Application and Service Support Bots

Pedro Avilar Calado Barco
pedro.barco@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2019

Abstract

Using chat support services promises huge cost reductions in human resources and can facilitate communication and even foster customer engagement through the use of natural language. But beyond the promising advantages are also identified challenges: identifying the entity and action in natural language requests, establishing a conversational context, identifying the need to redirect request to a user or system, their articulation with knowledge systems-based and with event managers, are all obstacles that have to be overcome in order to ensure the correct adoption of this communication channel, the purpose of this dissertation. The expected results of this dissertation are to propose, implement and evaluate the use of a chatbot as a communication vehicle to initiate and follow up first-line support requests made by an end user.
Keywords: Chatbot, Application and Service Support, Artificial Intelligence, Cloud-native

1. Introduction

Application and service support is one of the themes regarding user experience in online services. This support includes application troubleshooting and ticket management often delivered by the help desk software. However, this solution may not offer the automation to further improve a business user experience. When users are resolving a complaint or problem, a chatbot may provide the automation needed to refine application and service support by filtering out the issues that must be created and those that can be answered through a trained knowledge base.

Although a chatbot may prove to be an advantageous complement to ticket management, the proper development of such a channel poses challenges. In this regard, building and developing a chatbot has become a non-trivial implementation in which thorough testing and validation are necessary to deliver an application and service support channel that makes issue handling a simpler and easier experience.

This thesis presents a practical case study of a chatbot solution for a company. *Edoclink* was the case study of choice given it is a *Link Consulting* product which has an application and service support software.

2. Background

Interacting with a chatbot is similar to a conversation: each user phrase has an appropriate response. Upon receiving a new message, the agent must be able to identify the user's intention along with important parameters. This behavior is only achievable with Artificial Intelligence, Machine Learning and NLP (Natural Language Processing) tools. Such terms are explained in the following sections.

2.1. Conversation Concepts

Utterances are user sentences. Being an AI program, a chatbot is subject to training or testing utterances. These training phrases are a significant factor in determining a bot's precision. Usually, more than 10 utterances should be used to train each conversational use case.

Intents are intentions. A chatbot program uses phrases, whether via text or audio, to get intentions. Intents can either represent "small talk" or company use cases and can be described by a developer or provided by the chatbot platform. If no intention is identified, the chatbot triggers a special intent called "Fallback Intent" that represents every user request without well-defined intention identified. The more utterances a chatbot uses as training, the better it is to correctly identify which intent the user has.

Entities are parameters. When programming, a function with several parameters could be created and defined by their type and requirement. Just like functions, intents can be described by its utterances and entities. When creating an intent, new entities may be defined as required; if so, that intent should only be triggered whenever all required parameters are present. There are

typically two types of entities: system and custom. System entities are the default entities created and managed by the chatbot platform and its support and extensiveness vary between providers. Custom entities are created by developers and should be used for values that are not covered by system entities, or for objects or concepts that can have several different attributes (a developer composite containing other entities as aliases).

Context gives control over what happens to the conversation. A chatbot conversation resembles a state machine where contexts are machine/conversation states. Each intention may have an input context, which is required to trigger that intent, and an output context, which is created when the intent is successfully triggered and exited.

2.2. Edge Cases

Even if a chatbot is made to grant application and service support a specific and well defined goal, edge cases typically exist. Users need a human operator to handle delicate situations and to get complex problems properly solved, while the bot should be limited to situations that, with automation, can be handled. There are two scenarios in which a human operator may intervene: hijack and handover.

Hijack occurs when a human operator proactively takes over a conversation or at the request of a user. This operation disables further chatbot replies and inserts the user into a conversation room, where it will be handled by a human.

An handover request occurs when a user requests for a conversation hijack. This protocol may be manually triggered when asking for help or automatically when, for example, there are several and consecutive fallback intents, intentions that can't be recognized or handled by chatbots. The handover may notify a pool of operators or a specific one. From then on, operators are entitled to take over control of the conversation hijack (hijack).

Support for handover and hijack may vary, since these protocols are not scoped to chatbot development but rather what messaging platforms they use. Facebook is one of the most popular platforms to support it [1].

