

Architectural Issues with Chatbots

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

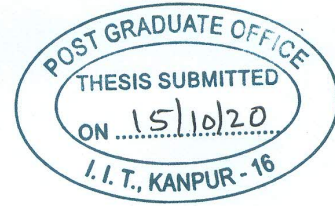
by
Saurabh Srivastava

to the



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR, INDIA

October, 2020



CERTIFICATE

It is certified that the work contained in the thesis entitled "*Architectural Issues with Chatbots*", by *Saurabh Srivastava*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

October, 2020

Handwritten signature of Dr. T.V. Prabhakar in black ink.

Dr T.V. Prabhakar
Professor,

Handwritten signature of Dr. Vinay P. Namboodiri in black ink.

Dr Vinay P. Namboodiri
Associate Professor,

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur,
Kanpur-208016, INDIA.

Declaration

This is to certify that the thesis titled “*Architectural Issues with Chatbots*” has been authored by me. It presents the research conducted by me under the supervision of **Dr T.V. Prabhakar** and **Dr Vinay P. Namboodiri**.

To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgements, in line with established norms and practices.



October, 2020

Saurabh Srivastava
Programme: PhD
Department: Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur 208016

Synopsis

Name of the Student	: Saurabh Srivastava
Roll Number	: 13111164
Degree	: PhD
Department	: Computer Science and Engineering
Thesis Title	: Architectural Issues with Chatbots
Thesis Supervisor	: Dr T.V. Prabhakar
Thesis Co-supervisor	: Dr Vinay P. Namboodiri
Month & Year of Submission	: October, 2020

Conversational interfaces for service portals have become reasonably common. The colloquial term for the software component that provides this interface is *chatbot*. Chatbots can either be built from scratch or, developed using one of many platforms available today. In either case, a chatbot usually functions as a part of a larger, *Containing system*, involving other software components such as Web Servers, Application Servers and Databases. Similar to any other software-intensive system, these systems also have architectural issues. This thesis studies the process of building a chatbot from an architectural perspective, highlighting the various design decisions associated with the process.

The first decision in the design process is to choose a development model. It includes decisions such as picking a development methodology (such as Agile Development) and deciding upon the building tools to use. The chatbot could either be

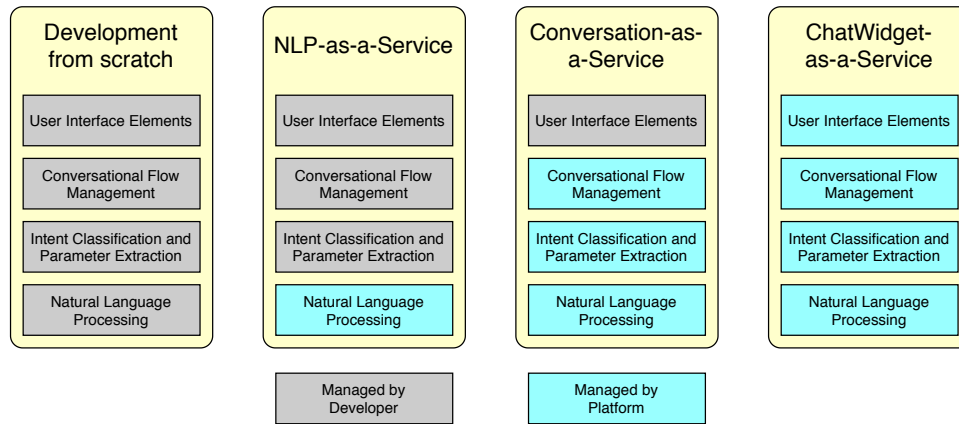


Figure 1: **Solution stacks offered by different Chatbot-building Platforms**

built in-house from scratch or; existing platforms can be used to aid the development. Figure 1 provides an overview of the available solutions. There are platforms which offer basic NLP services (termed as *NLP-as-a-Service* offering), that can be used to compose chatbot-specific functions. There are also specialised platforms that providing basic building blocks for creating a chatbot (grouped as *Conversation-as-a-Service* solutions). Some platforms also offer ready-to-deploy chat widgets for specific mediums (titled *ChatWidget-as-a-Service* platforms). Any of these platforms may be chosen to aid the chatbot-building process.

This decision, in turn, requires the understanding of the core elements of a typical chatbot and its Containing system. This thesis is focused mainly on platform-aided development. The chatbot-building platforms usually provide higher-level design abstractions to define the use cases for the chatbot, freeing up the developers' minds from managing complex Natural Language Processing (NLP) tasks. These abstractions also help in building the chatbot in multiple, short iterations, as is the requirement with several IT solutions today. There are, however, some new design decisions which become a part of the development process, solely because of the use

of these platforms. One such decision involves designing a definition template for defining the chatbot over a platform. These templates can significantly affect the behaviour of the built chatbot. Three popular platforms were used for performing the case studies and experiments presented in this thesis - Google Dialogflow, IBM Watson Assistant and Amazon Lex. The relevant Research Questions, along with our contributions towards their solutions, are as follows:

1. What is the architecture of a typical chatbot? What relationship does it share with its Containing system?

We begin by providing a dissected view of a chatbot to show its major elements. We present five different elements of a chatbot which are part of a typical chatbot. The *Intent Classifier* classifies any query that the chatbot receives in a class. These classes called the *Intents*, are predefined during the chatbot building phase. The *Parameter Extractor* attempts to locate named entities in the query. Similar to Intents, these parameters, called *Entities*, are defined before the chatbot is placed in operation. The *Flow Manager* element manages the discourse with the user. Its job is to make sure that the conversation sounds coherent. The *Response Generator* performs the tasks required to process the query and prepare a response for the user. It may include replying with static messages or invoking a complex processing pipeline that requires interacting with elements of the Containing system. *Voice Utils* are used when a chatbot supports an audio interface in addition to text messages. They perform the Speech-to-Text and Text-to-Speech conversions.

Next, we discuss how the Containing system of the chatbot may affect its design. The *System Interface* refers to the collection of modes through which the Containing system interacts with its uses. Common interfaces maybe a website, an app or via messaging platforms like Messenger, Telegram, WhatsApp or Slack. The chatbot

may be required to face the users at some or all of these interfaces, which can shape its requirements. *Actions* are business operations that the chatbot needs to invoke as part of processing a query, e.g. `create_new_order` function of an e-commerce enterprise. *Fulfilments* are one or more intermediaries between *Actions* and the chatbot, and help in shielding the enterprise’s business operations from the chatbot.

We also suggest a Reference Architecture for any application that contains a conversational interface. The Reference Architecture puts the different pieces of chatbots and the Containing system that we discussed before, in a coherent perspective. As Concrete Architectures for the Reference Architecture, we highlight the variations when different chatbot-building platforms are used for the process.

2. How can chatbot-building platforms be evaluated for their effectiveness for a particular chatbot project?

There are several platforms which can help the chatbot-building process in different capacities. We present three categories of commonly available platforms in this area, as shown in Figure 1. First, The *NLP-as-a-service* platforms provide support for common NLP tasks. They can be used not only for building chatbots but other applications as well (those which process data in Natural Languages). The platforms which we term as *Conversation-as-a-service* solutions, “blackbox” the details of the background NLP tasks and directly provide features and services which are essential to the working of a chatbot, such as Intent Classification, Parameter Extraction and Flow Management. The platforms which we termed as *ChatWidget-as-a-service* offerings, often provide a visual editor to create a conversational flow graph and provide a “chat widget” which may be deployed directly on a medium (usually one or more messaging platforms).

In this thesis, we study the Conversation-as-a-service (CaaS) platforms in detail. They provide enough abstractions for the chatbot building process to take away the worries of implementing core NLP tasks while providing the developers with enough flexibility to compose chatbots for a wide range of use cases. As a part of our analysis, we compiled a list of desirable features in these platforms. The list is in coherence with the Reference Architecture that we proposed. As case studies, we also present relative rankings for the support of these features on three different CaaS platforms.

The *Hospitality framework* is a platform evaluation framework based on Software Architecture Body of Knowledge. It provides a methodology to compare two or more platforms based on their support for achieving project-specific quality goals, such as a Quality Attribute or an Architectural Tactics. We show how this framework can be used to compare candidate platforms for a chatbot project. As a case study, we apply the framework to three platforms for a sample chatbot, assuming that the chatbot needs to achieve high levels of *Modifiability, Security & Privacy, Interoperability* and *Reliability* Quality Attributes.

3. How can a chatbot be defined over a platform?

We discuss the process of defining the use cases that a chatbot is supposed to serve, over a CaaS platform. These platforms expect the definition of the use cases in terms of certain pieces of information. In essence, these platforms enforce a pattern on the definition of a chatbot, which we call as the *Contextual Reactive* pattern. The overall idea of the pattern is to express each use case in terms of a *context*, i.e. what the user expects the chatbot to do; and a *response*, i.e. what the chatbot must do when the context is encountered. The context is defined by declaring a set of Intents and Entities, and supplying some Examples of relevant user queries associated with

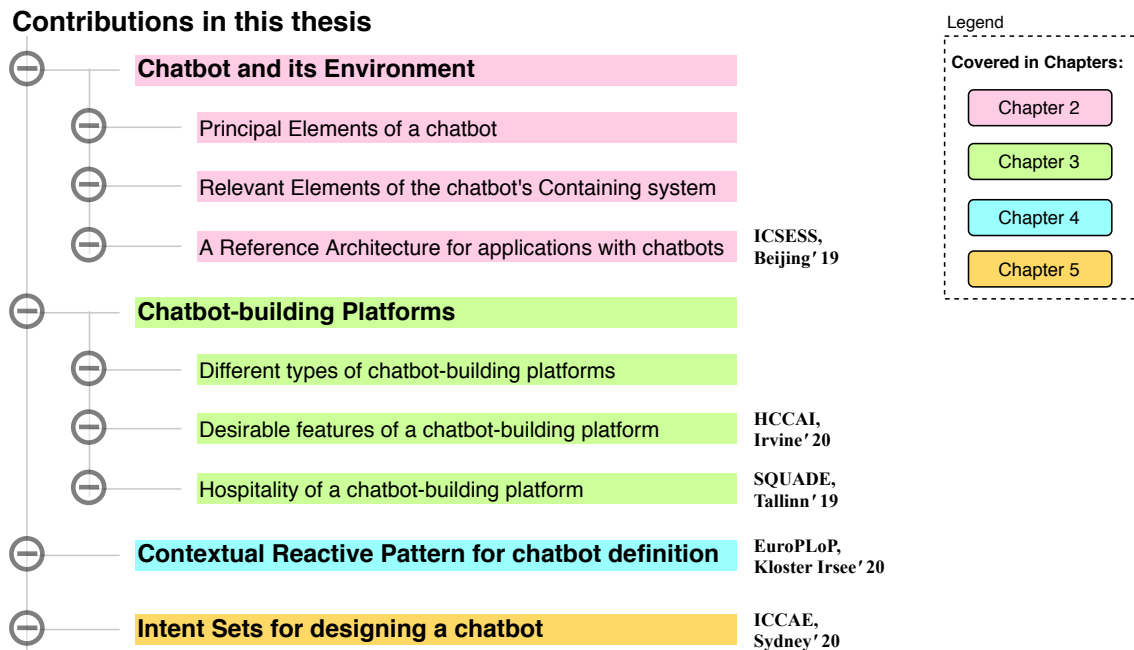


Figure 2: An overview of the contributions in this thesis

them. The response can be defined as static messages, or as a processing pipeline involving the execution of external business logic.

