologia NgRx

Como apresentado na vista de desenvolvimento em 5.3, a biblioteca NgRx é um dos componentes presentes neste projeto e fundamental ao desenvolvimento do mesmo. Apresentam-se nesta subsecção os diversos ficheiros que integram esta biblioteca e que a que parte da mesma se associam.

Store

O conceito de *Store* pode ser visto como um objeto (fig. 6.5) que guarda o estado da aplicação e pode ser acedido em qualquer momento, em qualquer circunstância. Este "Objeto" foi estruturado da seguinte forma:

- *WidgetHelpState* - é a interface raíz e contém os objetos *widget*, *button*, *context* e *transient*;

- *WidgetState* - é responsável por guardar o estado do Widget, se este está aberto ou fechado (*open*), a vista que está aberta no momento (*activeView*) e os dados da vista atual (*viewData*);
- *ToggleButtonState* - tem o conhecimento da posição do Widget (*position* - como x e y) e se este está ser arrastado (*dragged*);
- *ContextState* - guarda a informação da conta do cliente (*account*), a informação dos gestores da conta (*accountManagers*), as conversas entre o cliente e o *chatbot* (*conversations*), as notificações da conta, o total lidas e não lidas (*notificationsData*) e os *tickets* associados (*tickets*);
- *TransientState* - tem dois atributos, que guardam informação de se o utilizador está autenticado (*authenticated*) ou está a tentar autenticar-se (*attemptingAuth*).

```
4   export interface WidgetHelpState {
5     widget: WidgetState;
6     button: ToggleButtonState;
7     context: ContextState;
8     transient: TransientState;
9   }
10
11  export interface WidgetState {
12    open: boolean;
13    activeView: string | null;
14    viewData: any;
15  }
16
17  export interface ToggleButtonState {
18    position: Coords;
19    dragging: boolean;
20  }
21
22  export interface ContextState {
23    account: IAccount | null;
24    accountManagers: IAccountManager[];
25    conversations: IConversation[];
26    notificationsData: IExtraNotification;
27    tickets: ITicket[];
28  }
29
30  export interface TransientState {
31    authenticated: boolean;
32    attemptingAuth: boolean;
33  }
```

Figura 6.5: Excerto das interfaces utilizadas para guardar dados na *Store*

Selectors

Para poder aceder aos objetos/interfaces descritos anteriormente, foram criados vários *selectors* que acedem a diferentes estados e permitem consultar os valores guardados a qualquer momento, "subscrevendo" os dados com o método *subscribe()*. Estes estão apresentados na figura 6.6 e são:

- *getFullState* - retorna o objeto completo ou todo o estado;
- *getWidgetState* - retorna o estado apenas do *widget*;
- *getToggleButtonState* - retorna o estado apenas do botão do *widget*;

- *getContextState* - retorna o estado que contém apenas informações do cliente;
- *getTransientState* - retorna o estado do processo de autenticação;
- *getAuthState* - retorna a informação relativa ao token da conta (dados de autenticação perante o Services);
- *getAccountManagers* - retorna os gestores da conta;
- *getNotificationsData* - retorna os dados relativos às notificações do cliente;
- *getTickets* - retorna os dados relativos aos *tickets* do cliente;
- *getConversation* - retorna uma conversa por id;
- *getConversations* - retorna todas as conversas;
- *getNotificationsPlusTickets* - retorna a informação tanto das notificações, como dos *tickets*;

```
4  const widgetHelpState = createFeatureSelector<WidgetHelpState>('widgetHelp');
5
6  export const getFullState = createSelector(widgetHelpState, state ⇒ state);
7
8  export const getWidgetState = createSelector(widgetHelpState, state ⇒ state.widget);
9
10 export const getToggleButtonState = createSelector(widgetHelpState, state ⇒ state.button);
11
12 export const getContextState = createSelector(widgetHelpState, state ⇒ state.context);
13
14 export const getTransientState = createSelector(widgetHelpState, state ⇒ state.transient);
15
16 export const getAuthState = createSelector(widgetHelpState, state ⇒ state.context.account ? state.context.account.auth : null);
17
18 export const getAccountManagers = createSelector(widgetHelpState, state ⇒ state.context.accountManagers);
19
20 export const getAccountState = createSelector(widgetHelpState, state ⇒ state.context.account);
21
22 export const getNotificationsData = createSelector(widgetHelpState, state ⇒ state.context.notificationsData);
23
24 export const getTickets = createSelector(widgetHelpState, state ⇒ state.context.tickets);
25
26 export const getConversation =
27   (id) ⇒ createSelector(widgetHelpState, state ⇒
28     state.context.conversations ? state.context.conversations.find(conversation ⇒ conversation.id == id) : null);
29
30 export const getConversations = createSelector(widgetHelpState, state ⇒ state.context.conversations);
31
32 export const getNotificationsPlusTickets = createSelector(
33   widgetHelpState,
34   state ⇒ ({ notificationsData: state.context.notificationsData, tickets: state.context.tickets })
35 );
```

Figura 6.6: *Selectors* criados para aceder ao estado da aplicação

Actions

As *Actions* representam essencialmente uma classe que implementa a interface *Action* da biblioteca NgRx, contendo uma variável constante com o nome da ação e especificando um construtor com os parâmetros a passar ao *reducer*.

As classes apresentadas na figura 6.7, seguem todas esta norma, com parâmetros e nomes diferentes. Deste modo, omite-se os construtores e variáveis das restantes classes e como a designação das classes permite perceber o que cada uma das ações efetua, não será abordada a sua descrição, ao contrário dos componentes do NgRx explicados anteriormente.

```
17 > export class Authenticate implements Action {···
22   }
23
24 > export class LoadWidget implements Action {···
29   }
30
31 > export class ToggleWidget implements Action {···
36   }
37
38   export class ChangeView implements Action {
39     readonly type = CHANGE_VIEW;
40
41     constructor(public newView: string,
42       public viewData?: any) {
43     }
44   }
45
46 > export class ChangePosition implements Action {···
52   }
53
54 > export class SendMessage implements Action {···
60   }
61
62 > export class LoadNotifications implements Action {···
68   }
69
70 > export class LoadTickets implements Action {···
76   }
77
78 > export class UpdateNotification implements Action {···
83   }
84
85 > export class LoadAccountManager implements Action {···
90   }
```

Figura 6.7: *Actions* criadas para efetuar mudanças ao estado da aplicação

Effects

Apenas existem 4 efeitos criados, que ocorrem após a ação declarada no *ofType* ser resolvida pelo *reducer*. Utilizam normalmente um selector para aceder a dados guardados na *store* e invocar serviços, neste caso, para realizar algo, terminando com a invocação de uma ação que atualiza a *store* com os novos dados.

Posto isto, são apresentados, respetivamente:

- *auth* - quando a ação *Authenticate* ocorre, este *effect* é ativado e, com os últimos dados da conta que estão no estado *AccountState*, é invocado o serviço *AuthService* que utiliza a conta do utilizador e o *token* de autenticação, para autenticar o utilizador no Services e carregar o *widget* (disparando a ação *LoadWidget*);
- *loadNotificationsOnAuth* - ocorre aquando da ação *LoadWidget*, utilizando também os últimos dados da conta do cliente na *store* e invocado o serviço que acede ao Services para ir buscar as notificações do cliente, carregando-as para a *store* (disparando a ação *LoadNotifications*);
- *loadTicketsOnAuth* - igual ao efeito anterior, mas no fim, em vez de disparar a ação *LoadNotifications*, dispara a ação *LoadTickets*;
- *loadManagerOnWidgetLoad* - carrega os gestores da conta do cliente, tal como as anteriores quando o *widget* é carregado, recorrendo ao Services através do serviço *accountManagerService* (disparando depois ação *LoadAccountManager*).

```
19 > /** ···
22   auth = createEffect(() ⇒ this.actions.pipe(
23     ofType<fromWidgetHelpActions.Authenticate>(fromWidgetHelpActions.AUTHENTICATE),
24     withLatestFrom(this.store.select(getAccountState)),
25     switchMap(([action, account]: [fromWidgetHelpActions.Authenticate, IAccount]) ⇒
26     this.authService.auth(account, action.useCurrentToken)
27       .pipe(
28         map(newAccount ⇒ new LoadWidget(newAccount)), catchError(() ⇒ of(new LoadWidget(null)))
29       ))
30     )
31   );
32
33 > /** ···
36   loadNotificationsOnAuth = createEffect(() ⇒ this.actions.pipe(
37     ofType<fromWidgetHelpActions.LoadWidget>(fromWidgetHelpActions.LOAD_WIDGET),
38     withLatestFrom(this.store.select(getAccountState)),
39     filter(([, account]) ⇒ account ≠ null),
40     switchMap(([, account]) ⇒ this.notificationsService.fetchNotifications(account)),
41     map(data ⇒ new LoadNotifications(data, true))
42     )
43   );
44
45 > /** ···
48   loadTicketsOnAuth = createEffect(() ⇒ this.actions.pipe(
49     ofType<fromWidgetHelpActions.LoadWidget>(fromWidgetHelpActions.LOAD_WIDGET),
50     withLatestFrom(this.store.select(getAccountState)),
51     filter(([, account]) ⇒ account ≠ null),
52     switchMap(() ⇒ this.ticketsService.fetchTickets()),
53     map(tickets ⇒ new LoadTickets(tickets, true)),
54     )
55   );
56
57 > /** ···
60   loadManagerOnWidgetLoad = createEffect(() ⇒ this.actions.pipe(
61     ofType<fromWidgetHelpActions.LoadWidget>(fromWidgetHelpActions.LOAD_WIDGET),
62     withLatestFrom(this.store.select(getContextState)),
63     // Only load if there is an authenticated account and if it has no account managers loaded
64     filter(([, context]) ⇒ context.account ≠ null && context.accountManagers.length ⩵ 0),
65     switchMap(([, context]) ⇒ this.accountManagerService.fetchAccountManager(context.account)),
66     map(accountManager ⇒ new LoadAccountManager(accountManager)),
67     )
68   );
```

Figura 6.8: *Effects* criados para atuar quando uma ação é lançada

Reducer

O *reducer*, dependendo da ação que é enviada, é responsável por atualizar a *store* com a nova informação que é enviada através da *action*. A título de exemplo, escolheu-se mostrar o redutor que efetua a mudança de vista do Widget (fig. 6.9). É importante referir que, no Widget-Help, não existe o conceito de mudança de rota, portanto as vistas são todas mostradas no mesmo url. O que é trocado é apenas o código HTML que é mostrado ao utilizador.

Neste exemplo, os dados contidos na *action* são a *newView* (nova vista) para a qual se pretende transitar e os dados (*viewData*) a passar para essa vista (se não estiverem definidos coloca-se um objeto vazio - {}).

```
27    case widgetHelpActions.CHANGE_VIEW: {
28      return {
29        ...state,
30        widget: {
31          ...state.widget,
32          activeView: action.newView,
33          viewData: action.viewData || {},
34        }
35      };
36    }
```

Figura 6.9: Excerto de código que mostra o *reducer* a efetuar a mudança de vista

6.2.3 Internacionalização e Traduções

Como mencionado nos requisitos não funcionais em 4.2.2, tanto o *chatbot*, como o Widget devem suportar as 4 linguagens utilizadas pela E-goi. À semelhança de outros projetos da E-goi, foi utilizada a biblioteca *@ngx-translate*, que permite utilizar ficheiros locais JSON com as traduções associadas a uma *String*, para traduzir as *strings* colocadas no código através de um *pipe* (ver figura 6.10). Dependendo da linguagem da conta do cliente, esta biblioteca processa todas as *strings* começadas por *"TR/HelpWidget/..."* e *"TR/Botpress/..."* apresentando-as no idioma correto.

```
3   <!-- Header -->
4   <div class="hw-view-header">{{'TR/HelpWidget/Views/Services/Design/Title' | translate | propertitlecase}}</div>
5   <!-- Description -->
6   <div class="hw-view-text hw-service-description">
7     {{'TR/HelpWidget/Views/Services/Design/Description' | translate}}
8   </div>
```

Figura 6.10: Exemplo de utilização do *pipe* para traduzir uma *string*

Como o serviço de traduções do Botpress é pago, o texto é especificado no Botpress como *"TR/Botpress/..."*, para que quando for apresentado ao utilizador o projeto do WidgetHelp o possa traduzir para o idioma correto, em vez de ser o Botpress a efetuar as traduções.

6.2.4 Integração com a plataforma da E-goi

Em conformidade com o que estava planeado, que era colocar este projeto em produção, cumprindo o requisito do UC-18 (integração do *Widget* com os projetos da E-goi como *Web Component*), foi necessário compactar o código do mesmo para poder ser apresentado no *website* da E-goi.

Existiam duas alternativas: colocar o código deste projeto diretamente na máquina onde é executado o código de *front-end* da E-goi ou manter este código numa máquina própria, invocando-o sob a forma de *Web Component*. Considerou-se a segunda opção como a melhor, mantendo os projetos separados e, caso seja necessário o Widget sofrer alguma alteração, basta apenas alterar na máquina deste projeto.

De forma a agilizar o processo, foi definida uma *pipeline* no Jenkins, que efetua o *pull* das alterações, instala as dependências necessárias, realiza a atualização dos ficheiros de traduções, efetua o *build* do projeto e no final o *deploy* para a máquina de produção.

No passo de *build* existe uma *flag* (a de *"–optimization"*) que só é ativa, no caso de produção. Esta opção faz com que os ficheiros sejam minificados[6] aumentando a eficiência, acessibilidade e diminuindo a latência do carregamento das páginas (dificultando o *debug*, porque os ficheiros ficam praticamente impossíveis de entender ao olho humano e por isso ser feito apenas em produção). O projeto é convertido num único ficheiro de JavaScript (ou vários para compatibilidade), que pode ser servido estaticamente.

Depois deste processo, foi então colocado como *Web Component* no projeto de *front-end* da E-goi, que aponta para o servidor que o serve, através do código da figura 6.11. Este código, colocado no ficheiro de arranque do projeto, adiciona ao *"index.html"* (documento inicial apresentado) os estilos do WidgetHelp (código *Cascading Style Sheets* (CSS)) à *tag head* da página e o código JavaScript à *tag body*.

```
156  function insertWidgetHelp () {
157    const s = document.createElement('script');
158    const sheet = document.createElement('link');
159    const widget = document.createElement('hw-root');
160
161    if (STAGE !== 'production') {
162      s.src = "https://dev-help.egoiapp.com/widget-help.js";
163      sheet.href = "https://dev-help.egoiapp.com/widget-help.css"
164    } else {
165      s.src = "https://help.egoiapp.com/widget-help.js";
166      sheet.href = "https://help.egoiapp.com/widget-help.css"
167    }
168
169    sheet.rel = "stylesheet";
170    s.type = "text/javascript";
171    widget.style = "z-index: 9999; width: 100%; bottom: 0; top: 0";
172
173    document.getElementsByTagName('head')[0].append(s, sheet);
174    document.getElementsByTagName('body')[0].appendChild(widget);
175  }
```

Figura 6.11: Exemplo de utilização do *pipe* para traduzir uma *string*

6.3 ElasticSearch

Para que o Botpress possa devolver respostas ao cliente a E-goi pretendeu que fosse utilizado a base de dados do ElasticSearch, que como referido anteriormente, tem um registo de todos os artigos do HelpDesk. Em primeiro lugar, como uma das necessidades era acrescentar novas fontes de retorno de artigos a este projeto (UC-16), investigou-se como eram adicionados os artigos que já existiam. Além disso, explorou-se também a estrutura desta ferramenta e a documentação das API's *Document*, *Search*, *Indices*, *Mapping* e *Analysis*[7].

Na realidade, o desenvolvimento necessário neste projeto apresenta-se sob a forma de duas tarefas fundamentais: adicionar mais fontes de artigos e melhorar a pesquisa existente.

[6] https://angular.io/guide/workspace-config#optimization-configuration
[7] https://www.elastic.co/guide/en/elasticsearch/reference/6.4/docs.html

6.3.1 Preparação dos índices

Como o objetivo da E-goi é ter um *index* para os artigos de cada língua, considerou-se uma boa prática criar todos com as mesmas definições. Definindo um *template* de criação dos mesmos, é possível garantir que sempre que for criado um novo *index*, este fica com as definições pretendidas.

O conceito de *Mapping* refere-se ao "Processo de definir como um documento e os campos que contém são guardados e indexados. Por exemplo, utilizam-se *mappings* para definir: que campos devem ser tratados como texto, que campos contêm números, datas,...".