2.3. State of the Art

Rather than developers creating chatbots from scratch, current state-of-the-art allows the development of a chatbot to be based on the choice of the integral components, which are already supported by big companies. Evaluating the requirements and choosing the right alternative depends on what purpose the chatbot is built for. The following sections describe the recommended components when building an enterprise-grade conversational bot [3, 13, 2].

Platform	Lex	Dialogflow (api.ai)	Watson	LUIS	Wit.ai	Digital Assistant
Provider	Amazon	Google	IBM	Microsoft	Facebook	Oracle
Pre-built entities	93	60	7	15	22	11
Pre-built intents	15	34	0	20	0	0
Pre-built domains	3	45	0	20	0	3
Channels	4 + others via API	17 including web widget + others via API	only via API	only via Microsoft Bot Framework	Facebook, web widget + others via API	Facebook + others via API
Third party integration	Amazon services	Bots analytics, Google services	IBM services	Microsoft Bot Framework (deployment), Microsoft Azure, Microsoft Services	only via API	Quality Reports, Bots Analytics
Supported Languages	1: English	27	2: English and Japanese	12	80	1: English
Programming Languages	Android, iOS, Java, JavaScript, Python, .Net, Ruby, PHP, Go, and C++	Node.js, Python, Java, Go, Ruby, C#, PHP	Node.js, Java, Python, iOS, Unity	C#, Python, Node.js, Android	Node.js SDK, Python SDK, Ruby SDK	Android, iOS, Javascript
Limits for API calls	**Trial:** 10,000 speech and 5,000 text queries per month; **Paid:** Unlimited;	**Standard:** 100 speech and 180 text queries per minute; **Enterprise:** 300 speech and 600 queries per minute;	**Lite:** 1,000 queries per month;	**Free:** 10,000 queries per month & 5 queries per second; **Paid:** 10 queries per second;	Unlimited	Unlimited
Pricing	**Trial:** 1 year; **Paid:** $0.004 per speech query & $0.00075 per text query;	**Standard:** Free; **Enterprise:** $0.0065-$0.0085 per 15 seconds of audio & $0.002-$0.004 per text query	**Standard:** $0.0025 per API call; **Premium:** price as per the request;	**Basic:** $0.75 per 1,000 transactions.	Free	**Trial:** 30 days with 300$ credit; **Pay as you go:** $0.0053 per request; **Monthly flex:** $0.0035 per request;

Table 1: Comparison between popular chatbot platforms.

2.3.1 Chatbot Platforms

To promote a human-like conversational experience, a chatbot must present a wide range of intents and well-defined entities. A chatbot platform offers a set of prebuilt intents and entities and out-of-the-box integration for messaging channels. Some platforms even offer a selection of domains which can be loaded into a fully working new agent although language support, pricing and limits may vary.

Table 1 compares different chatbot platforms chosen based on popularity, relevance, and references from related work [8, 6, 12].

2.3.2 Serverless Computing

Serverless computing offers the attractive notion of a platform in the cloud where developers simply upload their code, and the platform executes it on their behalf, removing the need for server software and hardware management by the programmer. Applications are split up into individual functions, which can be invoked and scaled separately.

As discussed in the last section, chatbots often use a fulfillment webhook to add additional logic to a conversation. In this context, a function receives an object representing a "conversational turn" (intent matched, parameters, contexts, session identifier) and, upon validating parameters, managing conversation flow and integrating external services, outputs a coherent response.

Big companies are already supporting serverless computing in the form of functions: Google's alternative is called Cloud Functions, Amazon offers AWS Lambda and Microsoft is providing Azure Functions [16]. There is also an open-source project, the "fn" project, which allows serverless computing over most cloud and on-premise infrastructures [4].

2.3.3 Databases

Accessing systems of record from within a dialog is one way of enriching a conversation. For example, a chatbot may use information from the authenticated user to refine entities and responses throughout an interaction. Different chatbot use-cases may need to store different data and, therefore, a database should be chosen based on what information needs to be stored and how often it is accessed.
When providing support services, a user may need to authenticate and an agent will need to store information that identifies a user as such. Moreover, in edge-cases, where operators are intervening, a chatbot must provide a detailed conversation log where users and operators can interact accordingly.