4. Does the process of defining a chatbot on a platform involve any design choices?

We present a design choice that arises because of employing the *Contextual Reactive* pattern for chatbot definition. The choice that we term *Intent Sets*, are essentially different instantiations of the pattern for the same set of chatbot use cases. We show that the same chatbot can be defined in multiple ways, and the behaviour of the built chatbot differs significantly for each case. As proof, we present the results of some experiments that we performed as part of a case study of three CaaS platforms. Our results show that the different versions of the chatbot often provide a different response for the same user query.

The contributions in this thesis are summarised in Figure 2.

Acknowledgements

The list of people who crossed my paths during my PhD is really long. Still, I would like to mention a few. First, I would like to thank my guide, **Dr. T.V. Prabhakar**. It has been more than 10 years now since I met him. All these years, he has been a great mentor, not just professionally, but also at a personal level. I hope that he continues to mentor me, even after I complete my PhD requirements.

Next, I would like to thank the Department of Computer Science and Engineering at IIT Kanpur. In particular, I would like to thank **Dr. Vinay P. Namboodiri**, who provided me with his valuable guidance for some allied work in the field of Privacy Engineering. I am also thankful to all other faculty members, lab staff and office staff who helped me in numerous ways all these years.

The list of friends I made at IIT Kanpur is too big to be mentioned in even multiple pages. Still, some deserve a special mention because of how they made my stay pleasant. To begin with, I would like to thank **Puneet Gupta**, **Deepak Ojha**, **Anando Gopal Chatterjee** and **Ayan Chakraborty** to make my days at Hall-8 memorable. They were the main reasons why I loved Hall-8 so much. I would also like to mention **Naman Bansal**, who always treated me as a mentor, while for me, he has always been like a younger brother. Towards my later days at Hall-8, I interacted with **Arunava Karmakar**, a gem of a person, with whom I had many great conversations in the Mess. I would also like to mention **Aditya Desai** and **Nishigandha Patil**. They have made the campus a much better place for animal lovers like me. I hope they continue their great work wherever they go.

Next, I would like to thank my friends at my lab, with whom I shared my moments of joy, grief and frustration. I probably spent more time at KD106 than my hostel room. This is why I am grateful to **Awanish Pandey**, **Hrishikesh Terdalkar**, **Shubhangi Agarwal** and **Rujuta Pimprikar**, **Aakanksha Verma**,

Arvapalli Sai Susmitha and **Sumit Lahiri** in KD106. They all ensured that the lab remained a lively environment for work and relaxation.

As the strength of the PhD candidates at CSE Department has swelled, I may not have been in touch with much of the newer additions. However, I still spent a lot of time with many fellow PhD scholars. I have had so many discussions with **Pawan Kumar**, a fellow PhD scholar and my M.Tech senior, not only on topics related to Computer Science but politics and life in general. **Tejas Gandhi** and **Adarsh Jagannatha** have been my round-the-clock troubleshooters. They are probably the only guys in the Department whom I envy for their skills. A special mention to **Ashish Agrawal**, who has almost always been a defacto co-guide to me. He probably knows the details of each of my work, as much as I do. I would also like to thank **Sumit Kalra** for collaborating with me for our work related to Design Patterns. My discussions with **Siddharth Srivastava**, **Sai Charan Putrevu** and **Arun Jain** are also going to be a part of my memory.

My parents have been as supportive as parents could possibly be. **My father**, even though he has no understanding of my work, tried to find journals for me, whenever I was down after a rejected publication. I could sense how much he wanted to help from the efforts he would put in to do so, even at his age. **My mother** has always been the strict one in the family about academics. She has always inspired me to beat the best in whichever test I took up. Her belief in me and my capabilities often astonish me, but at the same time, makes me work a tad harder to achieve my goals. For the greater part of our married life, my wife and I have not stayed together. The idea was to help me do my work with minimal distractions. I, therefore, must thank my wife, **Swati**, for understanding our situation, and being a remote, yet substantial, support for my PhD work. My son, **Shaurya**, is too young to know why most of his childhood went away without his father, giving him more time. Albeit very young, his sacrifice helped me complete my work with little distraction. In the end, I would like to express my unparalleled love for my late brother, **Gaurav**. He was my first career guide, and it really hurts that he is not around today to see me fulfil one of his own dreams. I hope wherever he is, he finds solace that I am trying my best to be the brother of his thoughts.

Saurabh Srivastava

Dedicated To

My Family and My Guru

Contents

List of Tables	xvii
List of Figures	xix
1 Introduction	1
1.1 Literature Review	3
1.1.1 History of Chatbots	3
1.1.2 Selected Previous Work on Chatbot Design	5
1.2 Challenges in Building Chatbots	8
1.3 Research Questions	12
1.4 Thesis Organisation	13
2 Chatbot and its Environment	17
2.1 Architecture of the Chatbot	18
2.1.1 Voice Utils	18
2.1.2 Intent Classifier	19
2.1.3 Parameter Extractor	19
2.1.4 Response Generator	20
2.1.5 Flow Manager	21
2.2 Architecture of the Containing System	23
2.2.1 Fulfilments and Actions	23
2.2.2 System Interface	24
2.3 Reference Architecture for the Application	24
2.3.1 Concrete Architecture using Dialogflow	27
2.3.2 Concrete Architecture using Watson Assistant	29

2.3.3	Concrete Architecture using Lex	31
2.4	Related Work and Further Reading	31
2.5	Summary	34
3	Chatbot-building Platforms	35
3.1	Reasons for Custom Development	36
3.2	Types of Chatbot-building Platforms	39
3.2.1	NLP-as-a-Service Platforms	40
3.2.2	Conversation-as-a-Service Platforms	41
3.2.3	ChatWidget-as-a-Service Platforms	42
3.3	Dashboards on Chatbot-building Platforms	43
3.3.1	CWaaS Platforms	43
3.3.2	CaaS Platforms	47
3.4	Desirable Features of a Chatbot-building Platform	50
3.4.1	Features for Intents Management	51
3.4.2	Features for Entities Management	54
3.4.3	Features for Defining Fulfilments	56
3.4.4	Features Related to Integrations	59
3.4.5	Features for Managing Conversational Flow	61
3.5	Hospitality of a Chatbot-building Platform	63
3.5.1	Understanding Hospitality	63
3.5.2	Using Hospitality	66
3.6	Case Studies	72
3.6.1	Sample application of the Hospitality Framework	72
3.6.1.1	Phase 1: Identify Quality Attributes	72
3.6.1.2	Phase 2: Identify Architectural Tactics	74
3.6.1.3	Phase 3: Identify Platform Features	76
3.6.1.4	Phase 4: Evaluate Platforms	77
3.6.1.5	Phase 5: Calculate Hospitality Indices	77
3.6.2	Support of Desired Features in Three Platforms	80
3.7	Related Work and Further Reading	81
3.8	Summary	86

4	Contextual Reactive Pattern	89
4.1	Pattern Overview	90
4.1.1	Context	90
4.1.2	Problem	91
4.1.3	Forces	91
4.2	Solution	92
4.2.1	Structure	96
4.2.2	Dynamics	97
4.2.3	Deployed Chatbot's Sketch	98
4.3	Pattern Examples	100
4.3.1	Dialogflow	100
4.3.2	Watson Assistant	102
4.3.3	Lex	103
4.4	Case Study	104
4.4.1	User Stories	104
4.4.2	The First Sprint	106
4.4.3	The Second Sprint	111
4.4.4	Comparison of Sprint Deliverables	114
4.5	Consequences	117
4.6	Summary	118
5	Intent Sets	121
5.1	Intent Sets	122
5.1.1	Properties of Intent Sets	125
5.2	Case Study	127
5.2.1	Experimental Setup	129
5.2.2	Experiments	135
5.3	Observations	139
5.3.1	The <i>Accuracy Experiment</i>	139
5.3.2	The <i>Order Experiment</i>	141
5.3.3	Discussion	142
5.4	Related Work and Future Reading	143
5.5	Summary	144

6	Conclusions and Future Work	145
6.1	Conclusions	145
6.2	Future Work	149
	Appendices	151
A	Guide to Privacy Policy Resources of Selected Platforms	153
B	Explanations for Selected Values in Table 3.12	157
C	Comparison of the Elements of <i>Contextual Reactive Pattern</i>	161
	References	165
	Index	189
	Publications	189

List of Tables

3.1	Intents Management Features	52
3.2	Entities Management Features	54
3.3	Fulfilments Management Features	57
3.4	Integrations Management Features	60
3.5	Conversational Flow Management Features	61
3.6	Finding Tactics for Quality Attributes	75
3.7	A comparison between possible conversations for the Fruit Seller chatbot, with and without digression support	76
3.8	Using Platform Features for calculating Hospitality Index, at Tactic level	78
3.9	Availability of useful platform features in three candidate platforms	79
3.10	Hospitality Indices at Tactic level for the three candidate platforms	80
3.11	Hospitality Indices at QA level for the three candidate platforms	80
3.12	Relative support of the Desired Features on three platforms	82
5.1	An Example Table containing data related to phones in an inventory	122
5.2	Possible Intents for the <i>phoneBot</i>	123
5.3	Two possible Intent Sets for <i>phoneBot</i>	124
5.4	Some rows from the table <i>c</i> . The Shaded columns are “output” attributes, used for preparing responses.	130
5.5	A part of training data for the <i>statistical</i> Intents in <i>Intent Set₁</i>	133
5.6	A part of training data for the <i>statistical</i> Intents in <i>Intent Set₂</i>	134
5.7	Dissimilarity Scores on Dialogflow and Watson Assistant	142

List of Figures

1	Solution stacks offered by different Chatbot-building Platforms	v
2	An overview of the contributions in this thesis	ix
1.1	Typical stages of development for a traditional component	10
1.2	Typical stages of development for a component employing ML techniques	10
1.3	Organisation of the thesis along with major contributions	16
2.1	The Logical View of a chatbot	18
2.2	The Logical View of the chatbot's Containing system	23
2.3	A Reference Model for the Containing system	25
2.4	Proposed Reference Architecture for the Containing system	26
2.5	Example Concrete Architecture for a Containing system when Dialogflow is used for building the Conversational subsystem	28
2.6	Example Concrete Architecture for a Containing system when Watson Assistant is used for building the Conversational subsystem	30
2.7	Example Concrete Architecture for a Containing system when Lex is used for building the Conversational subsystem	32
3.1	Solution stacks offered by different Chatbot-building Platforms along with some examples	39
3.2	A view of the ManyChat Visual Editor, clipped from an image at [1]	42
3.3	A sample interaction with Lufthansa Airlines Chatbot [2] on Messenger	44
3.4	Screenshots from the Chatfuel dashboard	45
3.5	Screenshots from the Dialogflow dashboard	48

3.6	Screenshots of the Watson Assistant Dialog Tree	49
3.7	Top-level categories of desired platform features	51
3.8	A sample conversational session between a chatbot and a user	62
3.9	Different phases in application of the Hospitality Framework	66
3.10	Examples of <i>Feature Cards</i> for two different features and platforms	68
3.11	An overview of the Hospitality Indices Calculation process	71
3.12	Software Architect's notes, representing the requirements for the Fruit Store applications	73
4.1	Building Chatbots using a platform with the <i>Contextual Reactive</i> pattern	93
4.2	Solution Structure : How Entities, Intents, Examples and Fulfillments interrelate to each other in an application.	96
4.3	Solution Dynamics : How a platform uses Entities, Intents, Examples and Fulfillments for building the chatbot.	98
4.4	Deployed Chatbot's Sketch : How the platform uses the built NLP models along with the defined Fulfillments to construct the chatbot.	99
4.5	Deployed Chatbot's Sketch : Typical workflow observed during the operation of a chatbot.	99
4.6	Chanakya Airlines Chatbot: Handling use cases of future iterations	115
4.7	Chanakya Airlines Chatbot: Updating implemented use cases	116
5.1	Sample Conversations with <i>Cricket Novice</i>	128
5.2	Snapshot of a document containing some basic information about the game of Cricket. Each paragraph can be read independently as a different document.	129
5.3	The <i>statistical</i> Intents for Cricket Novice . The <i>descriptive</i> and default Intents were common in both Intent Sets, and hence, not shown.	131
5.4	A rough sketch of processing of a user query when the two Intent Sets shown in Figure 5.3 are put in use	139
5.5	Utility Scores for <i>Cricket Novice</i> on different platforms	140

Chapter 1

Introduction

A chatbot is a software component that can interact with a human being in some Natural Language, such as English. A survey that was taken by 800 top-tier business executives in 2017 estimated that four out five businesses want to have a chatbot by the year 2020 [3]. The respondents, which included Senior Sales Executives, Senior Marketers and Chief Marketing Officers explained that they are looking to employ chatbots for different tasks associated with Marketing, Sales as well as Customer Support. The domains where chatbots are operating include E-commerce, Digital Healthcare, Legal Advisory and News Reporting, among others [4]. Chatbots provide a Conversational Interface to an application, which the businesses intend to use to their advantage.