Optou-se por criar um novo *cron job* que efetua um PUT a um *template* pré-definido, caso no futuro se pretenda atualizar a forma como os índices são criados. A figura 6.12 apresenta:

- *index_patterns* - a que tipo de índices é que este *template* deve ser aplicado (neste caso pt_pt, en_en,...);
- *settings* - especificam 5 *shards* (espécie de motores de pesquisa que tratam de partes dos dados de um *index*) - valor que estava colocado previamente; réplicas é um mecanismo de segurança caso uma esteja em baixo outra entra em ação - sendo que 0 era o valor colocado também previamente; e, por fim, o *analyzer* que define dois métodos de pesquisa (por separação pelos espaços das frases em várias palavras e conversão das frases para minúscula e sem carateres especiais);
- *mappings* - é definido o *analyzer*, por defeito, que foi criado anteriormente; quanto aos *fields*, quando mais tarde for feita a *query* através de um POST à *Application Programming Interface* (API) do Elastic, é possível fazer, por exemplo, "title.keyword", para pesquisar apenas nos primeiros 256 carateres do título; já o "title.folded", pesquisaria no título, mas em mínusculo e sem carateres especiais.

```php
public function updateTplAction()
{
    $data = [
        "index_patterns" => ["pt_pt", "en_en", "es_es", "br_br"],
        "settings" => [
            "number_of_shards" => 5,
            "number_of_replicas" => 0,
            "analysis" => ["analyzer" => [
                "egoi_analyzer_content" => [
                    "type" => "custom",
                    "tokenizer" => "whitespace"
                ],
                // Analyzer to search without accents and in lowercase
                "egoi_analyzer_folding" => [
                    "filter" => ["lowercase","asciifolding"],
                    "tokenizer" => "standard"
                ]
            ]]
        ],
        "mappings" => ["_doc" => ["properties" => [
            "article" => [
                "properties" => [
                    "id" => [],
                    "title" => [],
                    "content" => [],
                    "content_text" => [],
                    "keywords" => [],
                    "url" => [],
                    "access" => [],
                    "type" => []
                ]
            ]]]
        ]
    ];
```

Figura 6.12: Configuração do *template* (parte 1)

Existem inúmeras configurações e combinações possíveis, além de parâmetros que não foram aqui apresentados e constam apenas na documentação. Contudo, após inúmeros testes, considerou-se esta configuração como suficiente e útil para a forma como, mais tarde, se pretende utilizar esta API para pesquisas através do Botpress.

```php
// Since the properties are the same we can just iterate every property and give it the same value
foreach ($data["mappings"]["_doc"]["properties"]["article"]["properties"] as &$property) {
    $property = [
        "type" => "text",
        // Set default analyzer for every field
        "analyzer" => "egoi_analyzer_content",
        "fields" => [
            // If we want to search only in the first 256 chars, we can do so, by specifying
            // field.keyword in the fields of the query
            "keyword" => [
                "ignore_above" => 256,
                "type" => "keyword"
            ],
            // If we want to search with the folding filter
            "folded" => [
                "analyzer" => "egoi_analyzer_folding",
                "type" => "text"
            ]
        ]
    ];
}
```

Figura 6.13: Configuração do atributo *mappings* do *template* (parte 2)

6.3.2 Melhoria da query de pesquisa

Na figura 6.14 é apresentada a configuração da *query* de pesquisa que será enviada através de um POST request, como *payload*. Esta configuração apresenta diversos parâmetros, que

permitem calcular um *score*, para cada artigo encontrado. No fim, os artigos com maior score, ordenados do maior para o menor, são retornados, considerando:

- *_source* - campos a serem retornados (neste caso apenas é relevante o título e o url do artigo);
- *query* - especifica-se que utilizando a pesquisa em "$this->query", apenas se irá pesquisar no título e no conteúdo sem *tags* HTML; calcula-se o *score*, prioritizando se o texto for encontrado no título (dando um peso acrescido de 2 vezes mais importância) e pesquisando pelos termos, tanto no título e conteúdo normais, como na propriedade *folded* definida anteriormente (no template); além disso, prioritizam-se também os artigos do helpdesk (com um peso acrescido de 1.5 vezes mais importância); e, por fim, aconsideram-se apenas os artigos públicos e que são do tipo que está definido em "$types" (foi criado este filtro, para caso se pretenda pesquisar apenas em artigos do Blog ou Youtube ou HelpDesk);
- *size* - número de artigos retornados (manteve-se um máximo de 5 artigos).

No *"type"* é especificada a função *"most_fields"*, que essencialmente significa que o *score* é calculado para cada campo (título, título *folded*, conteúdo...) e no fim combinado todos os scores. Por defeito, o utilizado é o *"best_fields"*, que apenas considera o maior score encontrado entre todos os campos. Foi utilizado o primeiro, porque segundo os docs deve ser utilizado quando se investigam múltiplos campos de formas diferentes (que é o caso).

```
90          $data = [
91              "_source" => ['article.title', 'article.url'],
92              "query" => [
93                  "bool" => [
94                      "must" => [["multi_match"  => [
95                              "query" => $this->query,
96                              "type"  => "most_fields",
97                              "fields" => [
98                                  // We can use the "field.keyword" to search in the first 256 chars only
99                                  "article.title^2", // Give higher priority to title
100                                 "article.title.folded^2",
101                                 "article.context_text",
102                                 "article.context_text.folded"
103                             ],
104                     ]]],
105                     "should" => ["term" => ["article.type" => [
106                                 // Prioritize helpdesk articles
107                                 "value" => "helpdesk",
108                                 "boost" => 1.5
109                     ]]],
110                     /* Reading through the docs, it says that "filters" cache querys,
111                     thus making them faster. Plus this is the correct way to do it */
112                     "filter" => [
113                         [
114                           "match"  => [
115                             "article.access" => 'P'
116                           ]
117                         ],
118                         /* Instead of a match query, you could simply try the
119                         terms query for ORing between multiple terms.*/
120                         [
121                           "terms" => [
122                             "article.type" => $types
123                           ]
124                         ]
125             ]]],
126             'size' => 5
127         ];
```

Figura 6.14: *Query* de pesquisa ao ElasticSearch

6.3.3 LiveAgent API

No projeto do Services, existe um *cron job* que corre todos os dias para fazer o sincronismo entre a base de dados do Elastic e a base de dados do LiveAgent. Neste controlador é feito um GET request à API do LiveAgent que devolve todos os artigos públicos, percorrendo, de seguida, cada um deles, construído-se um objeto no formato descrito na figura 6.15 e depois enviado para o Elastic através de um POST request (figura 6.16).

```
301                 $finalData = array(
302                     'id' => $data->kb_entry_id,
303                     'title' => ($title),
304                     'content' => ($content),
305                     'content_text' => (strip_tags($content)),
306                     'keywords' => $keywords,
307                     'url' => 'https://helpdesk.e-goi.com/index.php?type=page&urlcode=' . $data->urlcode,
308                     'access' => $data->access,
309                     'type' => 'helpdesk'
310                 );
```

Figura 6.15: POST *data* para enviar para o ElasticSearch

```
324                 // Send all the info to elastic: <url>/<index>/<mapping_type>/<id>
325                 $this->logInfo($locale, "HelpDesk Article: " . $finalData["url"]);
326                 $this->sendToElastic(
327                     $this->config['protocol'] . $this->config['url'] .'/' . $locale . '/_doc/hd_' . $finalData['id'],
328                     $finalData
329                 );
```

Figura 6.16: POST *method* para enviar para o ElasticSearch

Quanto aos campos apresentados:

- *id* - representa o id do artigo do HelpDesk;
- *title* - título do artigo;
- *content* - conteúdo do artigo (com tags HTML);
- *content_text* - conteúdo do artigo (sem tags HTML);
- *keywords* - palavras-chave do artigo;
- *url* - link público do artigo;
- *access* - acesso do artigo (público ou não listado);
- *type* - tipo do artigo (neste caso é *helpdesk*, mas para as outras fontes será o equivalente à fonte);

A imagem de baixo apresenta a invocação do método definido para efetuar um *curl request* à API do ElasticSearch com o url definido, passando como primeiro parâmetro de caminho, no url, o index onde se pretende guardar o artigo (pt_pt/en_en/br_br/es_es) e o id do mesmo (hd_id), enviando como segundo parâmetro o objeto construído anteriormente.

Tendo conhecimento desta implementação, pretendeu-se seguir a mesma abordagem para os artigos do Youtube (utilizando a API do Youtube da conta da E-goi) e do Blog (utilizando a API do Blog da E-goi construído em WordPress).

6.3.4 Youtube API

Depois de configurada a conta do Youtube E-goi para suportar pedidos via API, foi criado no Services um novo *job* para efetuar pedidos à mesma. Na figura 6.17, começou-se por construir um objeto que contém os parâmetros fundamentais para efetuar o pedido. Depois, como existe um limite de 50 vídeos por página, é devolvido um *token* da página seguinte, que permite navegar de "página em página", o que se traduz em efetuar vários pedidos seguidos, para se conseguir obter todos os vídeos no canal da E-goi[8].

Como existem *playlists* de vídeos que não são relevantes foi criada uma lista de supressão para serem ignoradas. Além disso, verificou-se também que muitos vídeos não pertenciam a nenhuma *playlist* e eram promocionais, isto é, não eram propriamente úteis para resolver problemas dos clientes. Ficou acordado que o gestor deste canal deveria agrupar todos os vídeos em *playlists* e na descrição da mesma estaria o idioma dos vídeos contidos na mesma (uma vez que não havia mais nenhuma forma de identificar o idioma do vídeo pela informação devolvida nos pedidos).

O objetivo passou por obter todas as playlists, iterarando cada uma delas e processando cada vídeo para ser submetido na base de dados (ver fig. 6.18).

```
357         // Create query params to send in the api request
358         $queryParams = [
359             "key" => $this->youtubeApiKey,
360             "part" => "snippet,contentDetails,status",
361             "channelId" => $this->youtubeChannelId,
362             "maxResults" => self::$_youtubeRequestLimit
363         ];
364         // Add path to api url
365         $youtubePlaylistsURL = $this->youtubeApiUrl . "/playlists";
366         // Send request and retrieve response
367         $content = json_decode($this->requestYoutube($youtubePlaylistsURL, $queryParams));
368         // Create a supression list of playlists we don't want
369         $suppressionIdsList = [
370             "PL608527DCE2219DA3", // Videos Comerciais PT
371             "PL0B700B289692CB5C", // Videos Comerciais BR
372             "PLbcDgMEmZrQ1FT-KO2Zg5mOADw-maxQJ6", // Fun
373             "PLbcDgMEmZrQ2_SwNb1X9quSxJifnzHX8L" // Old But Gold
374         ];
375         // (Dealing with pagination) This means we didnt get all playlists in a single request,
376         // and we need to go to the next page
377         $nextPageToken = isset($content->nextPageToken) ? $content->nextPageToken : null;
378         while (isset($nextPageToken)) {
379             // Add token to queryString
380             $queryParams["pageToken"] = $nextPageToken;
381             // Do the request and store results
382             $nextPageContent = json_decode($this->requestYoutube($youtubePlaylistsURL, $queryParams));
383             // Append new results to master array
384             $content->items = array_merge($content->items, $nextPageContent->items);
385             // Retrieve next page token or null
386             $nextPageToken = isset($nextPageContent->nextPageToken) ? $nextPageContent->nextPageToken : null;
387         }
```

Figura 6.17: Parte 1 da implementação dos pedidos à API do Youtube

[8] https://www.youtube.com/user/plataformaegoi

```php
foreach ($content->items as $playlist) {
    // Get lang from playlist description
    $indexLang = $playlist->snippet->description;
    // If it's not in the suppresion list
    // If it's public (not interested in private or unlisted playlists)
    // If it's has at least one video
    // If the lang in the description is one of the lang Strings (PT, ES or EN)
    if (
        !in_array($playlist->id, $suppressionIdsList) &&
        $playlist->status->privacyStatus == "public" &&
        $playlist->contentDetails->itemCount > 0 &&
        in_array($indexLang, array_keys($langsArray))
    ) {
        $youtubePlaylistItemsUrl = $this->youtubeApiUrl . "/playlistItems";
        // Add playlistId to queryString
        $queryParams["playlistId"] = $playlist->id;
        $response = json_decode($this->requestYoutube($youtubePlaylistItemsUrl, $queryParams));

        foreach ($response->items as $video) {
            // If it's a public video
            if ($video->status->privacyStatus == "public") {
                $finalData = [ ···
                ];

                $this->logInfo($langsArray[$indexLang], "Youtube Article: " . $finalData["url"]);

                // Send all the info to elastic: <url>/<index>/<mapping_type>/<id>
                $this->sendToElastic( ···
                );
                // Send also to the br index
                if ($indexLang == 'PT') {
                    $this->logInfo($langsArray['BR'], "Youtube Article: " . $finalData["url"]);
                    $this->sendToElastic( ···
                    );
                }
            }
        }
    }
}
```

Figura 6.18: Parte 2 da implementação dos pedidos à API do Youtube

São realizados portanto dois pedidos fundamentais: um para ir buscar todas as *playlists* e outro para ir buscar todos os vídeos de cada *playlist*. Analisando cada vídeo é submetido no Elastic search com a informação semelhante ao que foi referido para o HelpDesk (figura 6.15).

6.3.5 Wordpress API

O código implementado para a comunicação com a API do Wordpress é ligeiramente diferente, no sentido em que existem artigos publicados para cada linguagem da E-goi (PT, EN, BR e ES) e cada um deles pode ser acedido colocando o parâmetro no *url* ("api/pt/..."). Logo, itera-se um *array* de linguagens e, para cada uma, é feito um GET *request* modificando esse parâmetro, que trás os artigos até um limite de 100. Modificando o *query parameter* "page" obtêm-se as páginas seguintes.

```php
foreach ($this->languages as $langArray) {
    $queryParams = [
        "per_page" => self::$_wordpressRequestLimit,
        "page" => 1
    ];
    // If the lang is EN we don't need to specify any lang in the url
    if ($langArray["lang"] == "EN") {
        $langArray["lang"] = "";
    }
    $wordPressURL = $this->wordpressApiUrl . strtolower($langArray["lang"]) . '/wp-json/wp/v2/posts';
    do {
        list($content, $totalPages) = $this->requestWordpress($wordPressURL, $queryParams);
        // Note: by default wordpress returns only published articles, so we don't need to validate that
        // https://developer.wordpress.org/rest-api/reference/posts/#definition
        $content = json_decode($content);
        foreach ($content as $article) {
            $finalData = [ ···
            ];

            $this->logInfo($langArray["code"], "Blog Article: " . $finalData["url"]);
            // Send all the info to elastic: <url>/<index>/<mapping_type>/<id>
            $this->sendToElastic( ···
            );
        }
        $queryParams["page"] += 1;
    } while ($queryParams["page"] ≤ $totalPages);
}
```

Figura 6.19: Implementação dos pedidos à API do Wordpress

Neste caso não é necessário uma *apikey*, porque ambos os serviços pertencem à E-goi e os pedidos à API estão limitados por *IP*.

6.3.6 Resultado final

Após todo o trabalho realizado, o Elastic (ver figura 6.20) ficou pronto para serem realizadas as pesquisas a partir do Botpress, sendo que: o índice português ficou com 885 artigos, o espanhol 833, o inglês 490 e o brasileiro com 896. Em relação a, por exemplo o indíce português, este número representa um crescimento de 3 vezes mais artigos, uma vez que só existiam apenas 269.

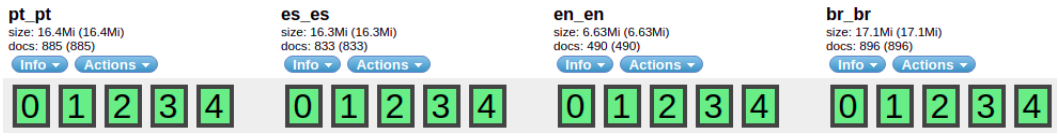


Figura 6.20: *Indexes* resultados e artigos por *index*

6.4 Botpress

Esta tecnologia foi já bastante detalhada durante o processo de investigação, realizado no subcapítulo do estado da arte em 2.4.3, sobre o modo de funcionamento e estruturação. Tal como mencionado existem duas alternativas de implementação: através de um *iframe* ou então via API. Como a E-goi desejava ter uma janela de conversa em que o código HTML é personalizado e em que os pedidos são processados pelo servidor do Services, escolheu-se utilizar a segunda opção.

Resta agora descrever a implementação realizada (fluxo de conversa) e apresentar o processo faseado realizado, de modo a poder utilizar o mecanismo de *Natural Language Understanding* (NLU) do Botpress, da melhor forma possível.

6.4.1 Fluxo de conversa (Chatbot)

Foram criados vários nós no fluxo do Botpress (figura 6.23), sendo que este é processado da esquerda para a direita. Assim que o fluxo é iniciado, é guardada a informação do utilizador em variáveis de sessão para serem acedidas, mais tarde, e extraída a língua para efetuar as pesquisas na base de conhecimento. De seguida, é feita a pesquisa de artigos através do nó *SearchArticles*, que invoca uma *action* intitulada *"lookup"*. Esta ação executa o código apresentado em 6.21, para invocar o Services e processar os artigos retornados. Esta função *"callApi"* invoca também um método para guardar na base de dados a pesquisa efetuada, junto com a informação relevante (fig. 6.22).

```
119  const callApi = async () => {
120    // Get config variables from config bot.config.json
121    let config = await bp.config.mergeBotConfig('chatgoi-help', {})
122
123    // If the services don't respond in 10 seconds, it will print an error message
124    var tookToLong = false
125    let lang = `${session.userLang}_${session.userLang}` || 'pt_pt' // Default to pt
126
127    let resp = await axios
128      .get(`${config['SERVICES_PROD']}/knowledgebase/search?language=${lang}&query=${event.preview}`, {
129        timeout: 15000
130      })
131      .catch(e => {
132        bp.logger.forBot(event.botId).error(e)
133        tookToLong = true
134      })
135
136    // If the resp is empty, it will create an empty (custom) response to be able to move on
137    const { data } = resp || { data: { count: 0, items: [] } }
138    const messages = await createMessages(data.items, tookToLong)
139    await storeOrUpdateDb(data.items)
140    await bp.events.replyToEvent(event, messages)
141
142    // Save the response in the session
143    session.response = data
144  }
```

Figura 6.21: Ficheiro *Lookup action* e o método *callApi*

```
72  const storeOrUpdateDb = async articles => {
73    // Map article titles to array
74    let suggestions = articles.map(a => stripHtml(a.title)).join(';') || ''
75    // Extract correct timestamp
76    let timestamp = new Date()
77    //timestamp.setHours(timestamp.getHours() + 1)
78    // Store article search in database to be consulted by admin project
79    bp.kvs.forBot('chatgoi-help').set(session.adminId, {
80      userId: event.target,
81      content: event.preview,
82      suggestions: suggestions,
83      util: 'indeterminado', // Default, we will change this when user answers either: "yes" or one of the other options
84      toSupport: false, // Same as above
85      date: timestamp.toISOString()
86    })
87  }
```

Figura 6.22: Ficheiro *Lookup action* e o método *storeOrUpdateDb*

Após retornados os artigos, caso existam ou não artigos associados à questão do utilizador, são dadas 3 opções comuns aos dois nós (*NoArticlesFound* e *ArticleFeedback*): falar com um agente, criar um novo *ticket* ou tentar novamente. No entanto, caso sejam devolvidos artigos, é-lhe dada a opção de responder se este foi útil ou não. Seguindo por esse nó (*SaveFeedback*) é invocada uma *action* que efetua um pedido semelhante ao da figura 6.22, mas neste caso para atualizar a pesquisa anterior. Aquando do registo do *feedback* do utilizador é-lhe realizada a pergunta "Tem mais questões?", à qual, se este responder "Sim"

o levará para o início, repetindo-se novamente o ciclo e se responder "Não" o *bot* devolverá "Estarei aqui...".