2.3.4 Ticketing Systems

Usually, users encountering issues contact customer service by creating what is called a ticket. A ticket is a data structure similar to a bug report which contains a subject field, where the user summarizes the problem, and a description field. This structure has metadata fields such as id, author, priority, and category, which are used to filter and search for. When a ticket is created, customer service agents are expected to respond to it. However, operators may find that many tickets have a dedicated FAQ or troubleshoot guide which explains it. Moreover, this information is not easily available for everyone and may not be quickly read, since troubleshooting guides give exhausting documentation on various problems and solutions.
Helpdesk ticketing software providers such as Zoho and Zendesk are now offering chatbots to help businesses automate customer interactions. Features may include real-time help, CRM integrations, ticket management and more. However, since an agent is coupled to its helpdesk software, this alternative is not providing nor supporting other ticket management software.

3. Implementation

Edoclink is a document management and process automation tool with three clear audiences.
***Edoclink* Users**, targeting functional use. This audience uses *Edoclink* on a daily basis and may use a chatbot for searching functional FAQs (how to use the product).
Vision Helpdesk Portal Users for management and functional use. This group is a subset of the above and is allowed to create and manage tickets, in case the FAQs do not clarify their doubts.
Customers for marketing. This set represents customers who are interested in acquiring the document management system.

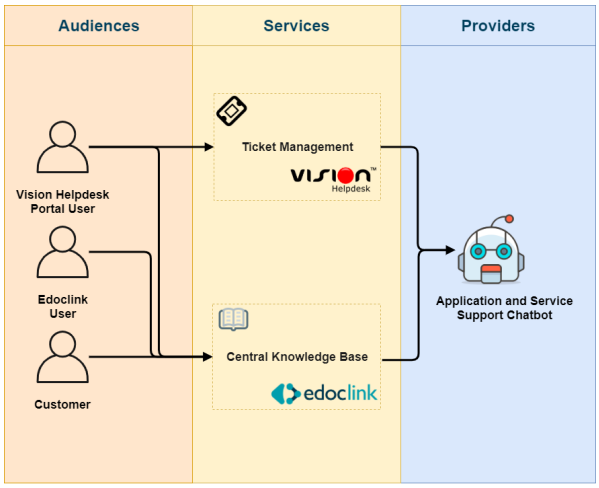

Figure 1: Chatbot application providing content to different audiences

Currently, *Edoclink* is approaching these audiences with different knowledge bases. FAQs are not stored in a single place, therefore, hindering its management. Moreover, this product often experiences duplicate or non-relevant issues which already have a solution; common issues are triggering ticket creation is being served within the Vision Helpdesk portal, without proper validation beforehand.
Edoclink's goal is to deliver an application and service support bot for FAQs and ticket management while supporting handover and hijack features.
Regarding the conversation handover, *Edoclink* agreed on the development of an operator backoffice web application providing the necessary channel for monitoring conversations and communicating with users. As mentioned above, this chatbot is being deployed to three audiences in different websites. However, since users must be connected with the chatbot and Vision Helpdesk is not providing a custom chat widget, there is a need to develop an alternative chatbot interface which could be deployed to each audience website.

3.1. Development Process

Given the greater complexity of a multi-cloud strategy and the timing and scope of the thesis, this project uses a cloud-native design restricted to a single provider, with the solution composed by three applications: the chatbot, a chat widget and a backoffice. The choice process and design method were primarily focused on choosing the platform on which the chatbot was built, since it is the main component of a conversational agent. In a single-provider cloud environment, the selection of other services would depend on this one.

The platform of choice was Google's **Dialogflow**. According to 1, this is the platform that supports

the largest number of prebuilt entities and intents within those that support the Portuguese language. Since Dialogflow is a Google service, the following components were going to be provided by GCP (Google Cloud Platform).

In order for a chatbot to handle more advanced actions, such as validating parsed parameters or requesting 3rd party services such as the ticket management API, chatbot platforms allow you to add logic to one via webhooks. This piece of code is often deployed into a stateless and serverless platform. In Dialogflow, this logic is called a fulfillment webhook and may be provided by an out-of-the-box integration with Google's **Cloud Functions**.

For the widget to be loaded in different scenarios, one would need to version it, distinguishing different audiences. One alternative would be to have 3 instances of web applications behaving similarly. However, saving it to a file in storage and parameterizing for each audience would be a more economical and simple proposition. Therefore, the widget was stored in **Cloud Storage**. This online object storage was used to host a small file meant to be sourced in each environment.