Simple chatbots can be built primarily using Regular Expressions (regexes) [5] [6]. However, regexes have their limitations. They cannot, for instance, be used to capture semantic variations of the same concept or phrase (e.g. a regular expression for `man` cannot capture `human`, even though they may have the same intention in a conversation). While regexes are certainly useful, they are usually employed for only localised tasks associated with a chatbot, for instance, for identifying an email address or a phone number. Almost all chatbots built today use some form of *Artificial Intelligence* (AI). Two common flavours of AI systems are *Rule-based* systems and systems based on *Machine Learning* (ML). Sometimes the terms ML and AI are used interchangeably in the context of chatbots, essentially disregarding Rule-based approaches as an example of AI [7] [8] [9].

The operation of Rule-based chatbots is similar to following a flow-chart. For

any received user query, a series of rules are evaluated to arrive at a juncture where a response can be sent back to the user. It makes them ideal for implementing a finite set of use cases, albeit with a backup mechanism of transferring the control to a human being, in case the chatbot cannot apply any of the available rules successfully [7]. Thus, the process of “training” a Rule-based chatbot involves mining more rules from customer interactions, especially when the chatbot failed to apply any existing rule. Despite their limitations, Rule-based chatbots have the advantage of simplicity and explainability, i.e. their behaviour can be accurately predicted and modified. For some businesses, they may be enough to cater to the requirements [9].

Another approach to building chatbots involve “training” the chatbot with data. This approach, based on ML techniques, involves building “models”, which perform the core Natural Language Processing (NLP) tasks that a chatbot is required to perform. An important task associated with a chatbot is to identify the overall intention of a user query (e.g. is it an enquiry related to buying a new product or a complaint about a sold product). It is because queries with different intentions usually have to be processed differently. Another task for the chatbot is to spot instances of any relevant input data in the user’s query, which may be necessary for processing it (e.g. parsing the name of the product from the user’s statements). Many platforms that aid in building chatbots perform these tasks behind the curtains. The developer is only required to supply some sample data to the platform, and the chatbot is equipped with models to do these tasks automatically.

A system that employs chatbots has Architectural Issues, similar to any other system. In this work, we investigate these issues and present our contributions towards enhancing the Software Architecture Body of Knowledge in the domain. The rest of this chapter is organised as follows; Section 1.1 presents an overview of relevant previous works related to chatbot’s history and design. Section 1.2 discuss the reasons why studying the architectural issues associated with chatbots and systems with conversational interfaces in general, is of interest. 1.3 provides a brief introduction to the Research Questions that we address in the current work. 1.4 provides an overview of our contributions and their discussion across various chapters.

1.1 Literature Review

The current state-of-the-art of the chatbots is fueled by much work in the past decade. We start by presenting a brief history of chatbots. An inspection of this history also provide hints to the challenges associated with a chatbot. While some of these have been resolved to some extent, others remain prominent. We then proceed to discuss some relevant related work, i.e. previous contributions that discuss some architectural aspects of chatbot development.

1.1.1 History of Chatbots

The first attempt at building a chatbot, materialised in 1966, was ELIZA [10]. ELIZA was built with the help of pattern matching and a set of rules which picked one response out of a limited set of possibilities. ELIZA could operate over “scripts” - a collection of rules and responses to take discourse in a conversation. The most famous script of ELIZA was one where it pretended to be a psychotherapist. A major achievement often associated with ELIZA was the reaction of users interacting with it. People shared complex thoughts with ELIZA as if it was a human being.

Next major milestone for chatbots is considered to be PARRY, which came out in 1972. PARRY pretended to be a patient of paranoia [11]. When interacting with PARRY, more than half of the psychiatrists were not able to differentiate between PARRY and a real patient of paranoia. The findings were made over a modified version of the Imitation Game proposed by Alan Turing in 1950 (commonly referred to as the Turing Test [12]). Even though PARRY too produced responses from a limited set, it used a complex mathematical model based on “mistrust”, a common problem with people who suffer from paranoia. From the perspective of processing inputs in a Natural Language, PARRY was significantly sophisticated as compared to ELIZA.

While ELIZA and PARRY tried playing specific roles, Jabberwacky [13] was aimed to be a general chatbot, intending to impress people with casual conversations. The project started in 1988, and it was launched on the internet in 1997. Jabberwacky did something that its predecessor did not do - it collected the responses provided by the users and added them to its database. It allowed Jabberwacky to update its responses over time, rather than using a fixed set of pre-defined responses.

The responses of Jabberwacky are chosen through a process that the author, Rollo Carpenter, terms as “contextual pattern matching”. There are no official descriptions of what the term means or how the chatbot works. Nevertheless, Jabberwacky boasted itself of being a chatbot driven by AI, and that it is unique when compared to its predecessors. Its successor, Cleverbot [14] has now replaced Jabberwacky.

Dr. Sabaitso [15], another chatbot that pretended to be a psychologist, appeared around 1991-92. It did not differ much in capability from ELIZA, repeating certain questions like “WHY DO YOU FEEL THAT WAY?” often. However, its major significance is in the field of speech synthesis. Dr. Sabaitso could “speak back” to its users (although the users still had to type their responses). It was distributed with various sound cards and worked over the Microsoft DOS operating system. Although the voice did not sound “humanly”, it is still considered one of the early attempts at generating speech, an attribute which is relatively common in many modern-day chatbots.

In 1995, Richard Wallace developed a chatbot called Artificial Linguistic Internet Computer Entity or A.L.I.C.E. It was inspired by ELIZA and powered by an XML dialect called the Artificial Intelligence Markup Language (AIML). A.L.I.C.E. won the Loebner Prize thrice (an annual competition in the field of artificial intelligence), even though it could not pass the Turing Test. A.L.I.C.E. showed the power of AIML, a simple language to express pattern-based rules for a chatbot. AIML can be used to define general patterns using wildcards, and configure templates to respond to these patterns [16]. Mitsuku [17], a chatbot built over AIML, initially launched in 2005, has won the Loebner Prize five times, the most recent one in 2019.

By the time Elbot (2000) [18] and SmarterChild (2001) [19] came out, chatbots had become far more sophisticated. Their creators openly bragged the use of AI; however, the details of the workings of the chatbots were seldom revealed. Elbot also won the Loebner Prize and is known to provide responses filled with sarcasm. SmarterChild was a chatbot that interacted with users of AOL Messenger. The idea was to provide basic information to users about weather, stocks or movie timings.

The beginning of the modern chatbot era can probably be traced to the success of IBM Watson [20], whose development started around 2005-06. Although Watson was not precisely a chatbot in the conventional sense (a more accurate term being question-answering system), it did increase the confidence of researchers in handling

NLP tasks with much higher precision. In 2011, it defeated two former champions of the popular television show called *Jeopardy!*, making a strong case for machines performing tasks that were thought to be doable by humans only. In the show, the contestants were given clues in the form of answers, and they were supposed to frame a question to match the same. It was an effort which involved processing a huge volume of data in a limited amount of time to come up with a response. IBM has since launched many AI products and services, including their chatbot-building platform called Watson Assistant [21]. Today, there are more than 50 business entities that provide support for building chatbots [22], all of them being founded in the post-Watson era (after 2008).

Finally, a special class of chatbots that we must mention here are those who fall in the category of Assistants. Apple’s Siri [23], Google’s Assistant [24], Microsoft’s Cortana [25] and Amazon’s Alexa [26] are some examples. These chatbots have evolved into an environment, where multiple pluggable applications co-exist as part of an ecosystem. They can perform common tasks on their own (e.g. searching for a keyword on the internet). At the same time, more specific behaviours can be interacted with by invoking them explicitly (e.g. an app on Google Assistant can be invoked by saying “talk to japp name;”). Some of these environments also provide a platform to build chatbot applications (e.g. [27] and [28]).

1.1.2 Selected Previous Work on Chatbot Design

Brandtzaeg et al. [29] present an interesting discussion over the need to rethink our user interfaces in the era of the Chatbot revolution. Some work has already started towing this line (e.g. [30] and [31]) as more systems are built with conversational interfaces ([32] [33] [34] etc.). The contribution by Allen et al. [35] is a good starting point to understand the basic aspects of building conversational components. It talks about many design decisions that are relevant when building a chatbot. For instance, the Model-View-Controller (MVC) [36] approach to building systems inspires architects to decouple the user interface from the business logic (models and controllers). The idea behind reducing this coupling is that both can be changed with minimal effect on each other. Allen et al. describe how the design of the user interface can significantly change the capability of the chatbot. For example, if the

user interface has a “push-to-talk button”, the chatbot needs to have a capability to process multiple statements from the user in one go. On the other hand, in an “open-mike” scenario, where both the chatbot as well as user can communicate any time they wish, the chatbot must decide when is it its turn to speak (and when the turn is with the user). The discourse management in both these cases will be very different. They divide the core responsibilities of the chatbot into three co-operating major categories. The *Interpretation Manager* (IM) deals with issues such as categorising an input statement as a problem (e.g. “the room is locked”) or the initiation of a new goal (e.g. “the room needs to be opened”). The IM parses the user utterances and decides upon what type of processing must be initiated. The *Behavioral Agent* (BA) is where the core business logic associated with the chatbot lies. BA is responsible for actually performing the tasks that are required at a particular stage (e.g. finding the solution to the user’s problem). BA is connected to the rest of the world and can initiate events in, and receive notifications from the outer world. The *Generation Manager* (GM) is responsible for the system’s discourse. It receives inputs from the BA, as well as any discourse-related information created by the IM (e.g. the turn information) and plans when and what to generate as the response.

The work by Allen et al. is more at a conceptual level. Pilato et al. [37] describe a “modular architecture” for building chatbots. They too define three significant categories of modules. The modules associated with the *Dialogue Engine* class are responsible primarily for parsing the user’s inputs for various reasons as per the requirement. For example, there could be a module which classifies the user’s tone as positive, negative or neutral. Another module may detect synonyms of specific terms from a dictionary and may replace them before any further processing takes place. The task associated with modules of *Dialogue Analyser* includes looking for information in the inputs which are required for processing the query. It may include modules to detect the topic of current conversation (e.g. is it about buying a product or initiating a return), details related to the topic (e.g. a product’s name) and any contextual information (e.g. is the current user permitted to perform the requested action or eligible to get the sought information). Based on the processing, a set of “context variables” are prepared, which essentially represent a state of the system. Finally, the modules related to *Corpus callosum* use the values of these variables to pick one out of many possible modules to perform the relevant task and return a

suitable response. In the paper, they show the use of a Bayesian network to activate the right module over a case study.