Ignorando os restantes nós por serem semelhantes, no sentido em que, dependendo da resposta selecionada pelo utilizador, é atualizada a informação na base de dados, no nó *AddTicketText* é dada a opção ao utilizador de acrescentar uma imagem ao *ticket* que este pretende criar (nó *SubmitImage*). Caso o cliente opte por adicionar uma imagem é-lhe pedido que faça *upload* e, caso não pretenda, o ticket é submetido ao HelpDesk, pelo projeto do Widget-Help. Alternativamente poderia ser colocar o projeto do Botpress a submeter o *ticket* (mas já que o Widget-Help contém os dados do utilizador, torna-se mais lógico efetuá-lo desta forma).

6.4.2 Botpress NLP

O diagrama da figura 6.24 é uma apresentação muito simples, do que se pretende fazer: interpretar a frase do utilizador, utilizando o mecanismo de NLP do Botpress; mapear as frases introduzidas para palavras-chave (com o conhecimento de palavras que devolvem artigos, e os corretos, para a pergunta realizada); enviar para o Services e este, por sua vez, remeter para o elastic as *keywords*, para pesquisar na base de dados e devolver resultados.

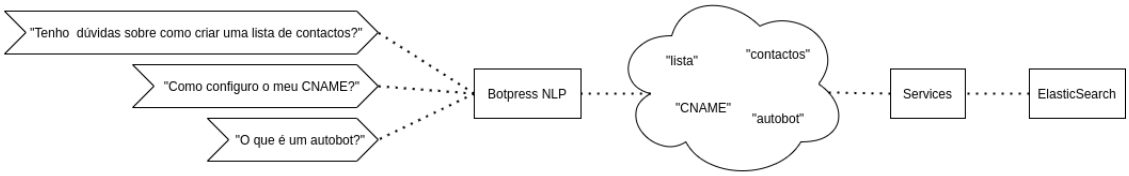


Figura 6.24: Ideologia inicial para o processamento textual

Antes da utilização de NLU do Botpress, optou-se por realizar uma investigação, de como e de que forma se deve treinar um modelo, para que este possa interpretar uma frase e devolver os resultados corretos aos utilizadores. Como tal, foram analisados os artigos muito interessantes [44], [45] e [46], que permitiram obter um conhecimento mais amplo e retirar algumas conclusões.

O notório destes 3 artigos analisados é: o pré processamento depende de cada contexto/situação; existem diversas/muitas técnicas de análise textual, que podem ser utilizadas para transformação de frases e palavras (serão apresentadas nas fases descritas); e, por fim, é necessário tanto automação e esforço humano para resolver problemas de análise textual.

Fase 1) Preparação dos datasets

Como descrito na parte introdutória em 1.2, o projeto do Admin mantém um registo das pesquisas efetuadas, bem como uma série de informações que permitem construir um *dataset*, isto é, um conjunto de dados útil para treinar o modelo NLU do Botpress. Sabendo as dúvidas dos clientes, a primeira fase consistiu em extrair um ficheiro *Comma-Separated Values* (CSV), para análise de dados e pré processamento textual.

Apesar da tabela no painel de administração conter filtros, exportaram-se todas as linhas da tabela para um ficheiro CSV, que segue o formato apresentado na figura 6.25 e contém um registo total de cerca de 232 mil pesquisas efetuadas pelos clientes, com datas compreendidas entre o ano de 2013 a 2021.

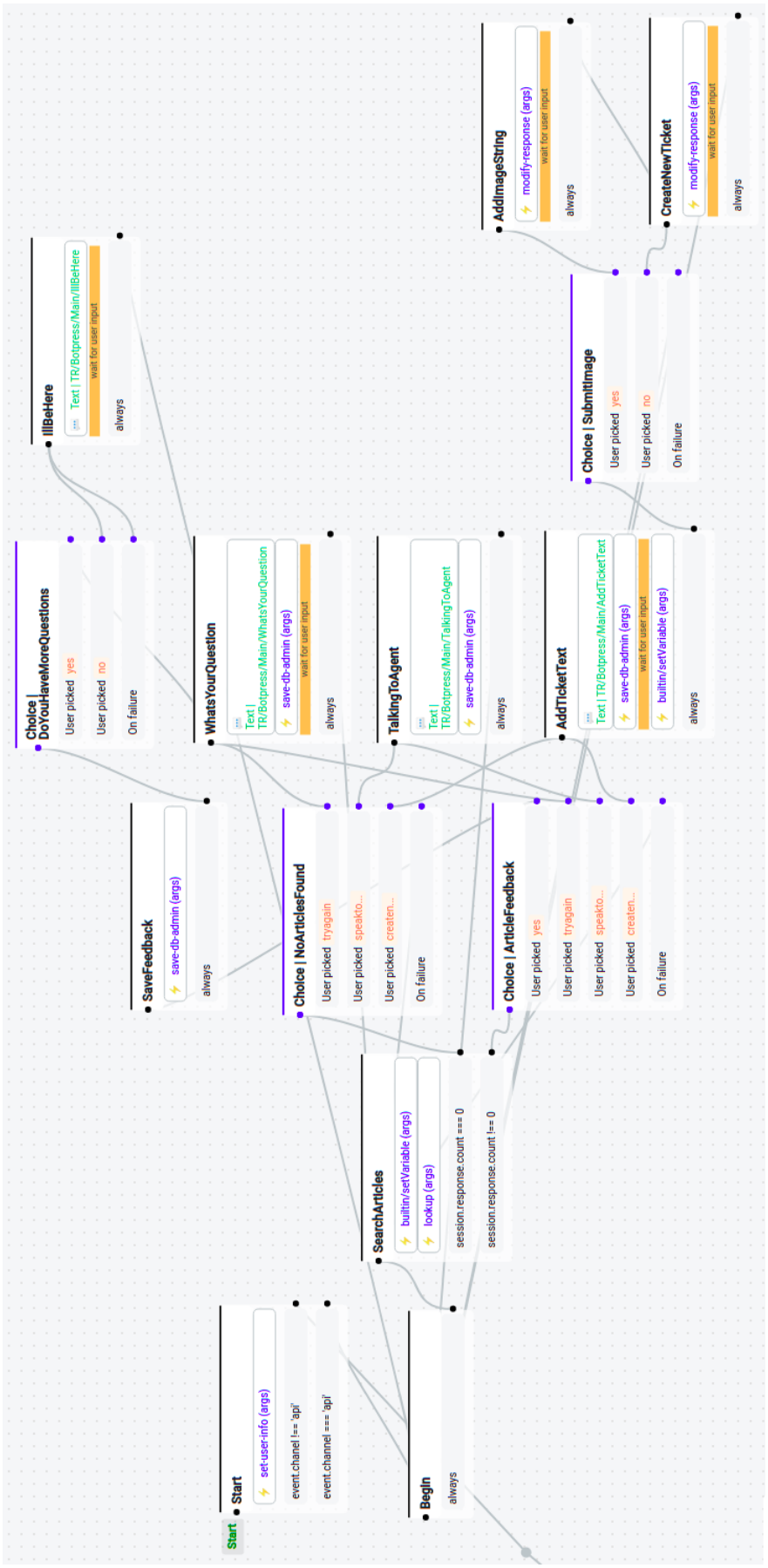


Figura 6.23: Fluxo criado no Botpress e respetivos nós

CLIENT_ID	USER_ID	CONTEUDO	CRIADO_EM	UTIL	PAIS	ENVIADO_AO_SUPORTE	SUGESTÕES
399329	422058	conteúdo da campanha desaparece quando se grava	2/1/2021 15:06	Não	Portugal	Sim	Que códigos de
811138	858527	Boa tarde, Enviamos uma newsletter, onde fora	2/1/2021 14:35	Não	Portugal	Sim	E-goi tells me th
811138	858527	Boa tarde, Enviamos uma newsletter, onde fora	2/1/2021 14:33	Não	Portugal	Não	E-goi tells me th
811138	858527	Boa tarde, Na última newsletter que enviamos,	2/1/2021 14:31	Sim	Portugal	Não	How does e-goi

Figura 6.25: Extrato do dataset inicial, obtido através do projeto Admin

Das colunas apresentadas, as relevantes são apenas o "Conteúdo", "Criado Em" e o "País", pois identificam a pesquisa realizada, a data da mesma e o país do cliente, respetivamente.

Fase 2) Pré-Processamento textual

Apesar de ser bastante útil saber quais as pesquisas que os utilizadores E-goi realizam, a forma como as escrevem produz um grande problema, a inconsistência dos dados. Por exemplo, a pesquisa pode ser tão simples como:

```
‘‘criar lista de contactos’’
```

...ou tão complexa como:

```
‘‘Boa tarde,<br><br>Estou a tentare remover subxcritorex na Lista x, só
que o E-Goi não me esta a aaaaapresent@r a caixa de confirmação de remoção na
página https://exemplo.com. Podem contactar-me em 999-999-999
ou enviar-me um email em exemplo@email.com<br><br>Obrigada.’’
```

É percetível, ao olho humano, que existem vários obstáculos/problemas (no segundo exemplo), para que se possa obter uma frase que permita ser adicionada ao modelo de processamento textual do *chatbot*, tais como: a utilização de informações pessoais como números de telemóvel ou emails, a aparência de tags HTML, erros sintáticos e semânticos, entre outros. Há ainda outros tipos de frases que, ou estão completamente fora do âmbito da E-goi (ex: "marcar um voo para a Madeira"), ou simplesmente não fazem, de todo, qualquer sentido.

Deste modo, recorreu-se à linguagem Python, com o intuito de normalizar os dados, criando um *script*, que permita automatizar e otimizar este processo de transformação, percorrendo o ficheiro *Excel* (XSLX) de *input*, tornando-o num outro ficheiro de *output* (no formato CSV), com frases "limpas" e que se adequem ao contexto da E-goi.

```
from bs4 import BeautifulSoup
from langdetect import detect
from spellchecker import SpellChecker
from datetime import datetime
import pandas as pd
import re
import nltk
import string
import unidecode
```

Figura 6.26: Parte I do *script*

Começando por uma breve explicação das bibliotecas utilizadas, na figura 6.26:

- BeautifulSoup - Ferramenta para análise de ficheiros HTML e XML;

- Detect - Deteção da linguagem de uma frase (baseada na biblioteca de deteção utilizada pelo google[9]);
- SpellChecker - Utiliza o algoritmo da distância de Levenshtein[10] (tal como o ElasticSearch) para encontrar permutações de uma palavra (corrigindo a palavra);
- Pandas - Biblioteca de análise e manipulação de dados (normalmente de um ficheiro csv ou xslt);
- Re - Operações de expressões regulares;
- Nltk - *Toolkit* para processamento de linguagem humana (ex: contém *stop-words* portuguesas como de, da, do, e, o, a...);
- String - Operações com strings (Pontuação, carateres ASCII, *whitespaces*,...);
- Unidecode - Operações com texto em Unicode[11].

Após importadas estas bibliotecas, inicializam-se as variáveis necessárias, que serão utilizadas mais à frente, é importado o ficheiro Excel com todas as pesquisas (figura 6.28), selecionadas as colunas relevantes e removidas todas as linhas que não são pesquisas portuguesas ou não foram realizadas após o ano de 2019 (figura 6.27).

```
# Initialize variables
nltk_lang = 'portuguese'    # Nltk language to use
spell_lang = 'pt'           # Spellchecker lang
spell_extra = 'en'          # Spellchecker extra lang
detect_lang = 'pt'          # Langdetect language to use
word_threshold = 20         # Max word length (https://www.dicio.com.br/maior-palavra-da-lingua-portuguesa/)

# Load excel file with all data
df = pd.read_excel (r'original.xlsx')
# Extract only relevant Columns
df = pd.DataFrame(df, columns = ['CONTEUDO', 'PAIS', 'CRIADO_EM'])
print(f'TOTAL = [{len(df)}] results')

# Load countries file with all countries and their lang
df_c = pd.read_csv(r'paises.csv')
# Select portuguese countries
pt_countries = '|'.join(df_c[df_c['group'].str.contains('ptPT|interPT')]['codigo'])
# Create array of booleans to filter dataframe
dateCondition = df['CRIADO_EM'] > pd.Timestamp(2019, 1, 1) # This has to be separated
# Filter dataframe by the Countries specified and by the previous condition
dfPT = df[df['PAIS'].str.contains(pt_countries, case=False) & dateCondition]
print(f'TOTAL = [{len(dfPT)}] results')
```

```
TOTAL = [232842] results
TOTAL = [22145] results
```

Figura 6.27: Parte II do *script*

Como a E-goi tem um registo de todos os países associados ao seu grupo de linguagem falada (PT, ES, BR e EN), selecionam-se, no caso de portugal, os grupos *ptPT* e *interPT* (figura 6.29), filtrando-se assim todas as pesquisas realizadas a partir de países portugueses (que, à partida, serão em português).

[9] https://code.google.com/archive/p/language-detection/
[10] https://en.wikipedia.org/wiki/Levenshtein_distance
[11] https://en.wikipedia.org/wiki/Unicode

	CONTEUDO	PAIS	CRIADO_EM
0	conteúdo da campanha desaparece quando se grava	Portugal	2021-02-01 15:06:55
1	reembolso	Brasil	2021-02-01 14:55:42
2	automação	Brasil	2021-02-01 14:38:03
3	Boa tarde, Enviamos uma newsletter, ond...	Portugal	2021-02-01 14:35:34
4	Boa tarde, Enviamos uma newsletter, ond...	Portugal	2021-02-01 14:33:56
...	...	...	...
232837	alterar o domínio. Meu domínio é mgxlimpezap...	Brasil	2013-08-30 19:27:28
232838	Boa tarde, estive a fazer listas para t...	Portugal	2013-08-30 18:46:30
232839	listas	Portugal	2013-08-30 18:46:13
232840	alterar forma de pagamento	Brasil	2013-08-30 18:43:31
232841	Amigo, não sei direito, mas acho que não está ...	Brasil	2013-08-30 18:41:39

Figura 6.28: Extrato do ficheiro de pesquisas

	codigo	group
0	afghanistan	interEN
1	albania	interEN
2	algeria	interEN
3	andorra	interEN
4	angola	interPT
...	...	...
226	aruba	interEN
227	ascension	interEN
228	bermuda	interEN
229	martinique	interEN
230	isle_of_man	interEN

Figura 6.29: Extrato do ficheiro de países

De um total de 232 mil resultados, obtiveram-se apenas 22 mil pesquisas realizadas em países que a língua falada é português e foram realizadas depois do ano de 2019 (apenas cerca de 10% dos resultados). Removendo os valores duplicados, pesquisas que são apenas números (ex: "44445", "999-999-999"), isto é, que não contêm letras, e linhas que são vazias, obtêm-se apenas um total de 14 mil resultados (figura 6.30).

```
# Remove empty values, number only strings and duplicates from a list
# @param l list to clean
# @return cleaned list
def clean_list(l):
    l = [str(e).strip() for e in l if str(e).strip() != '' and str(e).isdigit() == False]
    return list(dict.fromkeys(l))
```

```
dataPT = clean_list(dfPT['CONTEUDO'])
print(f'TOTAL = [{len(dataPT)}] results')
```

```
TOTAL = [14009] results
```

Figura 6.30: Parte III do *script*

Com estes dados, procedeu-se depois para a remoção de links *HyperText Transfer Protocol* (HTTP), *tags* HTML e emails pessoais (através das bibliotecas Re e BeautifulSoup - figura 6.31). Em seguida, selecionando apenas as frases que contém mais que 4 palavras (requisito para garantir que a biblioteca de Detect funciona da melhor forma), removem-se todas as frases que não são detetadas como português, obtendo um total de cerca de 13 mil resultados (figura 6.32).

```
# Filter HTTP/HTTPS links
dataPT = [re.sub(r'http\S+', 'url', e) for e in dataPT]

# Filter HTML content
dataPT = [BeautifulSoup(e, "html.parser").get_text() for e in dataPT]

# Filter emails
dataPT = [re.sub('\S*@\S*\s?', 'email', e) for e in dataPT]
```