For users to be helped, it would be necessary to access the history of the conversation. Also, in borderline situations such as handover and hijack, conversations would have to be updated in real time. Both live and recorded conversations had to be stored and displayed. Creating a backoffice needed a live notification engine to track chat sessions in almost real time. With Google's **Cloud Datastore** in Firestore mode (also know as *Cloud Firestore*), an application can be updated in near real time using the "realtime listener" mechanism. In this document based database, each file represents a database reference which can be attached to an asynchronous listener; the listener is triggered once upon its creation and whenever data changes.

However, a piece of code to store conversation history and dynamically update chat rooms was lacking. This operation is not within the scope of chatbot's operation and could therefore be asynchronously processed by event management mechanisms. In a pub/sub protocol, messages are published (pub) to a topic, similar to an event bus, and subscribers (sub) trigger functions once a new message is consumed. Since this service is asynchronous, a new event could be triggered without being a bottleneck at the webhook execution level. Using the current Google-supplied **Cloud Pub / Sub** software, the chatbot webhook could be extended to publish new messages.
Finally, the backoffice application would be deployed on Google's **App Engine**: a fully managed serverless application platform. Given the small size of the application, it was thought to use Cloud Storage. However, and preventing future scalability issues, App Engine ensures a more robust solution in an environment where instances can be customized to suit their compute resources and pricing.

In short, the solution is comprised of this Google ecosystem: Dialogflow, Cloud Functions, Cloud Storage, App Engine, Cloud Pub / Sub and Cloud Firestore.
Testing and functional validation were meant to follow three stages:

Internal Testing
: At this stage, an *Edoclink* operator is selected to interact with the chatbot. After a brief presentation on the possible use-cases, this user is being evaluated with metrics such as goal completion time, messages or taps. Expected feedback may be new training phrases and possible tweaks to the conversation flow.

Client Testing
: In client testing, an *Edoclink* customer company would be asked to test the bot. This company would select a few customers to test the application and fulfill a short user experience survey in the bot UI. Bugs may be reported in a dedicated Google Form.

Open Testing
: This stage is meant to be continuous, with no particular target, and metrics should be collected and monitored by Chatbase. Chatbase is a Google-made service focused on virtual agent analytics, providing metrics, session flows, rich filtering, and may help in identifying a problem, filtering and analyzing relevant conversation history.

Although this project consists of three applications (chatbot, widget and backoffice), only the chatbot and its behaviour would be exhaustively tested.

3.2. Development Environment

This project was developed using full stack Javascript technology. Back-end services were implemented in Node.js whereas front-end applications were created using the Vue.js framework (JavaScript, HTML and CSS) [7, 15]. The architectural style present in figure 2 structures this project as a collection of the following services:

3.2.1 Widget

This messaging channel connects users with the chatbot or a human operator. It features

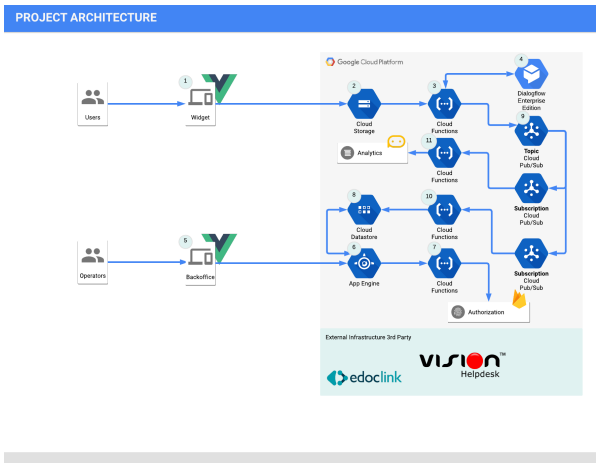

Figure 2: Project Architecture: a Google cloud-native approach

a small deployment script to include in a website and receives a parameter specifying if there should be an authentication screen prior to the chat view, useful when authenticating Vision Helpdesk Portal Users. Authenticated users must login using their Vision Helpdesk username (or email) and password. When a user logs in, a JSON Web Token (JWT) is received, stored and sent in subsequent requests, thus ensuring that requests come from a properly authenticated user [14].