Other than the above, the Senior Thesis work by [38] by Cahn can also act as a quick primer on the underlying tasks that a chatbot needs to perform. The work covers the major alternatives available to implement various stages of a chatbot such as Speech-to-Text Conversion, NLP, Response Generation and Text-to-Speech Conversion. The work majorly covers algorithmic aspects of these tasks, discussing possible implementation techniques that may be employed for each phase. Another work which stands out from the rest is the application of the *i* modelling framework* [39] over chatbot design process is by Babar et al. [40]. The framework is intended to model goals and their dependencies on actors. They divide all the chatbots into two major categories (Retrieval-based and Generative chatbots) and present their envisioned models for both. In addition to papers, there are a number of articles and blogs ([41] [42] [43] [44] to mention a few) which discuss almost the same concepts as discussed in [38] at abstract levels.

Lastly, there are articles which talk about building chatbots, with concepts that are more aligned with modern chatbot-building platforms ([45] [46] [47] [48] are some examples). This is because most of the platforms enforce a definition pattern for defining a chatbot. We discuss this pattern in detail in Chapter 4. This definition pattern that we call the *Contextual Reactive Pattern* abstracts the core NLP tasks associated with the chatbot (which are now handled in a “blackboxed” fashion). It allows the developers to concentrate on the business use cases that the chatbot has to serve. The highlights of the works we discussed are summarised below:

- The chatbot needs to implement some core NLP tasks to parse the user query for extracting essential information.
- There needs to be a management of the overall conversation with the user to allow flexibility and maintain coherence.
- The business logic that a chatbot needs to access, in order to perform the expected tasks, should be kept separate from the core tasks of the chatbot.
- The user interface of the chatbot is crucial. It may dictate the core tasks that the chatbot must perform.

1.2 Challenges in Building Chatbots

The recent rejuvenation of chatbots can be largely owed to advancements in Information Retrieval (IR) though Neural Networks (some of the works which cover them in detail are [34] [49] and [50]), which in turn, often rely on ML techniques [51] [52] [53]. The ML techniques are usually better at handling challenges where it is difficult to write down a set of finite rules to perform the task. Some of the challenges faced by a chatbot which fall in this category are:

- **Handling variations in Natural Languages:** Chatbots deal with Natural Languages. Natural Languages have several variation points. For instance, the same word may be pronounced with significantly different accents in different geographic locations. A model that deals with converting the speech fragments to the corresponding text must have enough data to cover all these accents, or, the conversion may have severe errors. Other examples include use of sarcasm (the literal meaning is different from the intention), use of synonyms and homonyms and handling statements in both Active and Passive Voice. Thus, it is practically intractable to provide data that can cover all possible variations for any of the core NLP tasks. This makes a strong case to use ML techniques for the same to interpolate cases where data is unavailable.
- **Expectation of worldly knowledge:** Usually the expectations from a software component are well-defined. Consider the case of a Cab-booking System with a web form to fill out the details of the user's demands. The form asks the user to specify all the required and optional information (such as Source and Destination address, time and date for the cab and preferences about the driver) in text-boxes. The user supplies this information, clicks on a button called "book", and the cab is booked. Now imagine the Cab company employing a conversational interface for the same. On asking the user, "What time do you want the cab?", she replies with "3 in the morning". What she means is "3 AM", but the chatbot may be misled by the word "morning", not able to grasp that the term contextually means past-midnight here. A chatbot often has to deal with such scenarios, where it is expected to have "common sense". In general, it is better to let a model *learn* these details through an appropriate

ML technique from the supplied data than to encode them explicitly.

- **Constant self-update to be in touch with the real-world:** The interfaces that a traditional software component provides does not change so often. Controls like Text-boxes, lists, combo-boxes and radio buttons have been around for a long time and expected to be around in the near future too. Their operation and semantics are well-understood. However, Natural Languages have their evolution graphs. A particular word or phrase may suddenly have a new meaning, or a rarely used word may become popular overnight. The chatbot must be updated periodically to handle these new variations in the Natural Language. It may be too difficult for humans to track these changes and update the rules manually. A feasible alternative is to employ some ML technique which can update the models “online”, i.e. while they are in use [54].

It is therefore nor surprising for most of the modern-day chatbots to use ML techniques for one or more of their core tasks. However, while ML techniques certainly have their advantages, they do have their liabilities as well. Figure 1.1 and 1.2 capture the difference between the development process with and without the use of ML techniques respectively.

For a traditional component, the development process can be simplified as follows; the developers write code, implementing the requirements related to the component. A set of tests are then performed over the code. If all the tests pass, the component is deployed. If even a single test fails, the developers debug the code, find the problems, rectify it, and the cycle is repeated.

In contrast, the development of an AI component involves building *models* alongside writing code. A set of tests are then run on the component, which primarily evaluates the inferencing capabilities of the models. Depending on the component, this evaluation will produce some results. If the results are “unsatisfactory”, the developers retrain the models. This may involve steps like adding more training data, filtering existing data to remove noise and tweaking the meta-parameters of the learning technique. While these techniques have proven their mettle when it comes to achieving high accuracy rates, they also have issues related to *explainability* [55] [56]. In particular, *Deep Learning* methods, arguably the most popular of

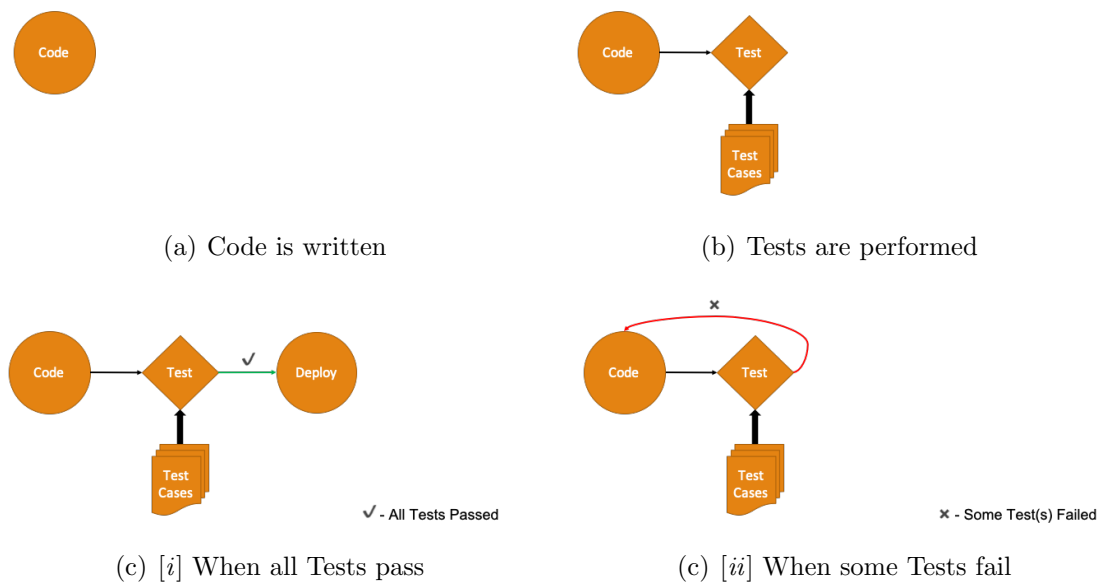


Figure 1.1: Typical stages of development for a traditional component

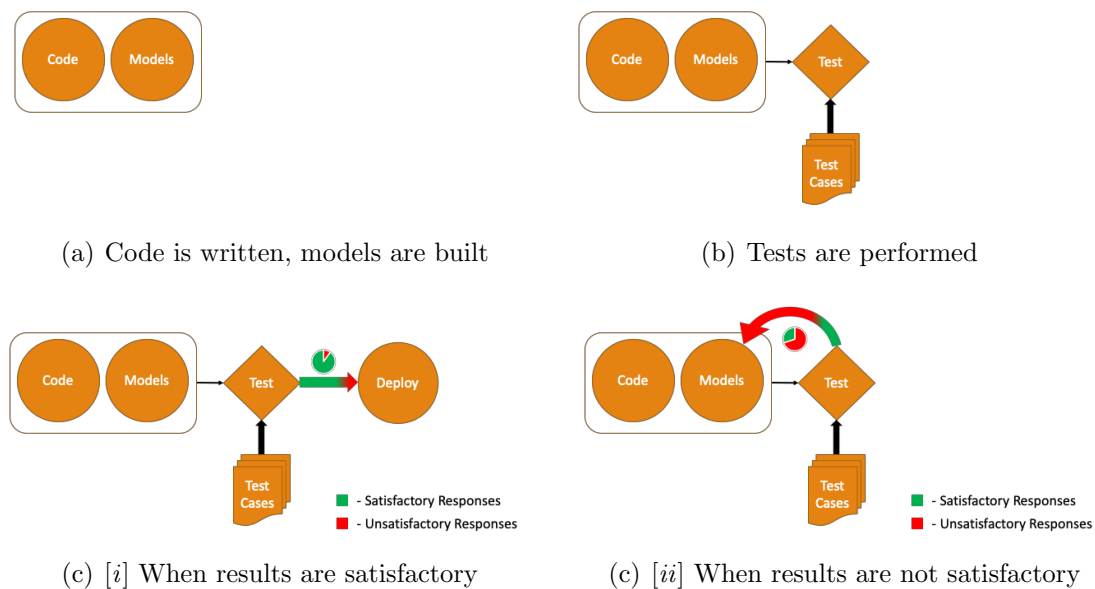


Figure 1.2: Typical stages of development for a component employing ML techniques

all ML techniques today, suffer from this issue significantly. This is why it is usually not possible to explain why the models perform poorly over specific inputs, and consequently, it is challenging to change models with surgical precision. The cycle to build new models and evaluate them continues, until the results are “satisfactory”. Here, satisfactory does not mean that they do well over all the inputs; it means that they do reasonably well over a wide array of inputs.

Most of the chatbots built using popular platforms today, follow the same development process, as shown in Figure 1.2. The developers provide some training data to the platform. The platform builds some models in the background (usually the details of this process is hidden from the developer) to perform the core NLP tasks associated with the chatbot. The developers evaluate the quality of the chatbot by firing some queries at the built chatbot and comparing the response with the expected output. If the chatbot’s responses are “good enough”, it is deployed. Otherwise, the developers repeat the cycle. These platforms provide sophisticated dashboards for every stage to make the overall process as smooth as possible.

We can now conclude that the biggest challenge in building a chatbot today is *handling uncertainties*. Developers often use ML techniques to avoid writing rules for core NLP tasks associated with a chatbot. Most chatbot-building platforms provide access to these tasks, through easy to use dashboards. The developers are only expected to provide training data, and the platform builds a chatbot for the use case, building the necessary models. Even though it speeds up and simplifies the development process greatly, the chatbots built this way are not perfect. For some inputs, they fail in one or more ways. What complicates the problem further, is that the inputs over which the chatbot would fail cannot usually be predicted before deployment. Provided that the testing phase can only test a handful of possible utterances that a chatbot will receive, this leaves a quantum of uncertainty in the behaviour of the chatbot. This uncertainty can create a ripple effect. For example, the chatbot may initiate an action in the real world, which the user did not intend. Therefore, the overall Containing system housing the chatbot needs to plan for this uncertainty by putting fail-safe mechanisms at different stages. This provides a motivation to study not only the chatbots but also the systems which employ them. In the current work, we investigate the architectural issues associated with these systems and highlight some of their prominent design decisions.