Figura 6.31: Parte IV do *script*

```
# Remove sentences that are bigger than 4 words
# (to ensure that detection is accurate) and not in Portuguese
newDataPT = []
for e in dataPT:
    # We dont care about this sentences
    if len(e.split()) > 4 and detect(e) != detect_lang:
        continue
    newDataPT.append(e)
dataPT = newDataPT
print(f'TOTAL = [{len(dataPT)}] results')
```

```
TOTAL = [13423] results
```

Figura 6.32: Parte V do *script*

Com uma lista de frases mais corretas, como muitos clientes escrevem parágrafos e de forma a não descartar estas tipo de frases, recorreu-se à biblioteca Nltk, para tentar separar estes parágrafos em elementos individuais, pois podem conter frases relevantes (ex: "Boa tarde. O meu nome é João. Preciso de ajuda a criar uma lista de contatos. Também precisava de ajuda a aceder à página da minha conta."). A figura 6.33 descreve essa mesma divisão dos parágrafos; seguido da correção de frases que não estão escritas de forma correta (ex: "Olá.Boa tarde.Preciso ajuda.") - como não existe um espaço entre o ponto final e o início da próxima frase, a biblioteca não consegue separar; por fim, dividindo-se novamente as frases e obtendo-se um total de quase 21 mil resultados válidos!

```
# Separate sentences from original sentence ("Today i eat. Tomorrow i sleep." => ["Today i eat.", "Tomorrow i sleep."])
dataPT = [nltk.tokenize.sent_tokenize(e, language=nltk_lang) for e in dataPT]
# Flatten
dataPT = [e for sublist in dataPT for e in sublist]
print(f'TOTAL = [{len(dataPT)}] results')
```

```
TOTAL = [18012] results
```

```
# Try to fix wrong syntax senteces with regex (3 Groups)
# Group 1 is before the punctuation (\w{2}\)*). It can end as "Today (i eat). So..."
# Group 2 is the punctuation which can be [!?.] and can repeat more than once so it is ([.!?]{1,}).
# Group 3 is the next word after punctuation which can anywhere be 1 or more character word like "I" (\w{1}) .
dataPT2 = []
for e in dataPT:
    # Loop until all sentences are splitted
    while True:
        e2 = re.sub(r'(\w{2}\)*)([.!?]+)(\w+)', r'\1\2 \3', e)
        if e2 == e:
            break
        e = e2
    dataPT2.append(e)
dataPT = dataPT2
```

```
# Separate sentences from original sentence again
dataPT = [nltk.tokenize.sent_tokenize(e, language=nltk_lang) for e in dataPT]
# Flatten
dataPT = [e for sublist in dataPT for e in sublist]
print(f'TOTAL = [{len(dataPT)}] results')
```

```
TOTAL = [20922] results
```

Figura 6.33: Parte VI do *script*

Ao contrário dos *emails*, *links* e *tags* HTML, que têm um formato bem definido, e que, pela análise visual inicial que foi realizada, nunca tinham grandes variações, os números telemóvel

ou fixos encontravam-se em diversos formatos (ex: (+365) 987654321; 987654321; +351-987654321; 987-654-321; 98 765 4321). Isto torna a filtragem, deste tipo de dados, muito difícil e complexa. De tal modo, optou-se primeiro (ver figura 6.34) por filtrar todos os acentos/carateres especiais, converter para minúsculas e remover toda a pontuação das frases, e só depois remover os números telefónicos (juntando primeiro todos os números com espaços entre eles - figura 6.35).

```
# Filter unicode chars (accents, ç, etc)
for i, e in enumerate(dataPT):
    dataPT[i] = ' '.join([unidecode.unidecode(w) for w in e.split()])

# Filter ponctuation and convert to lower case
dataPT = [e.translate(str.maketrans(' ', ' ', string.punctuation)).lower() for e in dataPT]
```

Figura 6.34: Parte VII do *script*

```
# Remove all spaces between numbers
dataPT = [re.sub(r'(\d)\s+(\d)', r'\1\2', e) for e in dataPT]
# Filter phone numbers (12 numbers or 9 number in a row)
dataPT = [re.sub(r'[0-9]{12}|[0-9]{9}', 'numero', e) for e in dataPT]
```

Figura 6.35: Parte VIII do *script*

Após ter um *dataset* mais robusto, procedeu-se à remoção de todas as palavras que têm acima de 20 letras, para todas as frases, pois, no dicionário da língua portuguesa não serão relevantes[12] (isto garante também que, informações sensíveis, como chaves api dos clientes, são removidas).

Depois, aplicam-se 2 filtros seguidos às frases (processo da figura 6.36), segundo dicionários: remoção de expressões detetadas como comuns, visualmente (ex: "Bom dia", "Boa tarde", etc), e de *stop-words* (palavras comuns de ligação, que não são necessárias para treinar o modelo).

```
# Filter unwanted common expressions ("Bom dia", "Boa tarde"...)
with open('egoi-common-expr.txt','r') as f:
    common_exp = [w.strip() for w in f.readlines()]
for i,_ in enumerate(dataPT):
    for c in common_exp:
        dataPT[i] = dataPT[i].replace(c, '').strip()

# Filter stop-words ("ser", "ter"...) and words that are 1 char only
# print(stop_words) to see them
stop_words = nltk.corpus.stopwords.words(nltk_lang)
stop_words.extend(('ser', 'ter', 'têm', 'havia'))
for i,e in enumerate(dataPT):
    dataPT[i] = ' '.join(w for w in e.split() if not w in stop_words and len(w)>1)
```

Figura 6.36: Parte IX do *script*

Como foi de notar, que muitas das frases continham erros ortográficos, a próxima etapa passou por tentar corrigir os erros ou, se de facto fosse um erro, remover a palavra. Para tal, num primeiro passo foi criada uma função que exporta para um ficheiro de texto todas as palavras que não são conhecidas pela biblioteca do SpellChecker como português ou inglês. Esta biblioteca procura a palavra em ambos os dicionários (PT e EN), e caso não encontre uma correção, utilizando uma distância de Levenshtein de 1 (1 variação - ex: trocar uma letra "ecoi" -> "egoi"), é adicionada ao ficheiro de erros (ver 6.38).

Com este ficheiro de erros, foram analisadas manualmente palavras que não eram erros, mas sim termos utilizados na E-goi (ex: *templates*, *webpush*, *smartsms*). Adicionando estas

[12] https://www.dicio.com.br/maior-palavra-da-lingua-portuguesa/

palavras ao ficheiro "egoi-dictionary.txt", e carregando esse ficheiro para o SpellChecker, estas palavras deixam de ser marcadas como um erro.

	Campanhas>PostSocial	Campanhas>Email	Campanhas>Transacional (Slingshot)	Campanhas>Ads	Campanhas>WebPush	Campanhas>Push	Campanhas>Voz
0	post	anular	transaccional	ads	webpush	push	criar
1	social	split-test	transacional	anúncio	enviar	criar	voz
2	redes	A/B	slingshot	google	campanha	enviar	campanha
3	sociais	ensaio	transaccionais	gmail	web push	campanha	multicanal
4	postar	AB	ecommerce	display	omnicanal	multicanal	omnicanal
...	...	...	...	...	...	...	...
262	NaN	templates de email	NaN	NaN	NaN	NaN	NaN
263	NaN	modelo de widget	NaN	NaN	NaN	NaN	NaN
264	NaN	template de widget	NaN	NaN	NaN	NaN	NaN
265	NaN	email builder	NaN	NaN	NaN	NaN	NaN
266	NaN	campanha de email	NaN	NaN	NaN	NaN	NaN

Figura 6.37: Extrato das palavras contextuais à E-goi, extraídas dos artigos do LiveAgent

Além deste processo manual, e de forma a tentar agilizá-lo, foi criado um outro *script* que, utilizando a API do LiveAgent, guardou no ficheiro do dicionário da E-goi todas as *keywords* utilizadas em todos os artigos da E-goi até ao momento (cerca de 1000 termos - ver 6.37).

```
# Initialize spellcheckers, tolerate max 2 mistakes
ptSP = SpellChecker(language=spell_lang, distance=2)
enSP = SpellChecker(language=spell_extra, distance=2)
# Load e-goi dictionary for both spellcheckers
ptSP.word_frequency.load_text_file('./egoi-dictionary.txt')
enSP.word_frequency.load_text_file('./egoi-dictionary.txt')
```

```
# Grab every single word
wordsList = [e.split() for e in dataPT]
# Flatten list of words and remove duplicates
wordsList = list(dict.fromkeys([e for sublist in wordsList for e in sublist]))
# Grab lists of unknown PT and EN words from previous list
unknownPT = list(ptSP.unknown(wordsList))
unknownEN = list(enSP.unknown(wordsList))
# Insertect lists of common unknown words (since we will be checking both) in a single list and remove duplicates
unknownWords = list(set(unknownPT).intersection(unknownEN))
```

```
# We can run this to get words that dictionary doesn't recognize and add them to the 'egoi-dictionary', hence making them known and not an error
# This will allows us to make corrections more accurate, since this words will not be changed or removed
def get_word_errors():
    with open('spell_errors.txt','w+') as f:
        for w in unknownWords:
            # distance=2 takes to long
            ptSP.distance = 1 #if len(w) <= 5 else 2
            enSP.distance = 1 #if len(w) <= 5 else 2
            c = ptSP.correction(w)
            c2 = enSP.correction(w)
            if c == w and c2 == w:
                f.write(f'{w}\n')
# Uncomment this line to see which words where detected as errors and have no suggestion, thus being removed next
# get_word_errors()
```

Figura 6.38: Parte X do *script* (Passo 1)

Com três dicionários (PT, EN e o da E-goi) prontos, percorreram-se novamente todas as frases, tentando corrigir as palavras marcadas com erro (ver 6.39), removendo-as caso não fosse possível corrigi-las.

```
for i, e in enumerate(dataPT):
    newSentence = []
    # Iterate sentences
    for w in e.split():
        # If the word is not in any of the dictionaries
        if w in unknownWords:
            # Reduce distance depending on word length (average word length 4.7chars=5chars)
            ptSP.distance = 1 #if len(w) <= 5 else 2
            enSP.distance = 1 #if len(w) <= 5 else 2
            # Try to fix word. If it's not able to, remove it // THIS TAKES TO LONG WITH "distance=2"
            correctWordPT = ptSP.correction(w)
            # If correction is different than original word, add to new sentence else do nothing (remove it)
            if w != correctWordPT and correctWordPT not in stop_words:
                newSentence.append(correctWordPT)
            else:
                correctWordEN = enSP.correction(w)
                if w != correctWordEN and correctWordEN not in stop_words:
                    newSentence.append(correctWordEN)
        else:
            # If its a valid word just add it
            newSentence.append(w)
    # Replace sentence (filter unicode chars, because of correction method from Spellchecker) and filter ponctuation left by unidecode
    dataPT[i] = ' '.join([unidecode.unidecode(w).translate(str.maketrans(' ', ' ', string.punctuation)) for w in newSentence])
```

Figura 6.39: Parte X do *script* (Passo 2)

Finalmente, e com as frases praticamente prontas para ser agrupadas (em 6.40), removeram-se todos os números, assumindo 6 digitos seguidos como o id de um utilizador E-goi, 8 digitos como datas e os restantes apenas foram removidos, dado que aquelas sequências de números não são relevantes para o modelo e poderão ser substituídos mais tarde. Após a remoção de todos os números, removeram-se novamente todos os valores duplicados e linhas vazias, obtendo-se um total de cerca de 16 mil resultados (menos 5 mil que a contagem anterior) e removendo todas as frases que não contêm mais que 2 palavras ou menos que 21 obtiveram-se um resultado final de cerca de 11 mil resultados.

```
dataPT2 = []
for s in dataPT:
    # Assume 6 digit numbers as userIds
    s = ' '.join(re.sub(r'^[0-9]{6}$', 'userId', w) for w in s.split())
    # Assume 8 digit numbers as dates
    s = ' '.join(re.sub(r'^[0-9]{8}$', 'data', w) for w in s.split())
    # Replace other numbers with XXX
    s = ' '.join(re.sub(r'[0-9]+', 'xxx', w) for w in s.split())
    dataPT2.append(s)
dataPT = dataPT2
```

```
# Final clean
dataPT = clean_list(dataPT)
print(f'TOTAL = [{len(dataPT)}] results')
```

```
TOTAL = [16068] results
```

```
# Select sentences that contain between 3 to 20 words only
sentences = [s for s in dataPT if len(s.split()) >= 3 and len(s.split()) <= 20]
print(f'TOTAL = [{len(sentences)}] results')
```

```
TOTAL = [11147] results
```

```
df = pd.DataFrame(sentences)
df.to_csv('E-goi-Clean-Pre-Processed.csv', index=False, header=None)
```

Figura 6.40: Parte XI do *script*

Com este total, a próxima fase passou por agrupar as frases por categoria, seguindo a estrutura de artigos da E-goi (no LiveAgent), para poderem ser criados os *Intents* no Botpress.

Fase 3) Agrupamento por Categorias/Intents

Depois de analisada a estrutura de artigos do HelpDesk da E-goi, considerou-se uma boa forma de organizar os *Intents* seguir a organização dos grupos de nomes utilizados neste sistema (ver figura 6.41). A plataforma da E-goi é complexa e implica ter um conhecimento multissetorial de vários domínios, para se efetuar o mapeamento das frases a incluir em cada categoria, bem como defini-las em primeiro lugar. Deste modo, garante-se uma consistência ao longo dos sistemas.

O IBM Watson é uma ferramenta que tem a capacidade de agrupar frases ou *Intents* por categoria (fornecendo recomendações de frases a partir de um ficheiro CSV), um trabalho que, se efetuado manualmente, seria muito mais vagaroso e difícil. Embora seja um serviço pago, foi possível obter um *Trial* de 30 dias, para experimentação. E, apesar de esta ferramenta não ser perfeita, consegue agrupar frases semelhantes até um limite de 20 por categoria.

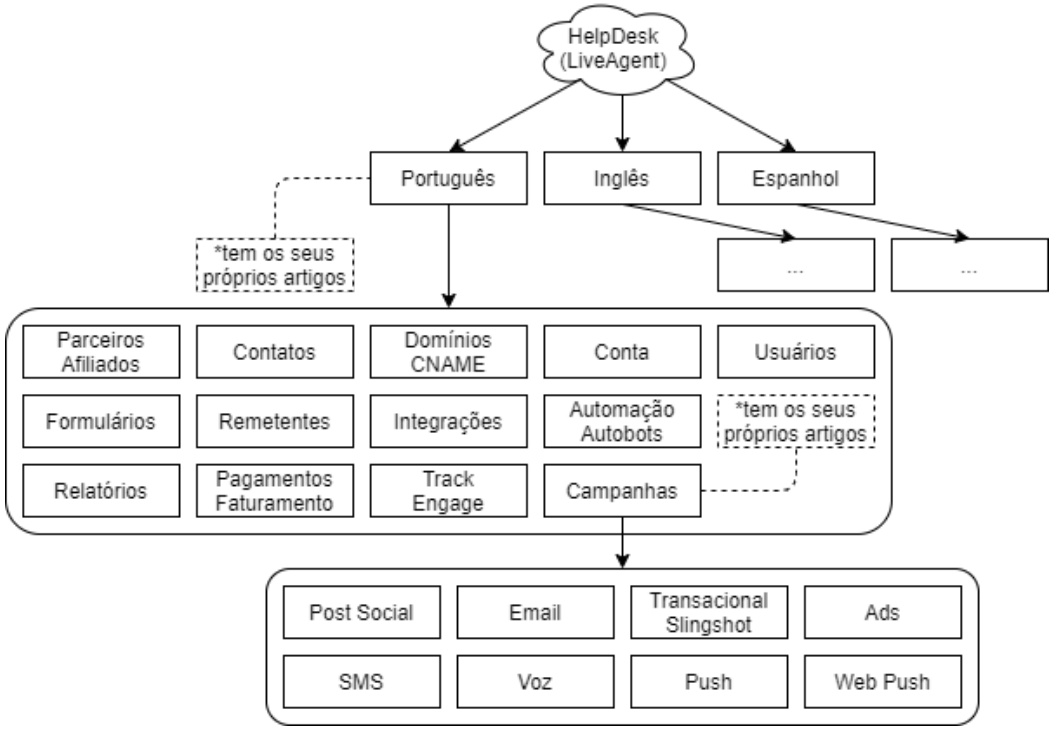


Figura 6.41: Estrutura de artigos retirada do HelpDesk da E-goi

Segundo a documentação[13] desta ferramenta, todos os dados sensíveis (nomes, emails, *ids* clientes) devem ser removidos das pesquisas, frases com menos de 3 palavras ou mais de 20 devem ser removidas, frases duplicadas devem ser removidas, não devem conter vírgulas e, por fim, devem existir pelo menos 100 frases (melhor resultados com mais de 500). Cumpridos todos os requisitos na fase anterior, submeteu-se no Watson um ficheiro CSV com um coluna, contendo todas as frases a agrupar a partir da fase anterior.