3.2.2 Cloud Storage

The chatbot widget is being hosted in an European public bucket in Cloud Storage. When a file is uploaded to a public bucket, a public URL is generated. This URL is then used in a script instruction to be place in any website to include the widget.

3.2.3 Middleware

Diagflow's webhook has some limitations: responses must occur within 5 seconds, otherwise the request will time out; the response must be less than or equal to 64 KiB in size; fulfillment libraries may have bugs and are not supporting every Dialogflow API functionality. Since users may be importing screenshots when creating a ticket, the dialogflow webhook integration was not a viable option.

This middleware component mimics a Dialogflow fulfillment webhook and connects the chatbot platform with the widget interface. This middleware responds to users queries by communicating with Dialogflow's NLP and executing additional logic to each intent.

"Webhook" logic is divided into slot-filling and fulfillment maps: the first validates and parses parameters when in slot-filling; the latter executes an action when an intent has every required parameter present. When a user triggers an intent that does not use the previously mentioned maps, Dialogflow responses are not subject to additional logic and this component sends the query result directly to the chat interface.

3.2.4 Dialogflow

Dialogflow is responsible for detecting intents, parsing entities, creating conversational context and generating static responses, being the core component of *Edoclink*'s chatbot.

To prevent intents from being wrongly triggered, two fallback intents were added to aid users into choosing what intent to trigger; the "Incident Fallback" is trained with generic user queries regarding tickets and suggests creating a ticket or listing available ones; the "Troubleshoot" intent is trained in a similar manner and helps in browsing the accessible FAQs by listing categories and questions for a category.

3.2.4.1 Knowledge Base

Regarding the knowledge base, *Edoclink* initially provided a set of functional FAQs to be imported to the Dialogflow Knowledge Base Connector. Since this feature is only available in English, the alternative was to map FAQs to intents. To identify the category and target audience of each intent, every FAQ is considered to be an intent beginning with the tag "[FAQ]" and may have two additional tags: the second one is required and indicates the category in which the FAQ should be presented; the last tag can be optionally added to distinguish the target audience. In sum, a FAQ template indicating a category and special target should be "[FAQ] [CATEGORY A] [TARGET X] $FAQ_TITLE". This mechanism is providing a custom and centralized knowledge base connector for every audience.

3.2.4.2 Routing and Templating

Regarding ticket management, intents are divided into a Submit Ticket intent and a Ticket Status intent, allowing users to create a new ticket or list and review one. A routing engine was implemented to seamlessly generate dynamic responses according to the results of integrations with the ticketing system, since Dialogflow is not providing such features. Each dynamically routed intent has an hidden parameter representing a JSON string relating

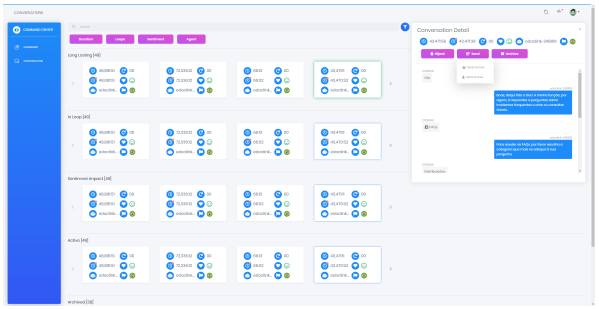

Figure 3: *Edoclink*'s operator backoffice with live conversations

HTTP status codes with an array of appropriate responses. For example, when in ticket creation intent, one possible router may be "{201: [1], 500: [2]}", where the first response block corresponds to a successfully created ticket (code 201) and the second response is used when there is a problem creating a ticket. Additionally, each response block may have template placeholders to be formatted with additional parameters in runtime. Following the previous example, a successfully ticket creation template may be defined as "Ticket ${ticketId} was created successfully" and generated in runtime with the appropriate ticket identifier.

3.2.5 Backoffice

This Vue app provides real-time conversation management for selected *Edoclink* managers. Every chat session represents a room, where metrics are collected along the conversation (average sentiment, duration, conversation loops, etc.). The conversations area lists all rooms, splitting into different ordered filters with one representing an handover pool. An operator may send an active room to the handover pool, hijack a conversation and mark a chat session as resolved. This implementation is based on the conversation lifecycle, later described in this section.