1.3 Research Questions

The questions that we investigate in this work are summarised below:

- What is the architecture of a typical chatbot? What relationship does it share with its Containing system?

Building a chatbot from scratch would require implementing some core NLP tasks. Even when a platform is used for building the chatbot, a broad understanding of these tasks can be helpful while crafting a training dataset. An understanding of these pieces in the bigger puzzle can also help tweak the meta-parameters of any ML techniques being employed in the chatbot. The Containing system that the chatbot eventually gets integrated with is also impacted by the addition of the conversational components. This impact can be seen in reverse as well, i.e. the chatbot's architecture is also affected by the architecture of the Containing system. A possible way to show these relationships is using Reference Architectures. Reference Architectures do not enforce strict constraints on the architecture of a system, but rather, provide hints or guidelines for building these systems. Thus, coming up with a Reference Architecture for systems with chatbots is a vital research objective that we attempt to achieve.

- How can chatbot-building platforms be evaluated for their effectiveness for a particular chatbot project?

There are several chatbot-building platforms that offer different types of offerings. Analysing their capabilities, and grouping them into various categories is an important starting point in their study. Platforms usually aim to provide a set of features which can help with the implementation of common use cases. For any chatbot project, there are specific requirements and constraints. Based on these, different chatbot-building platforms may be less or more useful. Studying the common features that these platforms provide, and then, evaluate their usefulness for a particular project is an interesting problem. A possible way to do this evaluation is with the help of Quality Attributes. We propose the use of the Hospitality Framework for the same. The framework,

which has been applied on Cloud platforms before, can be used for evaluation of chatbot-building platforms as well.

- How can a chatbot be defined over a platform?

The requirements for any project is usually collected in the form of standard templates. For instance, for Agile development, requirements are often expressed in the form of User Stories. When a chatbot-building platform is used, these use cases must be defined in a form that the platform can understand. We show how the platforms essentially enforce a pattern in this regard. The developer must map the use cases in terms of this pattern. In this work, we present this pattern which we call as the Contextual Reactive Pattern. We also discuss the advantages and limitations of this pattern.

- Does the process of defining a chatbot on a platform involves any design choices?

Consider a query that a chatbot is supposed to answer - “Who won the Men’s Gold Medal for 100m Sprint in the last Olympics?”. The query can either be called as a “query about *Male athletes*” or a “query about *last Olympics*”. In essence, both of these are perspectives or views to see the same data. The impact of using one of these perspectives over the other on the chatbot is worth studying. In this work, we introduce the concept of Intent Sets. When a chatbot is defined over a platform, the developer can do so in more than one way. Each Intent Set represents one such way. We present the results of some experiments to comment on the importance of Intent Sets in the development of a chatbot.

1.4 Thesis Organisation

The thesis is organised into six chapters. A concept map showing the major contributions, as well as the chapters which cover them is shown in Figure 1.3. The current chapter was devoted towards introducing chatbots, present a historical perspective of towards their design and a brief overview of the research problems we have attempted to solve.

Chapter 2 discusses the chatbot and its environment in detail. First, we discuss the core elements of a chatbot. The tasks that these elements perform can either be done through custom implementation or provided as part of a solution developed with a platform. Second, we talk about how the architecture of the Containing system is affected by the chatbot and vice versa. Third, we present a Reference Architecture for end-to-end applications having conversational capabilities. This Reference Architecture is designed assuming that a platform provides a significant part of the chatbot solution. We also present three Concrete Architectures derived from this Reference Architecture. These architectures essentially highlight the differences between the offerings of these platforms.

Chapter 3 investigates the next phase of the architecting process, i.e. provided that a platform is to be used for building the chatbot, how do we evaluate possible candidates? We begin by presenting the application of the *Hospitality Framework* on chatbot-building platforms. We walk through the different phases of the framework by picking a set of simple chatbot use cases. We show how this analysis could be done for the sample use cases over three popular platforms - Google Dialogflow, IBM Watson Assistant and Amazon Lex. The application of the Hospitality Framework requires mapping the required use cases to a set of Quality Attributes. The application achieves each Quality Attribute (QA) with the help of one or more Tactics, which, in turn, require some support from the platform for realisation. The support from the platform can be evaluated in terms of their exposed features. We conclude the chapter by presenting a hierarchical list of *desirable features* in a chatbot-building platform. We also present a report on their relative support in the three platforms mentioned above.

Chapter 4 initiates a discussion about the phase of defining a chatbot on a platform. Most of the platforms seek the definition of these use cases in a particular format. It means that the developer must map the use cases into a definition pattern in order to use the platform. We name this pattern the *Contextual Reactive Pattern* for defining a chatbot. We show that the pattern forces the developer to model all the use cases in terms of two major elements - a context, and an associated reaction. A context captures a pool of user queries which can be naturally grouped under one category, e.g. “pricing related queries on an e-commerce portal”. The reaction provides details of how the chatbot should respond, in case it encounters

that particular context. We discuss how this definition pattern can support the development of a chatbot over multiple iterations. To show that, we walk through the implementation details of a sample chatbot over two iterations, and show how the pattern allows developers to take care of some of the *uncertainties* that we discussed in Section 1.2.

We present the final contribution of this thesis in Chapter 5. In general, the use cases of a chatbot can be mapped to the *Contextual Reactive* Definition Pattern in ways than one. Here, we introduce the concept of *Intent Sets*. An Intent Set represents one of many ways to define the same set of chatbot use cases. We show that even though theoretically, all of these Intent Sets should be equivalent to each other, but in practice, they are not. To show how widely the behaviour of the defined chatbot varies with Intent Sets, we pick a particular class of chatbots called the Information Retrieval chatbots. These chatbots process a user query to fetch information from a data source, such as a table. We pick these chatbots because they can be easily evaluated in a quantitative setting by inspecting the data items that they pulled from the data source while processing a request. We can evaluate if their processing was correct, incorrect or partially correct by comparing their response with the expected response. We show the results of our experiments performed over three platforms with two Intent Sets for the same sample chatbot. The results show how picking one Intent Set over another can significantly change the behaviour of the chatbot, considering the same training data was used in all cases.

The Conclusion and pointers to the Future Work are provided in Chapter 6.

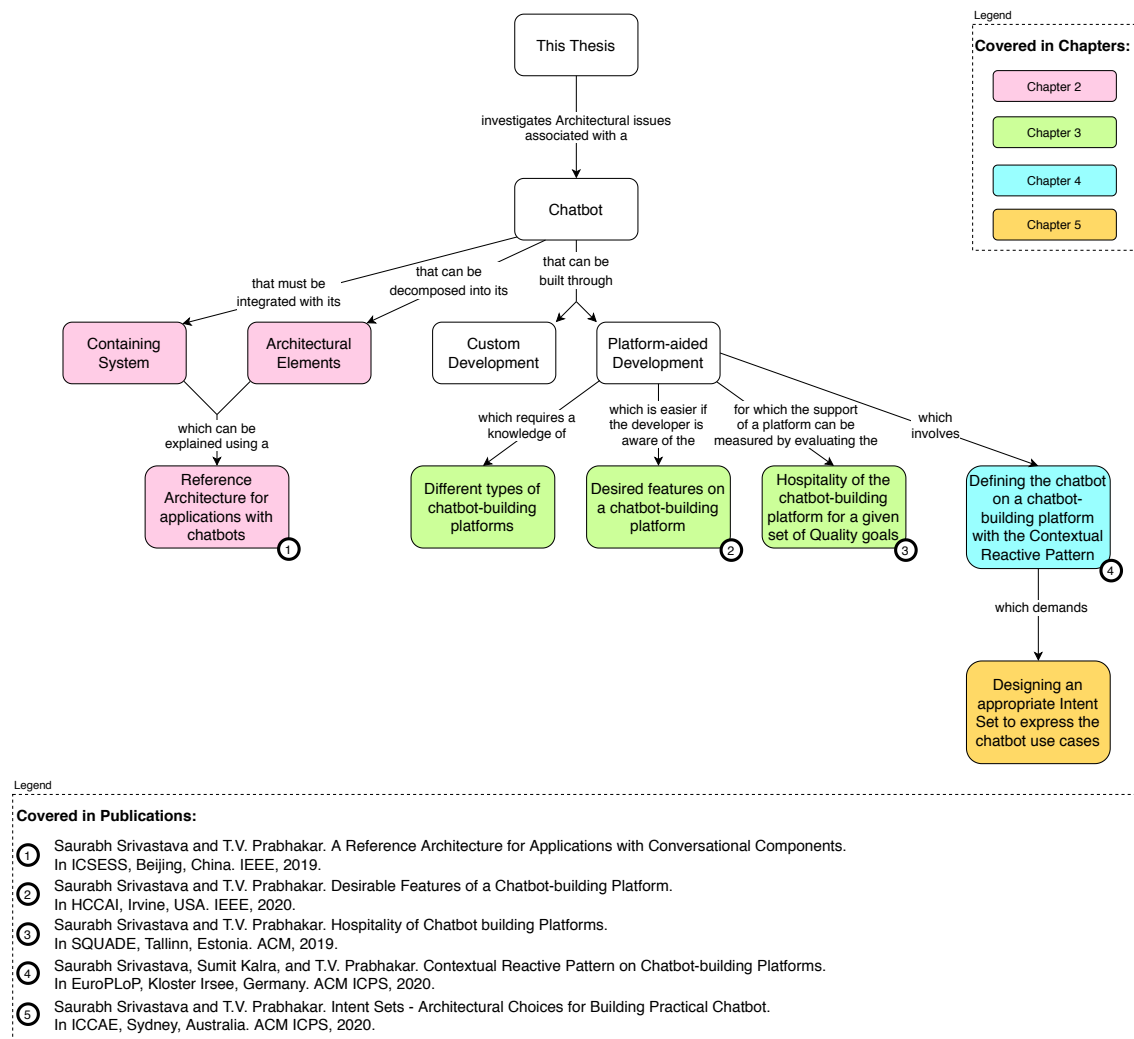


Figure 1.3: Organisation of the thesis along with major contributions

Chapter 2

Chatbot and its Environment

This chapter digs deeper into the first step in the building process of a chatbot. We refer to the system which employs the chatbot as a component, as the *Containing system* throughout this chapter. We analyse the following issues related to building chatbots in this chapter:

- What are the major elements of a chatbot, that should be implemented, or, used as a component or service?
- What is the architecture of a typical Containing system? How does it affect the design of the chatbot and vice versa?
- Are there any design guidelines to build an end-to-end application having the conversational capabilities?

This chapter is organised as follows; Section [2.1](#) presents the major elements of a typical chatbot. Section [2.2](#) discusses the architecture of the Containing system, in particular, those that are relevant for the chatbot. Section [2.3](#) presents a Reference Architecture for an end-to-end application containing a conversational interface. We also show examples of some Concrete Architectures based on the presented Reference Architecture, with the assumption that a platform was used to aid chatbot development. Section [2.4](#) presents some related works which can be pursued for further reading. Finally, we summarise the chapter in Section [2.5](#).