[13]https://cloud.ibm.com/docs/assistant?topic=assistant-intent-recommendations

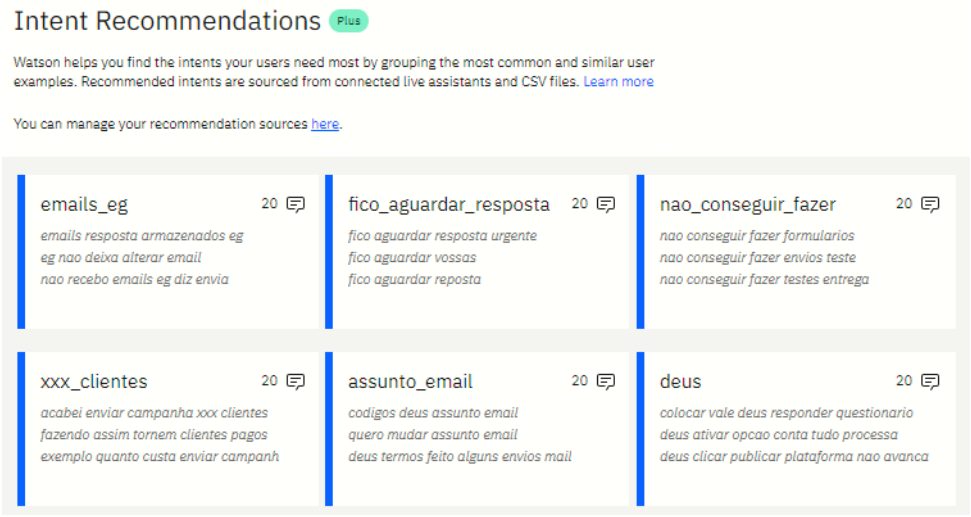


Figura 6.42: Watson - Extrato de recomendações de *Intents* e frases

Na figura 6.42 é possível ver um exemplo dos *Intents* sugeridos que, apesar de não serem perfeitos, quer por erros de pré-processamento como por exemplo, a palavra "deus" que foi mal corrigida, porque os clientes pesquisaram com "deps" (em vez da forma correta "depois"), quer por frases mal categorizadas, colocam grande parte das frases (ex: "nao_conseguir_fazer") no mesmo *Intent*.

Assim, manualmente, foram exploradas as diversas sugestões, criadas as categorias e agrupadas as frases na respetiva categoria. Como o Botpress sugere pelo menos 10 frases por categoria (quantos mais exemplos melhor), agruparam-se em média 50 frases em cada. Depois, exportaram-se os resultados para um ficheiro Excel e, utilizando novamente Python, criaram-se os ficheiros no formato usado no Botpress, movendo-os para a pasta do projeto e fazendo *push* para o servidor de produção (uma vez que não existe forma de efetuar o *upload* de uma grande quantidade de frases, sem ser copiar e colar).

Fase 4) Slot Tagging e Custom Entities

Assim que cada *Intent* tinha as suas frases associadas, para cada frase de cada categoria, começou-se a efetuar *Slot Tagging*. Como definido pela documentação do Botpress[14] os *sltos* podem ser vistos como parâmetros necessários para completar uma ação associada a um *Intent*. Por exemplo, não é possível marcar um voo, sem especificar o destino.

Com esta lógica, definiu-se para cada frase quais as "palavras-chave", que se utilizariam para efetuar as pesquisas no ElasticSearch. A ideia principal seria utilizar o nome da categoria e os *slots* identificados para efetuar pesquisas no Elastic, numa única frase separados por espaços (ex: "conta bloqueada").

É necessário ainda referir que, cada *slot* tem que ter uma de três opções: uma Entitidade de sistema associada (ex: data, número de telemóvel, email, *url*, etc), uma lista de *Entities* associada (tipicamente uma lista de sinónimos) ou um padrão *Regex* (expressão regular).

Como é possível verificar nas figuras 6.43 e 6.44, o *Intent* relacionado com problemas das contas, é utilizado quando, por norma, os clientes ficam com a conta bloqueada/limitada ou querem eliminar ou recuperar a mesma. A figura do 6.44, é a possibilidade de converter

[14] https://botpress.com/docs/nlu/skill-slot

qualquer palavra que, por exemplo no caso do eliminar, esteja relacionada com o cliente querer apagar a conta.

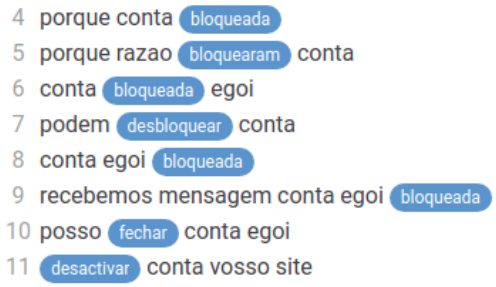


Figura 6.43: Excerto de frases do *Intent* relacionado com a conta dos clientes

Figura 6.44: *Entity* da conta para sinónimos do possível estado

Como os artigos que a E-goi tem, referem apenas a palavra "eliminar", pesquisando no ElasticSearch pela palavra "acabar" ou "terminar" não devolveria nenhum artigo. Deste modo, garante-se que, se o Botpress identificar algum sinónimo desta palavra, será convertido para a "raíz" ou para a palavra a azul na figura 6.44.

Um caso prático pode ser visto como:

1) O utilizador pesquisa por "Bom dia, gostaria de encerrar a minha conta da E-goi se possível";
2) O Botpress categoriza a frase como "Conta";
3) O Botpress identifica os *slot* "Estado da Conta";
4) O Botpress converte a palavra "encerrar" para "eliminar";
5) O Botpress envia a frase "conta eliminar" para o ElasticSearch;
6) O ElasticSearch pesquisa as palavras "conta" e "eliminar" no título e conteúdo nos artigos, devolvendo por ordem decrescente de *scores*/pontuação os artigos encontrados.

Fase 5) Fase Final e Extras

Como a quantidade de categorias da E-goi é muito grande (mais de 20), optou-se por realizar a fase anterior utilizando código JavaScript, de forma dinâmica, e não pela interface gráfica (através de nós específicos). Assim garante-se que o trabalho futuro passa apenas por: criar um novo *Intent* com frases associadas e realizar novamente a fase 4.

Além das frases identificadas através das fases iniciais (1, 2 e 3), foram também acrescentadas novas, analisando os artigos da E-goi em conjunto com a equipa de UXA, uma vez que não constavam nas pesquisas dos utilizadores e poderão todavia surgir no futuro.

Durante todo o processo da fase 4, treinou-se várias vezes o modelo, garantindo que as frases eram colocadas na categoria correta e que, caso não fossem, eram adicionadas a categoria adequada.

6.5 Outros projetos

Foram também realizadas implementações/alterações nos projetos da Comunidade, do Services, do Admin e criada uma API para realizar pesquisas na base de dados do Botpress (ver 5.2).

No caso da Comunidade, foram realizadas as seguintes tarefas:

- antes, era possível na E-goi colocar o email da conta como nome de utilizador, o que levava a que muitos utilizadores apresentassem na comunidade o email da conta (isto poderia fazer com que outros utilizadores aproveitassem esse email para fazer o envio de conteúdo de *spam* ou *phishing*). Foi adicionado portanto uma validação, para quando o utilizador alterar o nome não poder especificar o seu email e adicionado um aviso para os utilizadores que assim têm a conta, para atualizarem o nome associado (sendo que no processo de criação de conta foi também adicionada a mesma validação ao campo do nome);
- um enorme *revamp* ao estilo da Comunidade que existia, pois não se enquadrava ao estilo do novo Widget de ajuda (desta forma, mantém-se a consistência entre as duas aplicações);
- corrigidas funcionalidades que não estavam operacionais, como por exemplo comentários que não apareciam ou traduções que não estavam a ser efetuadas.

Quanto ao Services, além do trabalho realizado para o ElasticSearch descrito anteriormente, como grande parte dos *endpoints* já existiam, foram apenas modificados para se adequarem ao WidgetHelp (por exemplo, acrescentados/removidos parâmetros dos pedidos ou acrescentados novos, dentro dos *Resources* existentes). Contudo, foi criado um novo *Resource*, intitulado de *ChatResource* (fig. 6.45), para comunicação entre o Widget-Help e Botpress.

```
14  /**
15   * @SWG\Resource(
16   *     apiVersion="1.0",
17   *     swaggerVersion="1.2",
18   *     basePath="/api/knowledgebase",
19   *     description="Chat with the chatbot",
20   *     resourcePath="/chat"
21   * )
22   */
23  class ChatResource extends EgoiAbstractRestFullController
24  {
25
26      /**
27       * @SWG\Api(
28       *     description="Chat operations",
29       *     path="/chat",
30       *     @SWG\Operation(
31       *         method="POST",
32       *         summary="Post a new message to chatbot",
33       *         @SWG\Parameters(
34       *              @SWG\Parameter(
35       *                  name="text",
36       *                  paramType="body",
37       *                  description="Message to send",
38       *                  required=true,
39       *                  type="text"
40       *              ),
41       *         ),
42       *         @SWG\ResponseMessage(code=200, message="OK"),
43       *         @SWG\ResponseMessage(code=403, message="Forbidden"),
44       *         @SWG\ResponseMessage(code=500, message="Internal Error"),
45       *         @SWG\ResponseMessage(code=503, message="Botpress is not available")
46       *     )
47       * )
48       */
49      public function create($data)
50      {
```

Figura 6.45: Excerto do código do *ChatResource* no projeto Services

Este Resource pode ser invocado através de um POST *request*, recebendo no *payload* o texto a enviar para o *chatbot*. Caso o pedido não tenha a *apikey* ou *token* de acesso é rejeitado com um código *403*. Caso o Botpress não esteja disponível, de modo a cumprir o UC-20 (sistema de alarmística) é enviado um email para os adminstradores de sistemas com o ficheiro e linha em que ocorreu (com a mensagem ex: "Botpress indisponível - verificar"). Dentro do método *create()* (método utilizado para pedidos POST) é invocado o método "sendMessage()", que está no serviço "ChatService" (fig. 6.46).

```php
    public function sendMessage($data)
    {
        $botpressUrl = $this->getServiceLocator()->get('config')['botpress_url'];

        $ch = curl_init();
        curl_setopt($ch, CURLOPT_URL, "{$botpressUrl}/api/v1/bots/" . self::BOTPRESS_BOT_NAME . "/converse/{$data['conversationId']}");
        curl_setopt($ch, CURLOPT_POST, 1);
        curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode([
            'text' => $data['text'],
            'type' => 'text'
        ]));
        curl_setopt($ch, CURLOPT_HTTPHEADER, array(
            "Content-Type:application/json"
        ));
        curl_setopt($ch, CURLOPT_TIMEOUT, 15);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
        $res = curl_exec($ch);
        if (curl_error($ch)) {
            mail(
                [redacted]
            );
        }
        curl_close($ch);

        return $res;
    }
}
```

Figura 6.46: Excerto do código do *ChatService* no projeto Services

Dentro deste método é invocada a API do Botpress enviando para o id da conversa do utilizador o texto que este submeteu, através de um *curl request*. Depois de processados os dados pelo *chatbot*, são retornados ao *Resource* e gerado novamente um alarme caso a resposta do *bot* não seja a esperada.

Já o projeto do Admin destinou-se a apresentar uma listagem das pesquisas efetuadas no *widget*, exibindo se as mesmas foram marcadas como úteis e quais os resultados retornados, para poder otimizar o *chatbot*. Como o Botpress não tem, por defeito, uma forma de as consultar, criou-se uma nova aplicação em Python para ler da base de dados do Botpress e que permita fazer queries à mesma.

Esta aplicação foi desenvolvida com recurso à biblioteca Falcon[15], uma biblioteca de alta performance para construção de aplicações *Representational State Transfer* (REST). Depois, através do Admin são enviados pedidos para esta API e apresentados os resultados na forma de uma tabela, que pode ser exportada (cumprindo o requisito UC-23). Sendo ambos projetos internos, não será apresentado o código relativo ao desenvolvimento destas aplicações.

[15] https://pypi.org/project/falcon/

Capítulo 7

Experimentação e Avaliação

No capítulo de Experimentação e Avaliação avalia-se a solução desenvolvida (parcialmente ou totalmente), percebendo se os problemas/objetivos identificados foram cumpridos, quantificando também a qualidade da solução proposta. É expectável que a solução desenvolvida seja questionada e avaliada, ao longo do processo de integração do produto, percebendo se os requisitos estão a ser cumpridos e se a qualidade do projeto está dentro das expetativas.

7.1 Especificação de Hipóteses

Uma hipótese consiste em formular, avançar, construir ou desenvolver uma ideia, que depois é aceite/refutada através da análise dos resultados das experiências realizadas [47]. Começando por definir a teoria ou formulando uma hipótese, pretende-se avaliar as seguintes ideias:

- Hipótese 1) Cumprimento de pelos menos 90% dos requisitos definidos para a realização do projeto;
- Hipótese 2) Sucesso de 100% dos testes de *software* apresentados;
- Hipótese 3) Aumento da eficácia das respostas obtidas pelo novo *chatbot*.

Caso estas hipóteses sejam asseguradas como válidas, a garantia que o projeto desenvolvido está apto para ser disponibilizado para clientes selecionados, e mais tarde para todos os clientes E-goi, é grande. O intuito das hipóteses definidas é demonstrar que o projeto não apresenta falta de qualidade/funcionalidades, não contém falhas/erros/bugs e inclui garantias que a solução funciona como esperado, face aos objetivos que foram definidos.

A primeira hipótese é resguardada com a apresentação faseada do modelo *Quality Evaluation Framework* (QEF), isto é, apresentar o modelo em dois momentos distintos do desenvolvimento. A seguinte, é assegurada através de testes de *software*. E finalmente, a última é apresentada, com recurso um modelo estatístico, comparando com a solução que existia anteriormente.

7.2 Avaliação de Qualidade (Hipótese 1)

Considerou-se uma boa estratégia seguir o modelo QEF na subseção 7.2.1, uma vez que este é um modelo conceptualmente desenvolvido, com base nos paradigmas de engenharia de software, que valida e garante a qualidade de um sistema [48]. Este modelo foi desenvolvido em duas fases, primeiro numa fase inicial, que ocorreu em Fevereiro, isto é, no decorrer

do desenvolvimento, e numa fase final, quando foi terminado, de modo a efetuar-se uma comparação da qualidade do sistema inicial *versus* final.

7.2.1 Modelo QEF

Apesar de terem sido identificados e representados os requisitos fundamentais, para o negócio, através de diversos modelos, nomeadamente Casos de Uso e Modelo de Domínio nos capítulos 4 e 5, surgem vários problemas:

> *"Business process modeling is an important part of information systems design as well as of any business engineering or reengineering activity. Business process modeling languages provide standard ways of presentation and communication between different stakeholders. [...] This objective raises two major issues, (a) the identification of the quality factors relevant to business processes, and (b) the definition of the metrics that provide a means for objectively measuring quality of business processes."* ([49] - Heidari F, 2013)

Os dois problemas identificados no livro [49], a) - a identificação de fatores de qualidade para os processos de negócio - e b) - a definição de métricas que, efetivamente possam medir esses fatores -, podem ser respondidos através do modelo QEF.

Esta *framework* conhecida como QEF, permite especificar várias dimensões de qualidade, que categorizaram os processos de negócio. O modelo define que deve ser criado um cenário de qualidade, composto por: propósito/objetivos, casos de uso (requisitos funcionais), requisitos não funcionais, dimensões relevantes (funcionalidade, fiabilidade, manutenibilidade...) e quais os critérios de avaliação para cada requisito, bem como os procedimentos e ferramentas de avaliação.

Fase 1)

Como os objetivos se encontram definidos em 1.3 e os requisitos funcionais/não funcionais estruturados em 4.2, foram realizadas várias tabelas que abordam os restantes passos do cenário de qualidade. Com recurso à ferramenta Excel, foi elaborada a figura 7.1, que descreve as dimensões relevantes, os fatores para cada dimensão e os requisitos atribuídos a cada fator.