3.2.6 App Engine

This service is responsible for serving the backoffice app and retrieve live conversations updates.

3.2.7 Auth Server

Although this service is not being used and is not part of the project requirements, it was also thought to be part of the overall architecture. This function authenticates backoffice operators and proxies sensitive operations upon authentication. The login procedure and authentication protocol are provided by integrating with Firebase Authentication, supporting email and password based authentication, phone authentication and different OpenID providers such as Google or Microsoft [9].

3.2.8 Firestore

Datastore in Firestore mode is the database being used to store chat history. Conversations are stored in a database collection named "rooms", where each room has a collection of metrics to be displayed on the backoffice chat rooms screen. Each room stores previous messages in the "chat" sub-collection. Performance-wise, a backoffice operator would just query the relevant data in each screen: when listing rooms, metrics are retrieved; when clicking on a conversation, only the room chat would be queried.

3.2.9 Conversations Topic

This component is responsible for handling conversation updates. Upon receiving a new event, a custom JSON payload containing new messages, this topic triggers the store conversations and store metrics functionalities.

3.2.10 Store Conversations

This function stores new conversations in Firestore. Once it is triggered, this subscriber parses the payload, retrieves the session identifier, and attaches new messages to the end of the "chat" collection. Moreover, it collects metrics to be presented in the operator backoffice: document creation and last reply timestamps (the duration of a conversation), average sentiment (provided by Dialogflow's sentiment analysis) and conversation loops (how many times an intent is consecutively triggered).

3.2.11 Store Metrics

This service sends new messages to Chatbase Analytics. Similarly to the previous subscriber, this function uses the session identifier and mutates the conversation object to a chatbase-like request, emulating a user interaction in the mentioned platform.

4. Results

A chatbot must be tested to verify whether it meets the requirements for real use. However, in this context, there are different approaches for testing such a bot, depending on which goals to assess [10].
This chapter presents the metrics used in different assessment environments, justifying the results obtained with possible causes and solutions. The following data was the result of 18 user sessions, with an average of 11 queries per session.

4.1. Content Evaluation

By simulating a human conversation, a chatbot must be able to handle the intentions that are supported in its environment. Such conversations can occur in various ways, but only one goal should be achieved: fulfillment. Content is evaluated by the correct identification of intentions, the effective filtering of each entity, and the coherence between different flows and appropriate responses. Such tests can be done automatically or manually observed through conversation logs.

Chatbase provided table 2, which describe intents statistics and relate conversation states with exit points: points in which the user exits the conversation. For example, it shows that the "Default Welcome Intent" was triggered 33 times and 15.15% of these ended the conversation session; interestingly, the 42.86% exit in "Submit Incident - Subject", may reveal a bug regarding ticket creation, since this intent belongs to one of the intermediate ticket creation steps and should not have such a high exit rate.

Intent	Sessions	Count	Exit %
Default Welcome Intent	18	33	15.15%
Check Incident - Details	7	30	13.33%
Submit Incident	6	20	5%
Incident Fallback	9	13	0%
Troubleshoot - Category	6	11	0%
Troubleshoot	8	11	0%
[FAQ] [Distribuições] Criar Distribuição	2	9	0%
Submit Incident - Description	3	7	42.86%
[FAQ] [Pesquisas] Pesquisar Registo	2	6	0%
Submit Incident - Subject	3	5	0%
[FAQ] [Pesquisas] Pesquisar Distribuição	3	5	0%
[FAQ] [Acessos] Consultar Acessos	3	5	0%
Submit Incident - Confirmation - yes	2	4	25%
Check Incident	2	3	0%
[FAQ] [Pesquisas] Pesquisar Processo	1	2	50%
[FAQ] [Processos] Criar Processo	2	2	0%
Default Goodbye Intent	2	2	50%
[FAQ] [Registos] Registar Documento	1	2	0%
[FAQ] [Distribuições] Despachar Tarefa	1	2	0%
[FAQ] [Configurações] Configurar Página Inicial	1	1	0%
[FAQ] [Configurações] Criar Textos Predefinidos	1	1	0%
[FAQ] [Configurações] Delegar/Aceitar Delegações	1	1	0%
Handover Request	1	1	100%
[FAQ] [Registos] Registar Documento Office	1	1	0%
[FAQ] [Registos] Registar Mails	1	1	100%