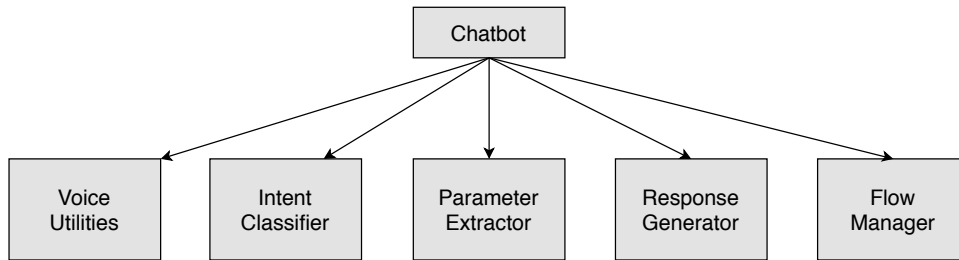


Figure 2.1: The Logical View of a chatbot

2.1 Architecture of the Chatbot

We now present the major elements that a chatbot component contains. Figure 2.1 presents an overview of these elements, along with some of the relevant components in the Containing system. The knowledge of these elements is essential if the chatbot is to be built through a custom development process. Even though, most of the platforms “blackbox” some of these elements, this knowledge provides a solid background which can be helpful in tweaking exposed meta-parameters on a chatbot-building platform. We now discuss each of these elements in brief.

2.1.1 Voice Utils

Voice Utilities (or Voice Utils) are relevant for chatbots which expose a Voice Endpoint for communication. There are two types of Voice Utilities that such a chatbot may require. The first kind are the Speech-to-Text Utils. It is essential because all the available libraries and services for performing NLP tasks expect the input to be in textual format (i.e. they cannot work directly over an audio file). The second kind includes the Text-to-Speech Utils. These are required if the chatbot also provides the response in the form of an audio file, where the response is spoken in a human-like voice. The set of libraries or services used for one are usually different from those used for the other since they involve a different set of challenges.

For the Speech-to-Text systems, the major hurdle is the inherent variations in speech signals, when produced by different human beings [57]. The variability has given rise two approaches for building these models - customised models for individual speakers and generic models that attempt to cover a wide range of speeches [58]. In either case, handling variations for the same words for different regions (e.g.

British pronunciation vs American pronunciation of the same word) is a challenge. Similarly, there are specific issues associated with the Text-to-Speech systems as well [59]. For example, a common problem includes managing prosody [60], especially in tone languages like Mandarin [61].

2.1.2 Intent Classifier

In order to process a particular query, the chatbot must first “understand” the type of query. For example, for an e-commerce chatbot, the query could be about buying a product, returning a product or feedback for the purchase process. The background processing of the query will be different for different types, such as initiating a purchase or recording a response in the feedback database. This step is known as Intent Classification, where each query that the chatbot receives must be classified under one of the pre-defined categories. In general, this process is not definitive, i.e. usually any algorithm that classifies a piece of text into a category, cannot say so with a probability of 1. The common approach is to consider Intent which has the highest “confidence score” - a number between 0 and 1 - representing the confidence of the algorithm that the text is associated with the Intent.

Most of the modern-day efforts at Intent Classification are based on deep learning techniques (e.g. as shown in [62], [63] and [64]). Usually, chatbot-building platforms provide Intent Classifiers as “blackboxed” components, i.e. they do not reveal the steps or algorithms behind the stage which do this job.

2.1.3 Parameter Extractor

Parameter extraction, also known as Slot filling, is a kind of Named entity recognition [65]. In the process, a chatbot attempts to extract any real-world data, which is required for processing the user’s query. In an e-commerce setting, an example of a Parameter is the name of the product that the user wishes to buy. The Parameter Extractor, along with the Intent Classifier jointly form the Natural Language Understanding (NLU) element of a chatbot. NLU is the subset of NLP, which attempts to associate a piece of text with its meaning. Parameters, when associated with a particular Intent, are known as Slots (we discuss this in detail in Chapter 4). For every Intent, some Slots are marked as *necessary*, i.e. a query belonging to the

Intent cannot be processed without the Parameter Extractor being able to extract a value for them. Other Slots are *optional*, i.e. there is a best-effort attempt by the Parameter Extractor to extract their values if the query provides the same.

Similar to Intent Classification, deep learning techniques are also popular among researchers for Parameter Extraction (e.g. as shown in [66], [67] and [68]). It is also not uncommon to find works which perform both tasks simultaneously (e.g. [69], [70] and [71]). Most chatbot-building platforms provide a common, “blackboxed” component to provide both Intent Classification and Parameter Extraction.

2.1.4 Response Generator

In order to produce a response for the user, the chatbot has to perform some processing. This processing may involve simple tasks - such as greeting the user with a pleasantry; or, it may involve complex background processing ranging from database look-ups to executing complex business functions. In any case, a chatbot is expected to respond to a user’s query, informing her of the performed tasks. The Response Generator element of a chatbot is tasked with producing a text response for the last received query. The Response Generator is aware of how a particular query is to be *fulfilled*. The fulfilment represents the associated background process for a particular query (we discuss this in Section 2.2.1). The response for a particular query can be classified either *Static* or *Dynamic* response.

A Static response is a fixed response for a particular class of queries. An example of such a response is a short description of an e-commerce firm’s refund policy for queries like `Can I get a refund for my product?` Static responses are often tied to queries where the user needs some information which seldom changes. Another example of a Static response is a *Prompt*. A Prompt is a pre-defined question, which is sent as a response to the user if the Parameter Extractor has not provided a value for a necessary Slot. An example of a Prompt is a response - `Can you help me with the Order Number?` It is possible that for a particular scenario, the chatbot is configured with a list of Static responses, and the chatbot picks a different response from the list every time to appear slightly more humanly (e.g. the greeting phrase could be anyone of `Hello`, `Hi` and `Hey` in a round-robin fashion).

Dynamic responses require stitching a different response every time, based on

some “variables”. These variables could either be extracted from Parameter values or could be provided as a result of one or more **Events**. Events can be considered as triggers, which initiate a processing pipeline. Events are often mapped to particular Intents, meaning that they are executed when the chatbot receives a query which has been classified with the respective Intent. Usually, Events represent one or more business processes, for which, the chatbot works like an initiator. The triggering of an Event involves invoking one or more functions, either within the realm of the chatbot or outside its scope. At the end of this processing, the Response Generator receives some inputs (e.g. a JSON object from an external API). The Response Generator then uses this information to construct a response. For instance, an e-commerce chatbot may craft a response from the template:

Kudos !! Your *\$product-name* has been ordered,
and it'll reach you by *\$expected-delivery-date*

by replacing the placeholders with an extracted Parameter (the name of the product) and a value received after executing the business function for ordering products (the expected delivery date for the newly created order).

2.1.5 Flow Manager

A user query can have a varying amount of information. For example, the query could be “complete” with respect to a business request, or the chatbot may need more data to process the request. As an example, consider the query **Please order a Samsung Galaxy Note for me, and ship it to my home.** The query is complete with respect to the business process of ordering a product. Assuming that the business function for ordering products only requires a product name and an address, the above query can be processed straight away. The name of the product can be extracted from the query itself, and the pre-stored home address of the user can either be extracted via another business function or provided by the user in previous conversations. In the latter case, the value may be picked from the chatbot Context Variables or just **Contexts** in short. Contexts are a set of variables, usually key-value pairs, that the chatbot maintains to store relevant information associated with the current conversation session. They can be handy in improving the overall user experience, by storing any piece of information temporarily, meaning that the

user does not have to enter it again.

The Flow Manager manages the discourse of the current conversation. An example of how the discourse could be different, consider the query - **I want to buy a phone** - with respect to the same business process discussed above. Clearly, this query cannot be processed straight away. The user has neither picked a particular product, nor has she mentioned an address where the product should be sent. In such cases, the Parameter Extractor cannot extract the required Parameters from the query. The common path that most chatbots take at such instances can be summarised as follows:

- Store the current state of the conversation (e.g. the Contexts) as well as the intention of the user (e.g. the output of the Intent Classifier) in some temporary storage.
- Make a list of the Parameters that the user query does not provide (or the Parameter Extractor failed to extract). If any of the associated Slots are marked as *necessary*, issue Prompt responses to the user to seek values for these Slots.
- When the values associated with all the required Slots are available, restore the state of the conversation, and proceed towards fulfilling the query, as if the initial query was “complete”.

This is not the only case when the Flow Manager has to plan a detour. A chatbot may have to deal with *Digressions*. A Digression is a scenario when the user temporarily changes her intentions and then come back to the previous topic of conversation (we discuss this in detail in Chapter 3). For example, assume that the chatbot issued a Prompt to the user - **Can you tell me the name of the product you wish to buy?** - for extracting the Slot value associated with the product name. Instead of providing the name of the product, the user, it is possible that the user replies with a different query, such as **What phones do you sell?**. This is known as a Digression. The user may be focused on achieving a goal, i.e. buying a phone, but she needs some information before making that decision. Handling these scenarios is a challenging task. Nevertheless, it is under the ambit of the Flow Manager to manage these complexities.

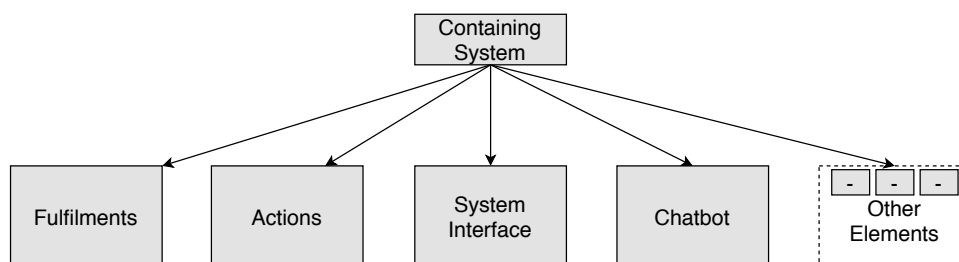


Figure 2.2: The Logical View of the chatbot's Containing system

2.2 Architecture of the Containing System

We now briefly discuss the elements of the Containing system that are relevant to the chatbot development process. Figure 2.2 provides an overview of the Containing system. The Containing system can be extremely complex. We only discuss the elements that are relevant for the chatbot development.

2.2.1 Fulfilments and Actions

Although not a part of the chatbot itself, Fulfilments and Actions are closely related to the working of the chatbot. Fulfilments are the processing endpoints into the business domain that a chatbot invokes as a part of processing a query. There are two ways to connect a chatbot with Fulfilments. The chatbot may be configured with a single Fulfilment endpoint, e.g. a URL. The chatbot invokes this URL with all the relevant information, such as Contexts, name of the detected Intent, name of any triggered Events and parsed values of Slots. The exposed endpoint is tasked with invoking relevant business functions based on the passed values. The second option is to expose multiple Fulfilment endpoints to the chatbot, associating these endpoints to specific Intents or Events. In this case, the chatbot is responsible for invoking the expected business function based on the triggered Event.

Actions represent the specific business functions that are invoked during the execution of a Fulfilment. There may not be an explicit differentiation between Fulfilments and Actions. Based on how the Containing system is designed, they may be same or different. The idea is that Actions are specific to the business domain. They may be invoked through other invocation paths as well (e.g. through a website or a command-line interface). Fulfilments are the facades which may be

created to hide the details of Actions from the chatbot. This may be necessary if the chatbot is built using a platform, and it is crucial to hide the internal business processes from the chatbot (which is developed on a different domain).