As três dimensões relevantes, identificadas para o contexto deste projeto, são distintamente: Funcionalidade, Adaptabilidade e Usabilidade. Na dimensão de Funcionalidade agruparam-se os requisitos segundo os fatores: Integração de Projetos - todos os requisitos relacionados com integração de código de outros projetos com o projeto Widget-Help -, Mecanismo Conversa - todos os requisitos que estejam relacionados com a conversa entre o utilizador e o *chatbot* -, Consulta Informação - todos os requisitos que sejam de simples listagens ou consulta -, e Outro(s). Quanto à Adaptabilidade e Usabilidade, foram agrupados segundo os fatores, respetivamente: Suportabilidade - funcionamento do Widget-Help em diferentes contextos -, Manutenibilidade - garantia de que o projeto funciona durante um longo período e é facilmente alterável -, Desenho - garantia de que a aplicação é fácil de entender para quem a utiliza - e Prevenção Falhas - certeza de que a aplicação funciona sem falhas ou com menos falhas possíveis.

q	D	Qi	Dim	Q_j	W_{ij} [0,1]	Fator	rw_{jk} [0, 10]	Requisito	wf_k [0, 100]
45%	1,10	53,14	Funcionalidade	67,24	0,29	Integração de Projetos	8	FIP01 - Integração com o projeto da Comunidade	50
							10	FIP02 - Integração com o LiveAgent/Botpress (Criação de Tickets)	100
							10	FIP03 - Integração com o LiveAgent (Continuação do fluxo do Ticket)	100
							10	FIP04 - Integração com o LiveAgent (Live-chat)	50
							10	FIP05 - Integração com o Botpress (Fluxo de conversa/Troca de mensagens)	50
							10	FIP06 - Integração do Widget com os projetos da E-goi como Web Component	50
				25	0,43	Mecanismo Conversa	10	FMC01 - Melhoria da pesquisa de artigos existente	0
							6	FMC02 - Consulta do histórico de conversas	0
							8	FMC03 - Persistência do estado de uma conversa	100
							10	FMC04 - Identificação NLP no Botpress	0
							6	FMC05 - Implementar a possibilidade de anexar ficheiros	100
							4	FMC06 - Implementar o envio de emojis	100
							8	FMC07 - Implementar persistência de frases incompreendidas	0
							10	FMC08 - Implementar sistema de alarmística	0
							10	FMC09 - Limitar o número de mensagens visíveis pelo utilizador	0
				87,5	0,24	Consulta Informação	8	FCI01 - Listar/Consultar tickets	100
							8	FCI02 - Listar/Consultar notificações	100
							4	FCI03 - Filtro de pesquisa de Tickets	0
							6	FCI04 - Distinção entre notificações lidas/por ler	100
							6	FCI05 - Número de notificações/tickets não abertos	100
				50	0,05	Outro(s)	8	FO01 - Disponibilizar o widget para utilizadores não autenticados	50
		34,22	Adaptabilidade	20	0,60	Suportabilidade	4	AS01 - A aplicação funciona em múltiplos browsers	100
							8	AS02 - A aplicação está disponível em vários idiomas	0
							8	AS03 - A aplicação deve ser testada em vários níveis	0
				55,56	0,40	Manutenibilidade	8	AM01 - O projeto deve ser bem documentado para futuras referências	0
							10	AM02 - O projeto do Botpress deve ser fácil de manter/alterar	100
		25	Usabilidade	33,33	0,75	Desenho	8	UD01 - A aplicação deve ser responsiva	100
							10	UD02 - A aplicação deve ser intuitiva	0
							6	UD03 - A aplicação deve ser fácil de memorizar	0
				0	0,25	Prevenção Falhas	8	UPF01 - A aplicação deve ter uma baixa taxa de erros	0

Figura 7.1: Modelo QEF com as dimensões, fatores e requisitos (fase 1)

Em anexo (ver B), é possível consultar as restantes tabelas que explicitam as métricas para cada requisito, bem como os critérios de avaliação (em percentagem). Considerou-se suficiente avaliar os requisitos com a dimensão Funcionalidade como 0% Não Implementado, 50% Parcialmente Implementado e 100% Implementado. Já as dimensões Adaptabilidade e Usabilidade contêm as suas próprias definições de avaliação, com exceção da UD02, seguindo as três percentagens definidas anteriormente (0, 50 e 100). É importante mencionar que todo o documento Excel foi apresentado à organização e aprovado pela mesma.

Retomando a análise da figura 7.1, a primeira coluna representa a qualidade do projeto (em percentagem), no momento em que foi avaliado, o que significa que se obteve um valor total de 45% de desempenho qualitativo no estado atual (27 de fevereiro). Assim que o projeto for concluído, espera-se que este valor seja muito maior, e que se aproxime do valor total (100%), isto é, que todos ou quase todos os requisitos propostos tenham sido cumpridos.

Fase 2)

Pretende-se nesta fase efetuar uma comparação com os resultados obtidos na primeira, de modo a validar se houve uma melhoria da qualidade da solução proposta.

As dimensões nesta nova versão mantiveram-se, assim como os fatores, existindo apenas mudanças em alguns dos requisitos e também o acréscimo de novos, nomeadamente:

- Alterado(s) - FMC07;
- Novo(s) - FIP07, FMC09 e FO02.

Na nova tabela criada, no anexo B (última figura), é possível constatar que existe uma grande diferença de qualidade para a versão inicial, sendo que o valor inicial de 45% subiu para 96%, o que é muito próximo da conclusão de todos os objetivos propostos no capítulo 4, ou seja, 100%.

Considera-se que não se atingiu o valor máximo, porque em relação ao FMC04 (identificação NLP - definido como 50% cumprido), este é um trabalho que deve ser continuamente melhorado, uma vez que, irão sempre surgir novos temas e artigos, sobre os quais o *bot* não terá conhecimento. Contudo, mesmo com os temas atuais considera-se que não tenham sido cobertas a totalidade das dúvidas dos utilizadores, visto haverem questões para as quais o *chatbot* não conseguirá devolver artigos úteis (mesmo testando para a maioria dos casos). De notar também que, há dois requisitos funcionais que não foram cumpridos, o que faz decrescer a qualidade do projeto desenvolvido (FMC09 e FCI03).

7.3 Testes de Software (Hipótese 2)

Embora o compilador aceite o programa como se estivesse bem desenvolvido, apenas consegue detetar erros em termos de forma, não de significado [50]. Muitos erros/*bugs* são apenas descobertos por acidente, quando as aplicações são utilizados pelos verdadeiros clientes. Contudo, os erros podem ser encontrados de um modo mais sistemático e eficiente do que "experimentação aleatória", sendo esse o grande objetivo dos testes de *software*.

Segundo Sestoft em [50], são distinguidas seis características de *software*: funcionalidade, usabilidade, eficiência, confiabilidade, manutenção e suportabilidade. Neste capítulo apenas se abordam os dois primeiros, uma vez que, existem equipas responsáveis pela gestão de recursos (eficiência), *stress-testing* (confiabilidade) - geridos pela equipa de sistemas e a equipa de UXA. Em relação aos dois últimos: embora se considere que tenha sido desenvolvida uma boa solução, com documentação detalhada, caso surjam problemas, poderão ser resolvidos rapidamente (manutenção); e, quanto ao acréscimo de funcionalidades ou adaptação do sistema operacional a novos requisitos (portabilidade), pensa-se que o desenvolvimento modular facilite este processo.

7.3.1 Testes de Funcionalidade

Existem dois tipos de técnicas para a concretização de testes funcionais: *white-box testing* e *black-box testing*. A primeira, também nomeada como testes internos, consiste em efetuar um conjunto de casos de teste (coleção de *inputs* e *outputs* correspondentes) que garantem que todas as raízes do programa (condições, *loops*, validações) podem ser executadas. Já o segundo, considerado como testes externos, é normalmente realizado por equipas de *Quality*

Assurance (QA), focando-se no problema que o programa tenta resolver, não sendo a E-goi exceção à regra.

O foco desta subsecção é garantir o tipo de testes *white-box*, visto que, os testes de "caixa preta" são executados pela equipa de QA (testes de sistema e aceitação).

Testes Unitários

Para a realização de testes unitários, no projeto do Widget-Help, utilizaram-se as funcionalidades providenciadas pela *framework* de Angular[1]. Esta ferramenta disponibiliza um conjunto de módulos para poder testar a aplicação, recorrendo ao Jasmine[2] - *framework* de desenvolvimento baseada em *Behavior Driven Development* (BDD) (técnica de desenvolvimento ágil) para execução de testes Javascript.

Cada parte da aplicação (componentes, serviços, modelos) pode ser testada criando um ficheiro com o mesmo nome, mas com a extensão *spec.ts*. Correndo o comando *"ng test"* são executados todos os ficheiros *spec* e apresentados os resultados (ver figura 7.2) no *browser* que está indicado no ficheiro de configuração do *Jasmine*.

Figura 7.2: Resultados da execução dos testes unitários apresentados pelo Jasmine

[1]https://angular.io/guide/testing
[2]https://jasmine.github.io/

Observa-se que foram realizados 41 testes unitários e que a segunda hipótese se verifica como verdadeira (o sucesso de 100% dos testes de *software* apresentados).

A estrutura do ficheiro, por exemplo, para testar um serviço que realiza pedidos ao Services, começa por apresentar um método - *describe()* - que recebe como parâmetros o nome e o código a ser executado (ver 7.3). Depois, no método - *beforeEach()* - são importados os módulos que o serviço utiliza (normalmente no caso dos serviços *HttpClientTestingModule*) e declarado como *provider* o nome do serviço que se pretende testar. É injetado no ficheiro de testes, logo de seguida, o controlador de pedidos *HyperText Transfer Protocol* (HTTP) e o serviço. Ao contrário do método *beforeEach()*, o *afterEach()*, executa depois de cada teste (declarado com o método *it()*) e verifica que nenhum pedido HTTP ficou "pendurado"/*hanging*.

Os testes são declarados através do método *it()*, que recebe como parâmetros o nome do que se está a testar e o código que irá executar dentro daquele teste. Neste exemplo, espera-se que o serviço que foi criado no *beforeEach()* tenha sido bem criado/injetado (*toBeTruthy()*).

```
5  describe('notifications service', () => {
6      let service: NotificationsService;
7      let httpTestingController: HttpTestingController;
8      beforeEach(() => {
9          TestBed.configureTestingModule({
10             imports: [
11                 HttpClientTestingModule
12             ],
13             providers: [NotificationsService]
14         });
15         // We inject the Test Controller
16         httpTestingController = TestBed.inject(HttpTestingController);
17         service = TestBed.inject(NotificationsService);
18     });
19
20     afterEach(() => {
21         httpTestingController.verify();
22     });
23
24     it('should be created', () => {
25         expect(service).toBeTruthy();
26     });
```

Figura 7.3: Declaração inicial dos testes unitários

Na figura 7.4 é possível observar que é criado um objeto *mockGlsCost*, que é o resultado esperado depois de ser executado o método *getCost()*, no serviço de obtenção de custos (limpezas de listas de contactos). É feito o *subscribe*, porque este método é assíncrono e retorna um *Observable* (linha 46), sendo de seguida realizados vários "*expect()*", em que se espera que o "valor" e o "símbolo" (linha 48 e 49), sejam iguais a "0,00" e em "€", respetivamente. Valida-se ainda que, o *url* de realização do pedido é o */products/custom-services/gslcleancost*. No fim, o método *flush()*, diz ao Jasmine que o *mock* que foi criado inicialmente é para ser utilizado no retorno da resposta do serviço (uma vez que não são na realidade efetuados os pedidos HTTP, é apenas um teste).

```
30      it('should return Observable that matches the service get request', () => {
31          const mockGlsCost = {
32              "data": {
33                  "currency": {
34                      "code": "EUR",
35                      "symbol": "€",
36                      "position": "0"
37                  },
38                  "totalSubs": "0",
39                  "totalAmount": "0,00",
40                  "unitAmount": "0,0020",
41                  "hasEnoughCredit": true
42              }
43          };
44
45          service.getCost()
46              .subscribe(glsCostData => {
47                  // @ts-ignore
48                  expect(glsCostData.value).toEqual('0,00');
49                  // @ts-ignore
50                  expect(glsCostData.symbol).toEqual('€');
51                  expect(glsCostData.formatted()).toEqual('0,00€');
52              });
53
54          const req = httpTestingController.expectOne(
55              '/products/customservices/gslcleancost'
56          );
57
58          req.flush(mockGlsCost);
59      });
```

Figura 7.4: Exemplo de um *mock* HTTP de um serviço e respetivo pedido

O teste de serviços e métodos HTTP relativos, apesar de ser importante, não é suficiente. Foram realizados também métodos que processam, por exemplo, as respostas desses mesmos pedidos, como é o caso da figura 7.5. Para estas situações, a sintaxe é menos complexa, devido a ser meramente necessário invocar o método, passando os parâmetros essenciais e comparando com o resultado esperado.

```
38      it('should return string without br tags from transformContents method', () => {
39          // @ts-ignore
40          expect(service.transformContents("texto\rtexto<br>texto<br>")).toBe("texto\ntextotexto");
41          // @ts-ignore
42          expect(service.transformContents("texto\rtexto\ntexto\r\n")).toBe("texto\ntexto\ntexto\n");
43          // @ts-ignore
44          expect(service.transformContents("texto<br>texto<br>texto<br>")).toBe("texto\ntexto\ntexto\n");
45          // @ts-ignore
46          expect(service.transformContents("texto\ntexto\ntexto\n")).toBe("texto\ntexto\ntexto\n");
47      });
```

Figura 7.5: Exemplo

7.4 Testes de Hipóteses (Hipótese 3)

Como definido inicialmente pretende-se demonstrar que, as respostas do novo *widget* melhoraram em relação à janela de pesquisa que existia anteriormente. Segundo o exemplo apresentado pela universidade de saúde pública de Boston em [51], cujo o teste estatístico que se pretende realizar para comprovar a hipótese se enquadra, definem-se as seguintes hipótese nula e alternativa:

$$H_0 : \mu_{Novo} <= \mu_{Antigo}$$

$$H_a : \mu_{Novo} > \mu_{Antigo}$$

Foi retirada uma amostra de pesquisas (n=50) em português (ver C), que continham entre 3 a 10 palavras, na mesma frase, realizadas pelos utilizadores, da mesma base que se utilizou em 6.4 para treinar o modelo. Garantiu-se também que estas pesquisas não eram duplicadas, não continham erros e estavam marcadas como "não úteis". Com o conjunto de pesquisas extraído, foram definidos 3 critérios de classificação das respostas obtidas na tabela 7.1.

Critérios	**Score**
Não existem artigos para a pergunta efetuada.	0
Contém artigos, mas nenhum satisfaz a questão.	1
Contém pelo menos 1 artigo útil.	2

Tabela 7.1: Critérios de classificação das pesquisas

Como não é fidedigno se os utilizadores responderam de facto "Não útil", às pesquisas que efetuaram, a equipa de UXA re-avaliou cada uma das questões, analisando os artigos devolvidos de modo a garantir que, de facto, aquela pesquisa não retornou os artigos que deveria. Realizaram-se as mesmas pesquisas, no novo *widget*, de modo a avaliarem-se os resultados obtidos também pelo projeto desenvolvido.

De notar que se considera que a pesquisa melhora, se o nível passar a ser de 0-1, de 1-2, ou de 0-2, porque o primeiro significa que não existiam artigos, mas agora existem; o segundo que foram devolvidos artigos úteis, que não eram apresentados antes, embora existissem; o último, que não existiam artigos, mas agora existem e são apresentados corretamente.

Com os dados obtidos, elaborou-se o gráfico de barras da figura 7.6, que permite tirar algumas conclusões, para a amostra analisada, como: a redução de 20 pesquisas que não devolviam artigos a 0, o aumento das fontes permitir que seja apresentado 1 artigo que não era apresentado antes, e o aumento da eficácia de resposta de 22 para 43, isto é, de 44% para 86%, novamente, apenas para esta amostra.

Uma consideração a ter é, que apesar das fontes de artigos passarem a ser três, em vez de apenas uma, para este conjunto de pesquisas, não houve uma melhoria significativa. Outro pormenor deve-se ao facto de não existirem respostas em que o novo *chatbot* contém artigos corretos, mas não os devolve (a amarelo). Neste estudo realizado isso não aconteceu, porque as pesquisas utilizadas, embora aleatórias, não continham erros nas palavras e podem considerar-se como pesquisas "corretas".

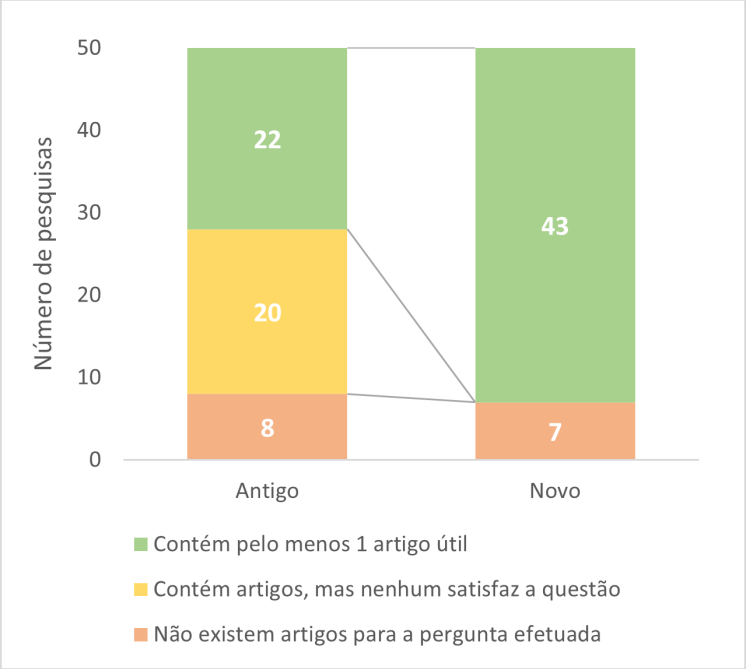


Figura 7.6: Análise gráfica das estatísticas obtidas para a antiga pesquisa *versus* a nova

Depois de se ter um conjunto de dados, à partida fiável (construído pela equipa de *User Experience and Assurance* (UXA)), optou-se por realizar o teste estatístico *Paired T-Test*, uma vez que as pesquisas estão emparelhadas, ou seja, a mesma pesquisa é feita em ambas as tecnologias, e o que se pretende testar é qual é mais eficaz. Assumindo um nível de confiança de 95% ou um nível de significância de 5%, e para os dados apresentados anteriormente, obteve-se o resultado seguinte:

```
Paired t-test
data: novoChatbot and antigoChatbot
t = 5.7553, df = 49, p-value = 2.782e-07
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
0.3118244       Inf
sample estimates:
mean of the differences
0.44
```

Como o valor do *p-value* $= 2.782e-07$ e é inferior ao nível de significância de $\alpha = 0.05$, conclui-se que existe uma probabilidade estatística alta de que a pesquisa no novo *widget* é mais eficaz do que na janela de pesquisa que existia e portanto é rejeitada a hipótese nula.

Aceita-se por isso a hipótese alternativa, verificando-se o cumprimento da hipótese 3 - o aumento da eficácia das respostas obtidas pelo novo *widget*.