Table 2: Intents handled by the conversational agent

By analyzing the previous results and using the Dialogflow conversation history, one may make the following observations:

FAQs are being wrongly triggered

Users are expecting a large knowledge base and, since these intents have similar utterances, a FAQ in the same category is often triggered. This behavior is a common problem in the first stage of training a chatbot and may be due to two different reasons:

- The lack of training phrases in a FAQ may influence the weight to be given in the intent detection process: being an artificial intelligence application, a chatbot needs a significant amount of information to be trained with. Dialogflow recommends a minimum of 10 utterances per intent and intent similarity can be a factor into increasing this number. Adding utterances to *Edoclink* functional FAQs may require a superior product knowledge and, for that reason, should be taken into consideration for future work.
- On the other hand, the absence of an intent that maps a question may also be a factor. In this case, although the solution should be creating a new intent for this question, it may suffer from the previous point: with more similar intents, the training data may not be enough to distinguish them.

After consulting the conversation history, it became apparent that both symptoms applied to the misidentification of intents. The bug reporting forms mentioned before have also shown to be a useful source for identifying such cases.

Ticket related entities need refining

Although the ticket creation ("Submit Incident - Subject") exit percentage was not related with a bug, the problem resided in using the entity present in the "Check Incident" intent. When querying for a ticket, Vision Helpdesk users are using the ticket hash (ABCD-1234, XPTO-5678) instead of the ticket ID (123). After accessing the portal and its ticket organization, we found that it would not be easy to access the ticket ID as it could only been seen in the URL string or in the admin portal used throughout the chatbot development. The problem, however, would be to map a Dialogflow entity that was not a number (system entity) but a set of letters and numbers (custom entity), since regular expression entities were not supported by the platform at the time. However, in September 13, Dialogflow released a new version in which it is possible to define Regexp entities [5].

Custom fallback intents are helping the conversation flow

Testers were often prompting for generic content such as "FAQs" or "Tickets". However, the fallback intents mentioned in the previous chapter and implemented in each use-case revealed to be aiding the user in following one of the possible conversation paths.

For example, if a user was to ask for "tickets", a chatbot would not be able to distinguish the

user intention of creating a ticket or checking the status of an already created incident. In this scenario, a trained fallback intent captures this query and prompts for the possible alternatives.

4.2. Functional Evaluation

In a goal-oriented chatbot, such as an application and support one, functional validation is granted when user goals are fulfilled. For custom integrations, a goal is achieved once a request for fulfilment is completed and successfully executed. In this project, fulfillment testing should cover all ticket management integration points and data sources. Moreover, given the modular cloud architecture of the system, this testing section is also designed to ensure that there is a quick response (performance) between the user's request and the corresponding chatbot action.

The following points were concluded after reviewing bug reporting responses (submitted to the previously mentioned Google Forms platform) and examining the middleware latency graphs provided by GCP:

Dialogflow templates are being poorly rendered

The implemented template engine aims to define and edit all generated responses in Dialogflow, rather than hard-coding each response and not being able to edit one without a new deployment. More accurately, if an action has multiple exit codes, responses are beforehand defined and chosen by the middleware component at run-time.

However, an update to the Dialogflow platform during the testing phase made the mechanism to render symbols while generating the final response [5]. Relying on such software tweaks may not be a viable alternative in an ever changing platform, specially when in a production environment.

Listing tickets may take long

It was thought that listing tickets would help users to quickly query one or more tickets. However, the more tickets a user has, the longer it takes to request the Vision Helpdesk API, even when limiting the number of tickets to be retrieved. This behavior was not detected since the Vision Helpdesk account used for development and internal testing had a limited amount of tickets. Reviewing the middleware logs and crossing the information provided by Dialogflow's conversation history, one may conclude that some high latency calls are, in fact, ticket listing requests. Therefore, it would be necessary to improve the conversation flow to reduce ticket listing operations, when, for example, the user already knows the ticket ID to retrieve. Changing the entity associated with the identifier may also help with the frequency of listing tickets. However, as far as latency is concerned, ≈ 3 seconds in the 99th percentile establishes an acceptable value for a human-like conversation experience.