2.2.2 System Interface

Any system that interacts with users has one or more user-facing interfaces. Common examples of interfaces are Web Portals and Android Apps. Interaction channels through popular social media pages and handles on messaging platforms are also becoming increasingly common. Chatbots are often integrated with a system's existing interfaces to provide a smoother experience to the users. For example, the Lufthansa Airlines Chatbot [2] is deployed on the Messenger platform [72]. It is, therefore, possible that a chatbot has to process queries from different origins as well as different formats (voice or text) and send responses in the same fashion. Chatbots always provide a **Text Endpoint** for connection, but based on the Containing system's requirement, it may also need to expose a **Voice Endpoint**. Some platforms also allow a mixed-input endpoint, which can accept both types of inputs.

The interfaces of a system over which the chatbot is to be integrated can be a strong driving force for selection of a particular platform for building the chatbot. This is why most of the platforms provide the facility of seamless integration of the built chatbot over popular messaging platforms such as Messenger[72], Slack [73] and Telegram [74]. However, many businesses wish to integrate the built chatbot with their mobile apps, or as a chat widget [75] on their website. In such cases, the chatbot must support a generic communication mechanism (e.g. through a REST API [76]) to cater to all required interfaces of the system.

2.3 Reference Architecture for the Application

We now discuss the final contribution of this chapter. We present a Reference Architecture for an end-to-end application that has some form of conversational capability. A Reference Architecture is different from a Concrete Architecture, which represents the schematics of an actual system. On the contrary, a Reference Architecture works like a design guideline for a class of applications which share some

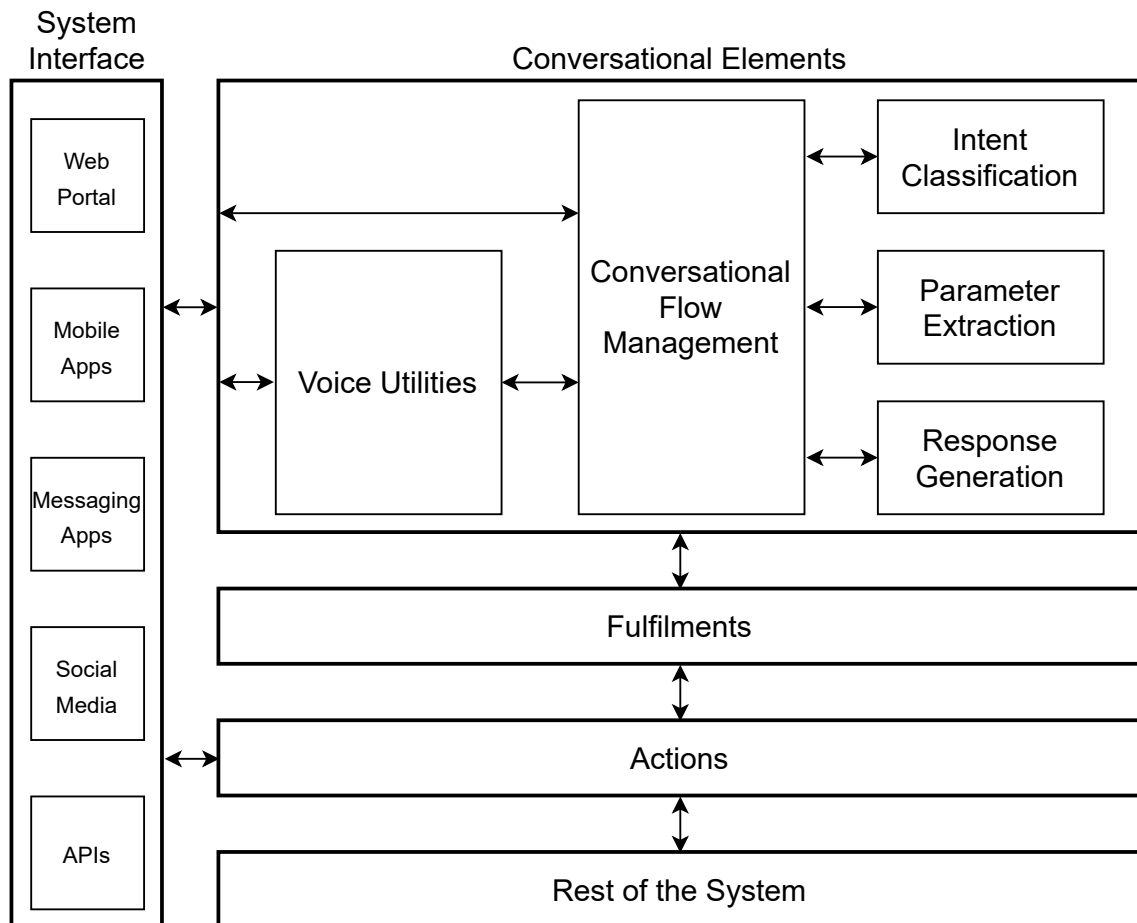


Figure 2.3: A Reference Model for the Containing system

common concerns. In our case, the common concern is to add a conversational interface to an existing or novel project, having other elements such as databases and modules implementing specific business functions. Another objective for studying and proposing Reference Architectures is to provide a set of common terminology and a shared vision to various stakeholders of the system [77].

Often, a preceding step towards coming up with a Reference Architecture is to start with a Reference Model. While a Reference Model provides an overview of the functionality that the system is expected to achieve, a Reference Architecture, maps these functionalities to system elements [78]. A fair explanation about the functionality of the chatbot, as well as its containing system has been discussed in Section 2.1 and Section 2.2. An amalgamation of the discussions in these two section

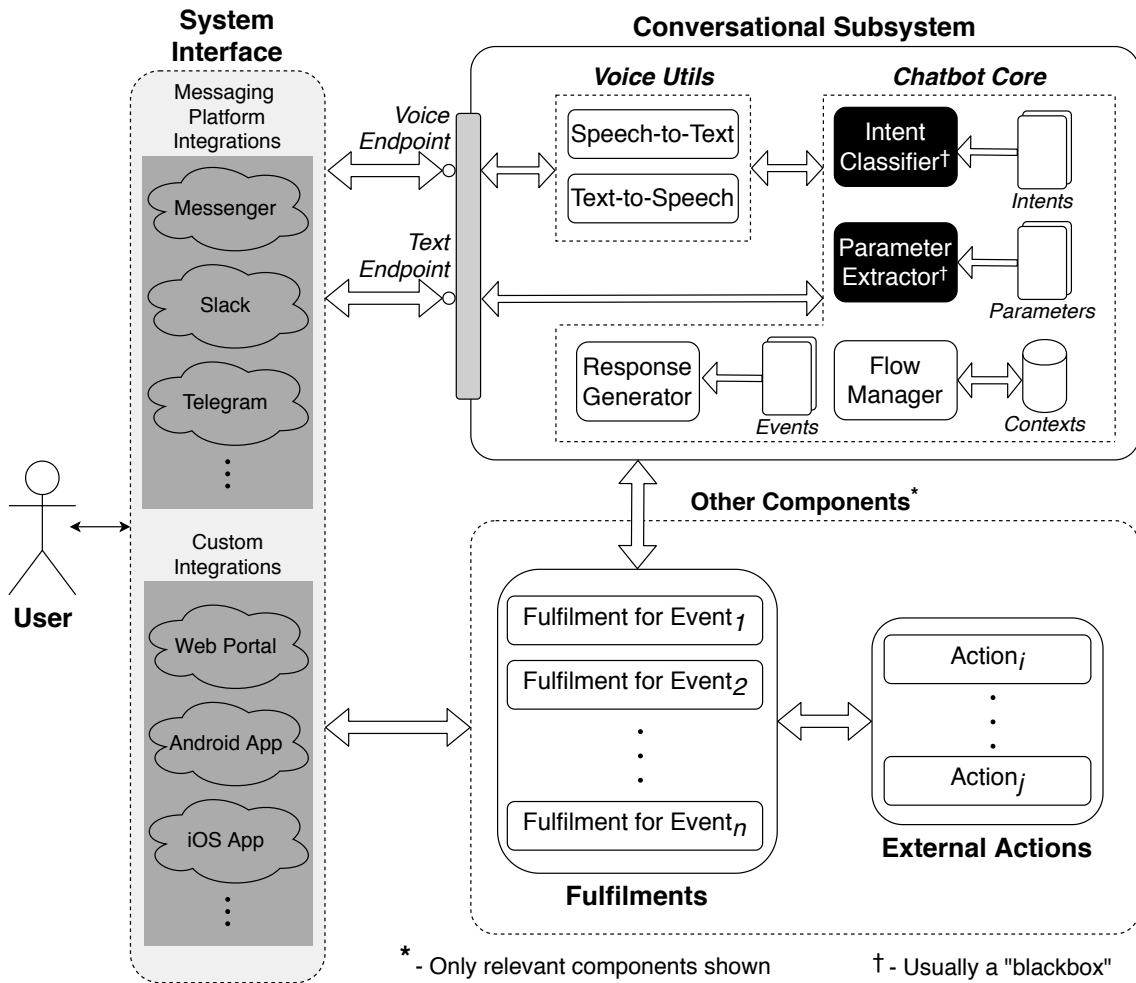


Figure 2.4: Proposed Reference Architecture for the Containing system

can be summarised using the Reference Model shown in Figure 2.3. The Reference Model shows the elements of the chatbots as well as the Containing system together in one frame, to provide the bird's eye view of the system. With this in sight, we can now move towards the proposed Reference Architecture which maps it to system elements.

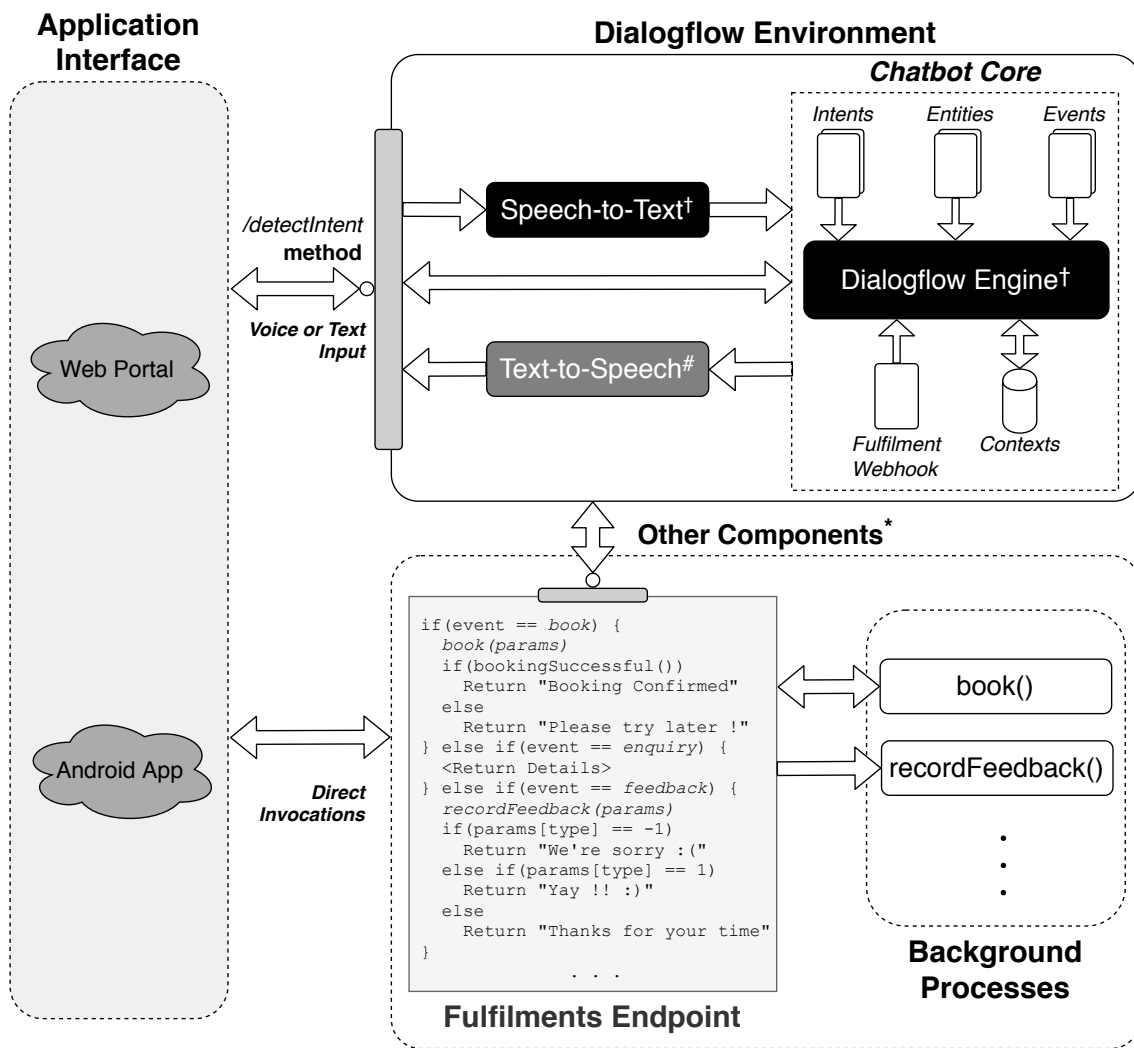
Figure 2.4 shows the proposed Reference Architecture. We have discussed the elements of the chatbots separately in Section 2.1. It must be noted that we use the term *Conversational Subsystem* in this figure to refer to the elements of the chatbot. It is because when a chatbot is integrated with a system, we view it as a subsystem of the larger environment. This Reference Architecture is designed with