Capítulo 8

Conclusão

O último capítulo da dissertação resume quais os objetivos cumpridos, em relação aos iniciais definidos em 1 e mais tarde refletidos nos casos de uso em 4; algumas considerações sobre o trabalho desenvolvido; quais as restrições/limitações que ocorreram ao longo da implementação do projeto; e, para terminar, o trabalho futuro necessário, do ponto de vista de melhoria, a ser realizado.

8.1 Concretização dos Objetivos

Quanto aos objetivos iniciais propostos em 1.3, classificam-se os seguintes como cumpridos:

- exploração da solução atual e ferramentas de trabalho;
- desenvolvimento de um novo projeto com uma ajuda centralizada;
- ligação/integração a/com outros projetos existentes (Comunidade, LiveAgent/HelpDesk, ElasticSearch, Botpress, Admin e Services);
- desenvolvimento de um fluxo/mecanismo de *Chatbot* ou de conversa;
- aproveitamento das funcionalidades de *Natural Language Processing* (NLP) para treino de um modelo;
- expansão das fontes de pesquisa;
- disponibilização de consulta de notificações e *tickets*;
- cumprimento dos objetivos secundários (com exceção da filtragem de *tickets*).

Em suma, observando os objetivos aqui apresentados, de modo geral, foram todos cumpridos e considera-se que a solução apresentada é viável para ser utilizada.

O estudo realizado inicialmente permitiu obter conhecimentos sobre o modo de funcionamento do antigo *Chatgoi* (janela de pesquisa que existia) e obter os conhecimentos necessários para a resolução do projeto.

Após a concretização do estudo e análise do projeto a desenvolver, implementou-se o necessário para que o Widget-Help pudesse interagir com outros projetos, agrupando todos os mecanismos de ajuda num só *widget*.

O desenvolvimento do fluxo de conversa foi desenvolvido, seguindo atentamente a documentação do Botpress e as boas práticas presentes na mesma. Foram realizadas diversas alterações consoante os requisitos e finalizado de forma a que seja facilmente mantido.

Finalmente, a utilização dos mecanismos de NLP do *chatbot* foi realizada com recurso a dados internos, ajustando as frases utilizadas ao registo de dúvidas dos utilizadores e a forma como estes pesquisam.

A garantia de cumprimento dos objetivos, advém dos testes realizados no capítulo 7 e da extensiva experimentação, pela equipa de *User Experience and Assurance* (UXA), ao longo de todo o processo de desenvolvimento.

8.2 Considerações e Aprendizagens

O trabalho desenvolvido durante o tempo da dissertação, ainda que a tempo parcial, permitiu adquirir diversos conhecimentos sobre as diversas tecnologias utilizadas, nomeadamente: Python, NgRx (estado reativo para aplicações de Angular), Integrações com API's, pré processamento textual, indexação de bases de dados, tecnologias *Natural Language Understanding* (NLU)/NLP, ambientes de desenvolvimento (Jenkins, GitLab, etc), Jira (gestão de projetos ágeis) e configuração de servidores.

Potenciou ainda a inserção num ambiente de equipa, como parte de uma organização, abrindo novos horizontes sobre possíveis interesses pessoais em projetos futuros e diferentes perspectivas de trabalho, bem como relacionamentos inter-pessoais, não só com pessoas do mesmo departamento, mas com todos os outros.

Como decerto acontece em outros ambientes empresariais, consideram-se algumas imperfeições, que provocam algumas dificuldades no desenvolvimento do projeto, designadamente: o trabalho simultâneo noutras áreas prioritárias de negócio da E-goi (como o processo de criação de conta); alguma indefinição dos objetivos prioritários em cada etapa do processo; a inexistência/falta de documentação de suporte aos projetos, o que implicou a necessidade de investigação, levando a um maior intervalo de tempo de realização das tarefas previstas; o enorme número de dependências de outras equipas e recursos (servidores e projetos), pelo que nem sempre houve disponibilidade para uma resposta imediata; e, em suma, alguma falta de apoio em áreas mais desconhecidas do contexto da plataforma.

8.3 Restrições e Limitações

A integração com o LiveAgent (UC-6) teve que ser feita através de um *iframe*, porque após contacto com o suporte deste serviço, verificou-se que a *Application Programming Interface* (API) (implementação prevista), não permitia uma integração direta e complexa como esta (realização e transferência de chamadas *live-chat*, anexo de ficheiros, *webcam* e microfone, etc). Nesse sentido invés da integração ser feita por *HyperText Transfer Protocol* (HTTP) *Representational State Transfer* (REST), foi realizada através da API de JavaScript, que se traduz essencialmente num *iframe* colocado no Widget-Help.

A tecnologia do Botpress, imposta pela empresa, é ainda uma tecnologia em desenvolvimento, que tem variados problemas de utilização: quer ao nível da documentação que é incompleta e pouco clara; quer ao nível das limitações do serviço gratuito (a tradução das mensagens para outras línguas é paga, tendo sido necessário implementar do lado do projeto Widget-Help); quer ao nível da API que é consideravelmente limitada, apenas podendo ser feito um pedido de uma única forma; a existência de novos erros, quando são realizadas atualizações à versão de produção para acesso a novas funcionalidades e garantia de segurança.

Como foi referido ao longo da dissertação, o contexto/áreas da E-goi é muito grande e, apesar de haver uma enorme quantidade de artigos, existe ainda uma quantidade significativa de "*grey areas*", isto é, temas que ainda não foram cobertos em artigos, não sendo possível devolver sempre ao utilizador a resposta que precisa. Com a implementação do UC-23 (consulta de pesquisas), espera-se que a falta de alguns artigos seja coberta ao longo do tempo.

8.4 Trabalho Futuro

Relativamente ao trabalho realizado, todos os casos de uso foram realizados, exceto o UC-9 e UC-21, o filtro de pesquisa de tickets e a limitação de mensagens nas conversas, respetivamente. Quanto ao primeiro, a barra de pesquisa foi criada, mas apenas permite efetuar uma pesquisa textual, isto é, filtrar os tickets pelo título e pré-visualização, apesar do pretendido ser uma pesquisa por estado, filtrando todos os que estão abertos ou todos que foram respondidos, etc. Já o segundo, este foi um caso de uso autoproposto, por um motivo de eficiência e de carregamento da página. Se o utilizador nunca limpar a *cache* do *browser*, a conversa ficará para sempre carregada inteiramente, o que levará a um tempo de carregamento da página cada vez maior. Apesar de não se considerarem estes casos de uso fundamentais, estima-se que a resolução dos mesmos não seja muito complexa, pelo que, poderá ser resolvido num curto espaço de tempo.

No projeto da Comunidade foram detetados alguns problemas, que já existiam e devem ser resolvidos, como por exemplo colapsar os sub-comentários de um comentário de um tópico (se um comentário tiver vários, são mostrado todos, escondendo outros tópicos), o *refactor* de todo o código que é desnecessário/inutilizado e alterações de estilos para adequação ao projeto do *Widget-Help*.

Quanto ao projeto do ElasticSearch, existe uma alternativa à forma como são submetidos os artigos. Atualmente, cada artigo das 3 fontes existentes (Blog, Youtube e Helpdesk) é submetido um a um, o que leva algum tempo. Como esta é uma tarefa diária e que não tem impacto para os utilizadores não há um problema visível. Contudo, poderá reduzir-se o tempo de *upload* deste artigos recorrendo a operações em massa/*bulk*[1]. Desta forma, poderão ser submetidos vários artigos apenas com um pedido o que reduzirá substancialmente o tempo que leva a colocar todos na base de dados.

Já para o fluxo de conversa, seria também interessante, em vez de serem sugeridos artigos, mostrado dentro da conversa o conteúdo dos mesmos. Esta informação já está presente no código, só não é mostrada, porque não foi um objetivo deste projeto. Contudo, para que isto seja fiável, é necessário que seja mostrado o "artigo certo" e não os 5 artigos que se enquadram na pergunta do utilizador.

[1] https://www.elastic.co/guide/en/elasticsearch/reference/6.8/docs-bulk.html

Bibliografia

[1] Leticia Polanco-Diges e Felipe Debasa. «The use of Digital Marketing Strategies in the Sharing Economy: A literature Review». Em: *Journal of Spatial and Organizational Dynamics* 8.3 (2020), pp. 217–229.

[2] Mrs Vaibhava Desai. «Digital Marketing: A Review». Em: *International Journal of Trend in Scientific Research and Development* (mar. de 2019), pp. 196–200.

[3] Bogdan-Alexandru Andrei et al. «A study on using waterfall and agile methods in software project management». Em: *Journal Of Information Systems & Operations Management* (2019), pp. 125–135.

[4] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[5] Roel J Wieringa. *Introduction to design science methodology*. 2019.

[6] Botpress. *Glossary*. url: `https://botpress.com/docs/main/glossary` (acedido em 09/01/2021).

[7] Alex Liverant et al. *System and method for enhanced interaction between an iframe or a web page and an embedded iframe from a different domain*. US Patent App. 14/200,970. Jul. de 2014.

[8] Botpress. *NLU*. url: `https://botpress.com/docs/build/nlu#how-it-works` (acedido em 09/01/2021).

[9] Facebook. *Quick Replies*. url: `https://developers.facebook.com/docs/messenger-platform/send-messages/quick-replies` (acedido em 09/01/2021).

[10] Gartner. *Information Technology - Gartner Glossary*. url: `https://www.gartner.com/en/information-technology/glossary/sdk-software-development-kit` (acedido em 09/01/2021).

[11] Facebook. *Webhooks*. url: `https://developers.facebook.com/docs/graph-api/webhooks/` (acedido em 09/01/2021).

[12] Facebook. *Webview*. url: `https://developers.facebook.com/docs/messenger-platform/webview` (acedido em 09/01/2021).

[13] Aarsh Trivedi, Vatsal Gor e Zalak Thakkar. «Chatbot generation and integration: A review». Em: *International Journal of Advance Research, Ideas and Innovations in Technology* 5.2 (2019), pp. 1308–1311.

[14] «The 62 Billion Customer Service Scared Away». Em: *New Voice Media* (2016). url: `https://www.vonage.com/resources/articles/the-62-billion-customer-service-scared-away-infographic/` (acedido em 09/01/2021).

[15] Sam Smith. «Chatbots, a Game Changer for Banking and Healthcare, Saving 8 billion Annually by 2022». Em: *Juniper Research* ().

[16] Abbas Saliimi Lokman e Mohamed Ariff Ameedeen. «Modern chatbot systems: A technical review». Em: *Proceedings of the future technologies conference*. Springer. 2018, pp. 1012–1023.

[17] Ilya Sutskever, Oriol Vinyals e Quoc V Le. «Sequence to sequence learning with neural networks». Em: *arXiv preprint arXiv:1409.3215* (2014).

[18] Kishore Papineni et al. «Bleu: a method for automatic evaluation of machine translation». Em: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.
[19] Facebook. *Built-in NLP*. Jun. de 2020.
[20] Facebook. *Message Templates*. url: `https://developers.facebook.com/docs/messenger-platform/send-messages/templates` (acedido em 09/01/2021).
[21] Botpress. *Tutorials*. url: `https://botpress.com/docs/developers/tutorials` (acedido em 09/01/2021).
[22] Botpress. *Actions & Hooks*. url: `https://botpress.com/docs/main/code` (acedido em 09/01/2021).
[23] Botpress. *Quick Start*. url: `https://botpress.com/docs/quickstart` (acedido em 09/01/2021).
[24] Tony Woodall. «Conceptualising "value for the customer": an attributional, structural and dispositional analysis». Em: *Academy of marketing science review* 12.1 (2003), pp. 1–42.
[25] Nick Rich e Matthias Holweg. «Value analysis». Em: *Value engineering* (2000).
[26] Peter A Koen et al. «Fuzzy front end: effective methods, tools, and techniques». Em: *The PDMA toolbook* 1 (2002), pp. 5–35.
[27] Maruti Techlabs. *Can Chatbots Help Reduce Customer Service Costs by 30 percent?* Abr. de 2017.
[28] Andy Haas. *Global Contact Center Survey*. 2017.
[29] IBM. *How chatbots can help reduce customer service costs by 30 percent*. Mai. de 2019.
[30] Thomas L. Saaty. «The Modern Science of Multicriteria Decision Making and Its Practical Applications: The AHP/ANP Approach». Em: *Operations Research* (2013), pp. 1101–1118.
[31] Coventry University of Warwick. *Quality Function Deployment*. Warwick Manufacturing Group, 2019.
[32] Alexander Osterwalder e Yves Pigneur. «Modeling value propositions in e-Business». Em: *Proceedings of the 5th international conference on Electronic commerce*. 2003, pp. 429–436.
[33] Alexander Osterwalder et al. *Value proposition design: How to create products and services customers want*. John Wiley e Sons, 2014.
[34] Roger S Pressman. *Software Engineering A Practitioner's App: Seventh Edition*. Raghothaman Srinivasan, 2010.
[35] Andrew Butterfield, Gerard Ekembe Ngondi e Anne Kerr. *A dictionary of computer science*. Oxford University Press, 2016.
[36] Robert B. Grady. «Successfully applying software metrics». Em: *Computer* 27.9 (1994), pp. 18–25.
[37] Eeles Peter. *Capturing Architectural Requirements*. Nov. de 2005. url: `https://www.ibm.com/developerworks/rational/library/4706.html#N100A7` (acedido em 10/02/2021).
[38] Simon Brown. «The C4 Model for Software Architecture». Em: (ago. de 2018). url: `https://www.infoq.com/br/articles/C4-architecture-model/` (acedido em 15/04/2021).
[39] Philippe Kruchten. «Architectural Blueprints — The "4+1" View Model of Software Architecture». Em: (nov. de 1995). url: `https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf` (acedido em 15/04/2021).

[40] Matthew Foemmel Martin Fowler Dave Rice. *Patterns Of Enterprise Application Architecture*. Addison-Wesley, 2002.
[41] IBM Corporation. *Deployment diagrams*. url: https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=diagrams-deployment (acedido em 12/04/2021).
[42] IBM Corporation. *Component diagrams*. url: https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=diagrams-component (acedido em 12/04/2021).
[43] Max Rehkopf. *What is a kanban board?* url: https://www.atlassian.com/agile/kanban/boards (acedido em 14/02/2021).
[44] Jason Brownlee. «How to Clean Text for Machine Learning with Python». Em: (ago. de 2019). url: https://machinelearningmastery.com/clean-text-machine-learning-python/ (acedido em 25/05/2021).
[45] Diana Lee. «How we analyzed 200K messages to design a chatbot». Em: (jan. de 2019). url: https://chatbotsmagazine.com/how-we-analyzed-200k-messages-to-design-a-chatbot-264a13724752 (acedido em 25/05/2021).
[46] Kavita Ganesan. «All you need to know about text preprocessing for NLP and Machine Learning». Em: (abr. de 2019). url: https://www.kdnuggets.com/2019/04/text-preprocessing-nlp-machine-learning.html (acedido em 26/05/2021).
[47] Oxford Dictionary. *hypothesis*. url: https://www.oxfordlearnersdictionaries.com/definition/english/hypothesis (acedido em 20/02/2021).
[48] Paula Escudeiro e José Bidarra. «Quantitative evaluation framework (QEF)». Em: *Conselho Editorial/Consejo Editorial* 16 (jan. de 2008), p. 16.
[49] Loucopoulos P Heidari F. *Quality evaluation framework (QEF): Modeling and evaluating quality of business processes*. Int J Account Inf Syst, 2013.
[50] Peter Sestoft. «Systematic software testing». Em: *Version* 2 (2008), pp. 2008–02.
[51] Wayne W. LaMorte. *T-test for Two Dependent Samples (Paired or Matched Design)*. url: https://sphweb.bumc.bu.edu/otlt/MPH-Modules/PH717-QuantCore/PH717-Module7-T-tests/Module7-ttests8.html (acedido em 15/06/2021).

Apêndice A

Folha Excel - Modelo AHP

Pode ser consultado o ficheiro [AHP.xlsx] para visualização dos cálculos realizados.