Cold start latency

In a serverless environment, a cold start latency is the worst-case time that a function execution may take [11]. This latency occurs whenever a container needs to be started due to previous inactivity. Although it is an economic initiative, it may compromise system performance, with a maximum loading latency of 5.78 seconds.

This latency may not be so worrying in a production environment where the chatbot is constantly being queried. Nevertheless, a "container warmer" may be developed and deployed to prevent further latency variations. Since inactivity is a major factor in large loading latencies, a custom warmer service could regularly request the middleware API with a dummy payload, preventing the container service from shutting down. It is important to notice that, even when in constant load, serverless platforms periodically recycle containers, and, in this scenario, it is inevitable to have a cold start.

5. Conclusions

The objective of this work was the application of chatbots to application support and support. The proposal was to use a cloud-native architecture using GCP's services in a decoupled architecture, developing three applications: a ticket management and FAQ's based chatbot, a backoffice in which service support operators could hijack conversations in real time and a web widget acting as a communication channel between users and both agents (bots) and/or (human) operators.

From the three previously mentioned applications, only the chatbot was subject to testing. It stems from the evaluation in the previous chapter that this chatbot is not yet ready to be tested in a different environment nor it should be deployed into such a stage while other applications still need to be properly validated and thoroughly tested.

Despite the strong focus on developing these products, current limitations depend heavily on changes that may happen during the development phase. In particular, this project would benefit, for example, from a (currently supported) regex entity that was not supported by Dialogflow at the time. The platform lacks a router in which developers could match

responses with status codes, therefore reducing additional logic to implement such a feature and replying with a more accurate response. Moreover, dynamic templates cloud be supported in order to match with external parameters often provided by custom integrations.
As regards testing, the different applications were subjected to individual and collective testing throughout their development. However, a functional approach would be needed to consolidate the tests done, especially when testing the backoffice application. Moreover, operationalization of chatbots in a production environment should be monitored, inspecting relevant conversations in order to update and add utterances. In this regard, the backoffice should have an area dedicated to this training process.
In conclusion, the FAQs used in chatbot training were not sufficient to achieve filtering between FAQs and tickets. However, features that facilitate the training process and the enhancement of knowledge base will provide a practical interface for a process that would otherwise be time consuming.
A chatbot can be a viable solution to emulate a conversation in a human-like manner, while being always available, performant and less expensive than a human agent. For now, however, application and service support bots should be designed to help users and not to replace the human role totally, since it is still a strong contender to the most accurate knowledge base.

References

[1] Handover protocol.

[2] Serverless bot framework, Jul 2018.

[3] Building and deploying a chatbot by using dialogflow (overview) — solutions — google cloud, Mar 2019.

[4] Fn project, 2019.

[5] Release notes — dialogflow documentation — google cloud, Oct 2019.

[6] M. Canonico and L. De Russis. A comparison and critique of natural language understanding tools. pages 110–115, 2018.

[7] N. Chhetri. *A Comparative Analysis of Node.js (Server-Side JavaScript)*. PhD thesis, 2016.

[8] O. Davydova. 25 chatbot platforms: A comparative table. *Business Insider*, May 2017.

[9] D. Fett, R. Küsters, and G. Schmitz. The web SSO standard openid connect: In-depth formal security analysis and security guidelines. *CoRR*, abs/1704.08539, 2017.

[10] W. Maroengsit, T. Piyakulpinyo, K. Polyiam, S. Pongnumkul, P. Chaovalit, and T. Theeramunkong. A survey on evaluation methods for chatbots. pages 111–119, 03 2019.

[11] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[12] A. Patil, M. Karuppiah, N. Rao A, and R. Niranchana. Comparative study of cloud platforms to develop a chatbot. *International Journal of Engineering & Technology*, 6:57, 06 2017.

[13] Roalexan. Build an enterprise-grade conversational bot, Jan 2019.

[14] K. Shingala. JSON web token (JWT) based client authentication in message queuing telemetry transport (MQTT). *CoRR*, abs/1903.02895, 2019.

[15] J. Voutilainen. *Evaluation of Front-end JavaScript Frameworks for Master Data Management Application Development*. PhD thesis, 2017.

[16] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.