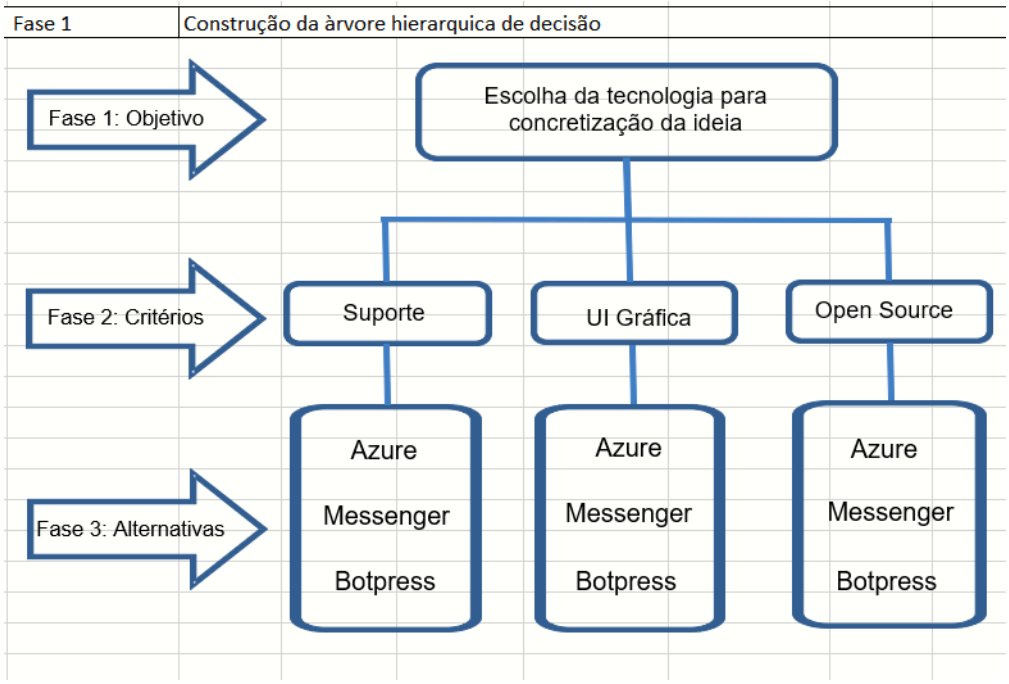


Figura A.1: Modelo AHP - Fase 1

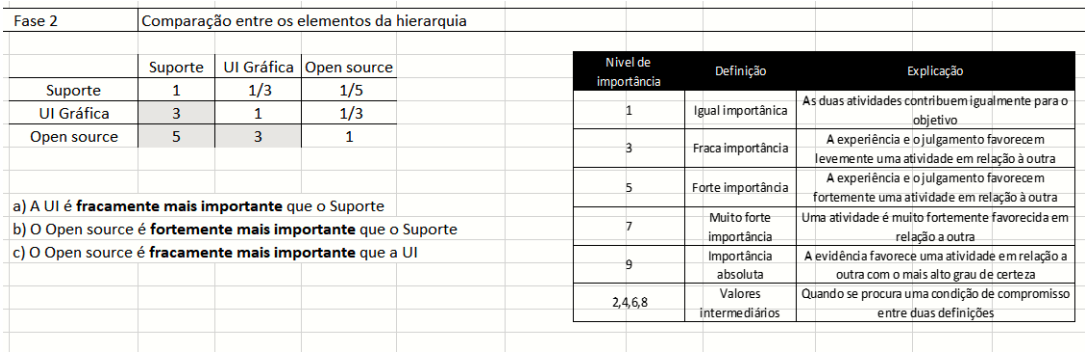


Figura A.2: Modelo AHP - Fase 2

Fase 3	Prioridade relativa de cada critério		
	Suporte	UI Gráfica	Open source
Suporte	1	1/3	1/5
UI Gráfica	3	1	1/3
Open source	5	3	1
Soma/Total	9	4,3333333	1,53333333

Normalizada	Suporte	UI Gráfica	Open source	Prioridade Relativa ou Vetor Próprio
Suporte	0,111111	0,076923077	0,13043478	0,106156
UI Gráfica	0,333333	0,230769231	0,2173913	0,260498
Open source	0,555556	0,692307692	0,65217391	0,633346

Figura A.3: Modelo AHP - Fase 3

Fase 4	Avaliar a consistência das prioridades relativas

[A*x=hmax*x]
A: Matriz de comparação dos critérios
x: Vetor Próprio
hmax: Valor Próprio

IC=(hmax-n)/(n-1)
IC: Indice de Consistência
n: Número de Critérios (N=3)

RC=IC/IR
RC: Razão de Consistência
IR: Indice Aleatório

TABELA - Valores de IR para matrizes quadradas de ordem n

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

A	Suporte	UI Gráfica	Open source		X		hmax*x
Suporte	1	1/3	1/5		0,106156324		0,31965812
UI Gráfica	3	1	1/3	*	0,260497956	<=	0,790082167
Open source	5	3	1		0,63334572		1,945621206

n =	3	IR =	0,58
hmax =	3,03871468		
IC =	0,01935734		
RC =	0,03337472	< 0,1	

Figura A.4: Modelo AHP - Fase 4

Fase 5	Construção da matriz de comparação paritária para cada critério, considerando cada uma das alternativas selecionadas

Suporte	Azure	Messenger	Botpress
Azure	1	1/7	1/2
Messenger	7	1	4
Botpress	2	1/4	1
Soma/Total	10	1,3928571	5,5

Normalizada	Azure	Messenger	Botpress	VP
Azure	0,1	0,102564103	0,09090909	0,097824
Messenger	0,7	0,717948718	0,72727273	0,715074
Botpress	0,2	0,179487179	0,18181818	0,187102

a) O Messenger tem um suporte que é **fortemente favorecido** em relação ao Azure
b) O Botpress tem um suporte que é **igualmente/fracamente favorecido** em relação ao Azure
c) O Messenger tem um suporte que é **fracamente/fortemente favorecido** em relação ao Botpress

UI	Azure	Messenger	Botpress
Azure	1	5	1/5
Messenger	1/5	1	1/7
Botpress	5	7	1
Soma/Total	6,2	13	1,34285714

Normalizada	Azure	Messenger	Botpress	VP
Azure	0,16129	0,384615385	0,14893617	0,231614
Messenger	0,032258	0,076923077	0,10638298	0,071855
Botpress	0,806452	0,538461538	0,74468085	0,696531

a) O Azure tem uma interface gráfica que é **fortemente favorecida** em relação ao Messenger
b) O Botpress tem uma interface gráfica que é **fortemente favorecida** em relação ao Azure
b) O Botpress tem uma interface gráfica que é **muito fortemente favorecida** em relação ao Messenger

OS	Azure	Messenger	Botpress
Azure	1	1	1/5
Messenger	1	1	1/5
Botpress	5	5	1
Soma/Total	7	7	1,4

Normalizada	Azure	Messenger	Botpress	VP
Azure	0,142857	0,142857143	0,14285714	0,142857
Messenger	0,142857	0,142857143	0,14285714	0,142857
Botpress	0,714286	0,714285714	0,71428571	0,714286

a) O Messenger não é Open Source, o que tem uma **igual importância** em relação ao Azure
b) O Botpress é Open Source, o que é **fortemente favorecido** em relação ao Azure
b) O Botpress é Open Source, o que é **fortemente favorecido** em relação ao Messenger

Figura A.5: Modelo AHP - Fase 5

Apêndice B

Folha Excel - Modelo QEF

Pode ser consultado o ficheiro [QEF.xlsx] para visualização dos diagramas abaixo.

Fator	Desenho	Wfk - Cumprimento (%)		
Requisito	**Avaliação métrica**	**0**	**50**	**100**
UD01	A aplicação segue as boas práticas de Angular Material funcionando em diferentes resoluções de ecrã ou ao encolher a janela do browser	Apenas funciona em fullscreen	Apenas funciona em diferentes resoluções ou dimensões	Funciona em qualquer circunstância (resolução e dimensão)
UD02	A aplicação é fácil de utilizar, intuitiva, apresentando tudo que foi explicitado no design fornecido pela empresa	Não implementado	Implementado Parcialmente	Totalmente Implementado
UD03	Após um período de inutilização da aplicação, o utilizador deve ser capaz de realizar qualquer tarefa sem necessitar ajuda na interação	Utilizável mas precisa de ajuda	Utilizável apenas depois de aprendida	Facilmente utilizável, após períodos de inutilização

Fator	Prevenção Falhas	Wfk - Cumprimento (%)		
Requisito	**Avaliação métrica**	**0**	**50**	**100**
UDPF	A aplicação cobre grande parte dos erros nas funcionalidades mais importantes e apenas ocorrem erros em casos excecionais	Não cobre erros nenhuns	As grandes funcionalidades estão operacionais	É possível interagir com a aplicação realizando todas as tarefas

Figura B.1: Modelo QEF - Métricas da dimensão Usabilidade

Fator	Suportabilidade	Wfk - Cumprimento (%)		
Requisito	**Avaliação métrica**	**0**	**50**	**100**
AS01	A aplicação deve ser funcional em múltiplos browsers (Chrome - mais utilizado, Firefox, Internet Explorer,...)	1+ browser	2+ browsers	3+ browsers
AS02	A aplicação deve estar disponível nos quatro idiomas utilizados pela E-goi (Português, Português do Brasil, Inglês e Espanhol)	1+ linguagem	2+ linguagens (PT e EN)	4 linguagens
AS03	A aplicação deve ser testada de diferentes maneiras, como por exemplo através de: testes unitários, testes de integração ou testes manuais	0 ou nenhum	1+ tipos de testes	2+ tipos de testes

Fator	Manutenibilidade	Wfk - Cumprimento (%)		
Requisito	**Avaliação métrica**	**0**	**50**	**100**
AM01	O projeto deve ser documentado a nível de código, a nível de repositório, a nível de integrações com outros projetos e a nível arquitetural	Sem documentação	Documentação Parcial	Documentação Total
AM02	Sendo o projeto do Botpress atualizado constantemente, deve ser facilmente alterável tendo um fluxo percetível e entendível por *non-developers*	Só entendido pelo *developer*	Apenas entendível por *developers*	Entendível por todos

Figura B.2: Modelo QEF - Métricas da dimensão Adaptabilidade

Fator	Integração de Projetos	**Wfk - Cumprimento (%)**		
Requisito	**Avaliação métrica**	**0**	**50**	**100**
FIP01	O utilizador deve poder ter uma visibilidade total da Comunidade dentro do Widget-Help, com possibilidade de criação de tópicos, comentários e votos	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FIP02	O utilizador deve poder criar tickets pelo fluxo de conversa com o chatbot, com especificação do problema e com possibilidade anexo de imagem	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FIP03	O utilizador deve, após criado um ticket, poder continuar a conversar com o suporte através de uma janela específica para o efeito, visualizando toda a informação relativa aquele ticket	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FIP04	O utilizador pode contactar o suporte do SAC, através de um chat, trocando mensagens, com a possibilidade de anexo de ficheiros e sistema de avaliação final	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FIP05	O utilizador deve conseguir trocar mensagens com um mecanismo de Conversação Autonóma, sem necessidade de qualquer resposta humana	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FIP06	O utilizador deve conseguir aceder à pagina da E-goi e visualizar/interagir com o Widget-Help	Não cumprida	Parcialmente cumprida	Totalmente cumprida

Fator	Mecanismo Conversa	**Wfk - Cumprimento (%)**		
Requisito	**Avaliação métrica**	**0**	**50**	**100**
FMC01	A pesquisa na base de conhecimento atual deve ser melhorada, para ir de encontro às necessidades do utilizador	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FMC02	O utilizador deve poder consultar o histórico de conversas com o chatbot	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FMC03	O utilizador deve poder continuar uma conversa inacabada, saindo da página e voltando ou saindo da conta e feito novamente o login ou minimizando o Widget e voltando a abrir	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FMC04	Devem ser identificadas as frases do utilizador, para que o problema seja identificado e respondido da melhor forma, através de mecanismos de AI	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FMC05	O utilizador deve poder enviar mensagens não com conteúdo textual, mas também sob a forma de imagens	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FMC06	O utilizador deve poder enviar emojis, de modo a tornar a conversa mais amigável	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FMC07	As frases incompreendidades pelo chatbot devem ser guardadas numa base de dados para poderem ser consultadas mais tarde	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FMC08	O sistema de alarmística já existente na E-goi deve ser utilizado no projeto do Widget-Help para caso o mecanismo de conversa não esteja disponível gerar um alerta	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FMC09	O utilizador deve apenas ver um número de mensagens limitado, para não existir sobrecarga da página (para ver mais terá que efetuar o *scroll* para cima para carregar mensagens antigas)	Não cumprida	Parcialmente cumprida	Totalmente cumprida

Fator	Consulta Informação	**Wfk - Cumprimento (%)**		
Requisito	**Avaliação métrica**	**0**	**50**	**100**
FCI01	O utilizador deve poder consultar os tickets criados, bem como visualizar o estado dos mesmos	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FCI02	O utilizador deve poder consultar as notificação recebidas e poder apagá-las	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FCI03	O utilizador deve poder filtrar os tickets de acordo com o estado em que se encontram	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FCI04	O utilizador deve poder ver a quantidade de notificações que ainda não foram abertas	Não cumprida	Parcialmente cumprida	Totalmente cumprida
FCI05	O utilizador deve poder ver o total de notificações que ainda não foram vistas + o número de tickets que ainda estão por resolver (bem como os valores individuais associados a cada um)	Não cumprida	Parcialmente cumprida	Totalmente cumprida

Fator	Outro(s)	**Wfk - Cumprimento (%)**		
Requisito	**Avaliação métrica**	**0**	**50**	**100**
FO01	Um utilizador não autenticado deve poder falar com o chatbot (embora que limitado apenas a esclarecimento de dúvidas)	Não cumprida	Parcialmente cumprida	Totalmente cumprida

Figura B.3: Modelo QEF - Métricas da dimensão Funcionalidade

q	D	Qi	Dim	Q_j	W_{ij} [0,1]	Fator	rw_{jk} [0, 10]	Requisito	wf_k [0, 100]
96%	0.15	84.98	Funcionalidade	100	0.30	Integração de Projetos	8	FIP01 - Integração com o projeto da Comunidade	100
							10	FIP02 - Integração com o LiveAgent/Botpress (Criação de Tickets)	100
							10	FIP03 - Integração com o LiveAgent (Continuação do fluxo do Ticket)	100
							10	FIP04 - Integração com o LiveAgent (Live-chat)	100
							10	FIP05 - Integração com o Botpress (Fluxo de conversa/Troca de mensagens)	100
							10	FIP06 - Integração do Widget com os projetos da E-goi como Web Component	100
							10	FIP07 - Integração do projeto Admin com o Botpress (Consulta de pesquisas)	100
				79.17	0.39	Mecanismo Conversa	10	FMC01 - Melhoria da pesquisa de artigos existente	100
							6	FMC02 - Consulta do histórico de conversas	100
							8	FMC03 - Persistência do estado de uma conversa	100
							10	FMC04 - Identificação NLP no Botpress	50
							6	FMC05 - Implementar a possibilidade de anexar ficheiros	100
							4	FMC06 - Implementar o envio de emojis	100
							8	FMC07 - Implementar persistência das pesquisas efetuadas	100
							10	FMC08 - Implementar sistema de alarmística	100
							10	FMC09 - Limitar o número de mensagens visíveis pelo utilizador	0
				87.5	0.22	Consulta Informação	8	FCI01 - Listar/Consultar tickets	100
							8	FCI02 - Listar/Consultar notificações	100
							4	FCI03 - Filtro de pesquisa de Tickets	0
							6	FCI04 - Distinção entre notificações lidas/por ler	100
							6	FCI05 - Número de notificações/tickets não abertos	100
				100	0.05	Outro(s)	8	FO01 - Disponibilizar o widget para utilizadores não autenticados	100
							8	FO02 - Implementar traduções no Widget-Help	100
		100	Adaptabilidade	100	0.60	Suportabilidade	4	AS01 - A aplicação funciona em múltiplos browsers	100
							8	AS02 - A aplicação está disponível em vários idiomas	100
							8	AS03 - A aplicação deve ser testada em vários níveis	100
				100	0.40	Manutenibilidade	8	AM01 - O projeto deve ser bem documentado para futuras referências	100
							10	AM02 - O projeto do Botpress deve ser fácil de manter/alterar	100
		100	Usabilidade	100	0.75	Desenho	8	UD01 - A aplicação deve ser responsiva	100
							10	UD02 - A aplicação deve ser intuitiva	100
							6	UD03 - A aplicação deve ser fácil de memorizar	100
				100	0.25	Prevenção Falhas	8	UPF01 - A aplicação deve ter uma baixa taxa de erros	100

Figura B.4: Modelo QEF com as dimensões, fatores e requisitos (fase 2)

Apêndice C

Pesquisas Realizadas

Tabela de todas as pesquisas realizadas para o teste de hipóteses 3.

Pesquisa	**r(Antigo)**	**r(Novo)**
alterar email da conta	1	2
exportar newsletters para fora do e-goi	0	0
automatizar e-mail e sms	0	2
quero criar uma campanha de email	2	2
impossibilidade de criar mais listas	2	2
nao consigo fazer o pagamento	0	0
bom dia! quero apagar uma lista de contactos. como fazer?	2	2
aumentar o numero de sms	0	0
api e extra fields	1	2
erro no relatório de envio	0	0
domínio da minha conta e-goi	1	2
como alterar o utilizador principal?	2	2
modelos de email e-goi	2	2
erro na integração formulário subscrição	1	2
contacto com o suporte	2	2
não consigo configurar o cname.	2	2
publicação de campanhas no facebook	1	2
não consigo confirmar novo remetente.	2	2
erro ao instalar modulo prestashop	2	2
tamanho máximo das mensagens de sms	0	0
como aplicar questionários numa campanha	2	2
alterar cartão de credito na renovação do plano	2	2
quero que me desbloqueiem a conta!	1	2
como iniciar a acção do autobot?	2	2
segmentar lista de contatos (segmentos, tags e campos extra)	2	2
personalizar o email de double opt-in	2	2
no google chrome, não consigo editar	2	2
duplicar campanha smart sms	2	2
preciso exportar os e-mails como faço?	1	2
como excluir a minha conta no egoi	1	2
email vai diretamente para spam	2	2
integrar formulário google no e-goi	2	2

track & engage, wordpress e google tag manager	1	2
preciso de saber mais sobre os planos de envios	0	0
importação de base dados	2	2
factura não aparece na minha conta	1	2
web push nao funciona no site	1	2
envio sms não funciona	2	2
erro no registo spf	1	2
ligar o quero ao sage	1	2
não vejo os tickets	1	2
qual o plano a escolher?	0	0
minha conta está bloqueada, preciso de ajuda	2	2
configuração track & engage	2	2
fazer upload de imagem	1	2
envio de newsletter demasiado lento	1	2
onde está o meu client id	1	2
preciso de um contato telefonico vosso urgente	1	2
editar um email agendado	1	2
cname para envios transaccionais	1	2

Tabela C.1: Lista de pesquisas efetuadas em ambos os mecanismos