an assumption - the chatbot is to be built using a platform, and not from scratch (this is why the Intent Classifier and Parameter Extractor elements are shown as “blackboxes”). The platforms that we considered here fall into a specific category that we call the *Conversation-as-a-Service* offerings. We will discuss this in detail in Section 3.2. To explain the Reference Architecture, we show some Concrete Architectures derived from it.

Consider the case of an Airline chatbot. It allows users to perform basic business operations, such as enquiring about a flight, booking a flight or providing feedback for their last journey. Let us assume that the chatbot must be integrated with two interfaces which are part of the Airlines’ IT operations - their Web Portal and their Android App. There are three candidate platforms that the company is considering to aid the chatbot development - Google Dialogflow [27], IBM Watson Assistant [21] and Amazon Lex [28]. We now present three Concrete Architectures of the Airlines’ operations environment, assuming that one of these platforms was used for building the chatbot element. These architectures are derived out the Reference Architecture shown in Figure 2.4. The three possible Concrete Architectures are shown in Figures 2.5, 2.6 and 2.7. These architectures differ from each other in multiple ways. We now provide a short description of these Concrete Architectures, highlighting major differences between them.

2.3.1 Concrete Architecture using Dialogflow

Figure 2.5 shows a possible Concrete Architecture of the system when Dialogflow is used for building the chatbot. Dialogflow provides a single, mixed-input endpoint to communicate with the chatbot, using the `detectIntent` method. A query from the user, either in text or audio format, is sent to this REST endpoint, with appropriate credentials (for authentication and authorisation). The Dialogflow Environment hosts the different elements of the chatbot, including the Intent Classifier and Parameter Extractor, which we collectively show as part of the **Dialogflow Engine**. Similar to other platforms, the details about the implementation of these elements is “blackboxed” from the developers. Dialogflow allows configuring a *single Fulfilment webhook* to connect to the external business operations, in this case, business functions like `book` or `recordFeedback`. It forwards all the required information, such



* - Only relevant components shown † - A "blackbox" # - Provided implicitly via WaveNet

Figure 2.5: Example Concrete Architecture for a Containing system when Dialogflow is used for building the Conversational subsystem

as the name of the detected Intent and extracted Parameter values to this webhook through a POST HTTP call, and expects a text response in return in a particular format. The response is then relayed back to the caller of the `detectIntent` method. The Fulfilment webhook decides what actions are to be taken for a particular user request. Here the actions are the business functions like `book` or `recordFeedback`, which are not exposed to Dialogflow.

2.3.2 Concrete Architecture using Watson Assistant

Figure 2.6 shows a possible Concrete Architecture of the system when Watson Assistant is used for building the chatbot. Chatbots built using Watson Assistant cannot accept audio input files. Watson Assistant provides a single endpoint using the `message` method, which accepts only text inputs, meaning that the Voice Utils (as discussed in Section 2.1.1) are not provided as a part of the chatbot. However, IBM provides two independent services, called IBM Speech to Text [79] and IBM Text to Speech [80] which provide methods called `speech-to-text` and `text-to-speech` respectively to fill this void. Thus, if the requirements involve supporting a voice-based interaction, there need to be one or two additional API calls. Watson Assistant provides a mechanism to invoke an IBM Cloud Function [81] as a Fulfilment for any Intent. Each Intent can invoke a different Cloud Function. They recommend writing the business logic for every operation in Cloud Functions, or, make HTTP requests to the business operations' URLs from these functions. *This Concrete Architecture is based on the platform's state before August, 2019.* Prior to this, it was not possible for a chatbot to directly invoke an external endpoint. Instead, external endpoints were only accessible to the Cloud Functions, and any relevant business operations invocations were supposed to be made via a Cloud Function only. In any case, the architecture shown in Figure 2.6 reflects a valid architecture. As shown, such an architecture can reveal the details of the business operations to the platform. It is possible add some proxy Fulfilment code between the business operations and IBM cloud functions, similar to the case with Dialogflow. However, it will add another level of redirection (chatbot to Cloud Function, Cloud Function to the proxy Fulfilment code and then the relevant business operation), meaning more delay between the arrival of a query and its response. The August 2019 releases provided the

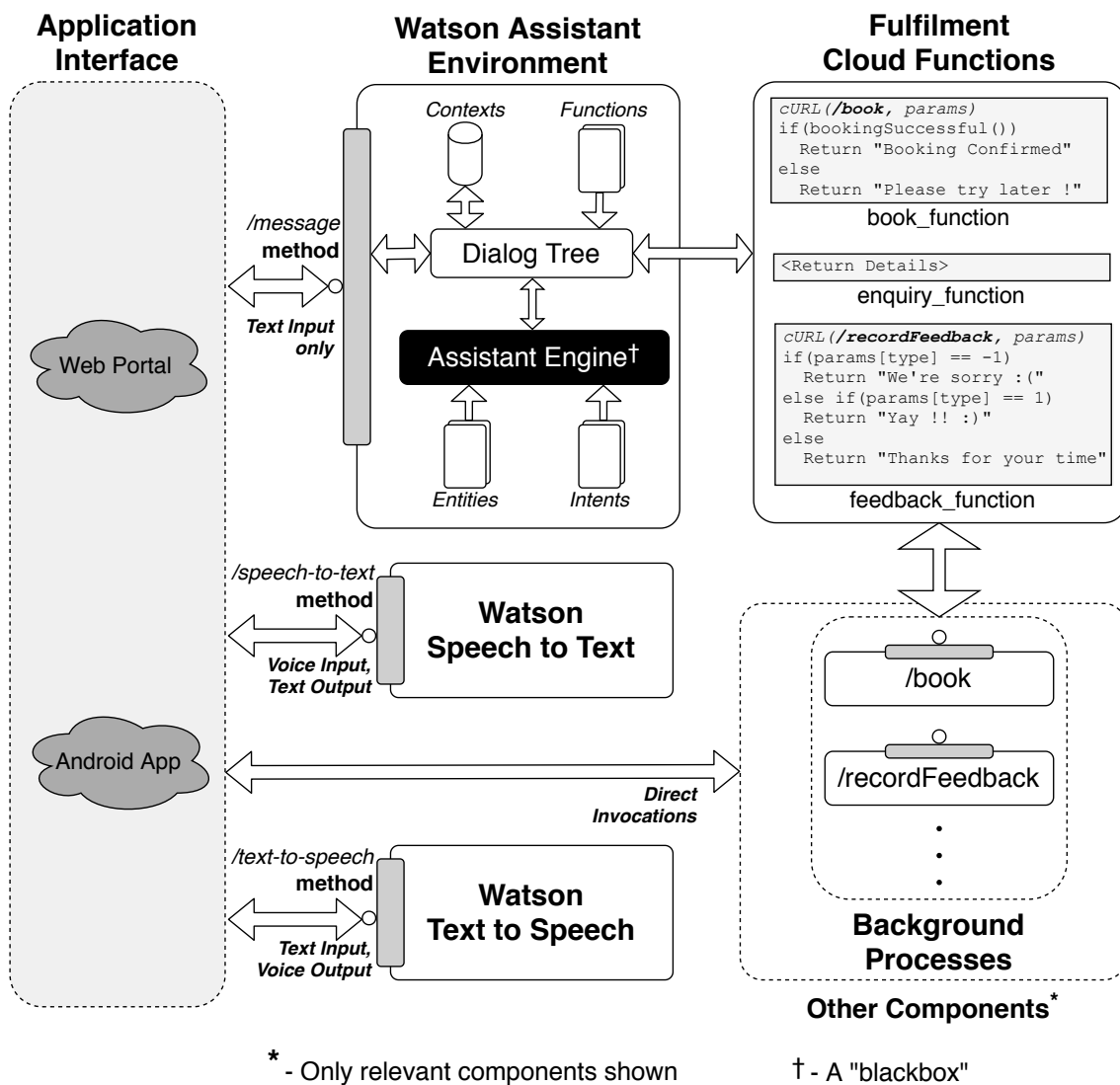


Figure 2.6: Example Concrete Architecture for a Containing system when Watson Assistant is used for building the Conversational subsystem

option of configuring a single Fulfilment webhook, similar to Dialogflow, essentially mitigating this drawback of the platform.

2.3.3 Concrete Architecture using Lex

Figure 2.7 shows a possible Concrete Architecture of the system when Lex is used for building the chatbot. Lex provides two different endpoints for interacting with the chatbot. The `postText` method, as the name suggests, is a text-only endpoint. The `postContent` method, is a mixed-input endpoint, which can take either text or audio input. The `postContent` method depends upon the HTTP headers to differentiate between the input types. Fulfilments in Lex can only be provided with the help of AWS Lambda Functions [82]. Lex does provide an additional feature - possibility of configuring a distinct Lambda Function for the sake of validation of extracted Parameters. For simple use cases, this can also be done by other means; hence we only focus on the Fulfilment Lambda Function. Lex expects that any business operations required for the chatbot are either defined in terms of Lambda Functions or be invoked through the same, as shown in Figure 2.7. Each Intent can invoke a different Lambda function. Lambda function can then invoke one or more business operations. As of writing this thesis, Lex does not provide the option of configuring an external webhook to invoke business operations. Thus, this architecture, similar to the architecture shown in Figure 2.6, reveals the details of the business operations to the chatbot platform. Similar to the discussion above, adding some proxy Fulfilment code can be a solution, albeit with time penalties.

2.4 Related Work and Further Reading

Reference Architectures have been suggested for many classes of applications (e.g. [83], [84], [85] and [86]). Attempts to elaborate on their benefits and demerits have also been made [87]. They have also been contrasted with Product Line Architectures [88]. There are articles on the Internet as well which cover general design guidelines about chatbots (e.g. [89], [90] and [91]). There are some surveys too, which compare different chatbot platforms (such as [92] and [93]). However, chatbots and the platforms which provide support for building them are still evolving.

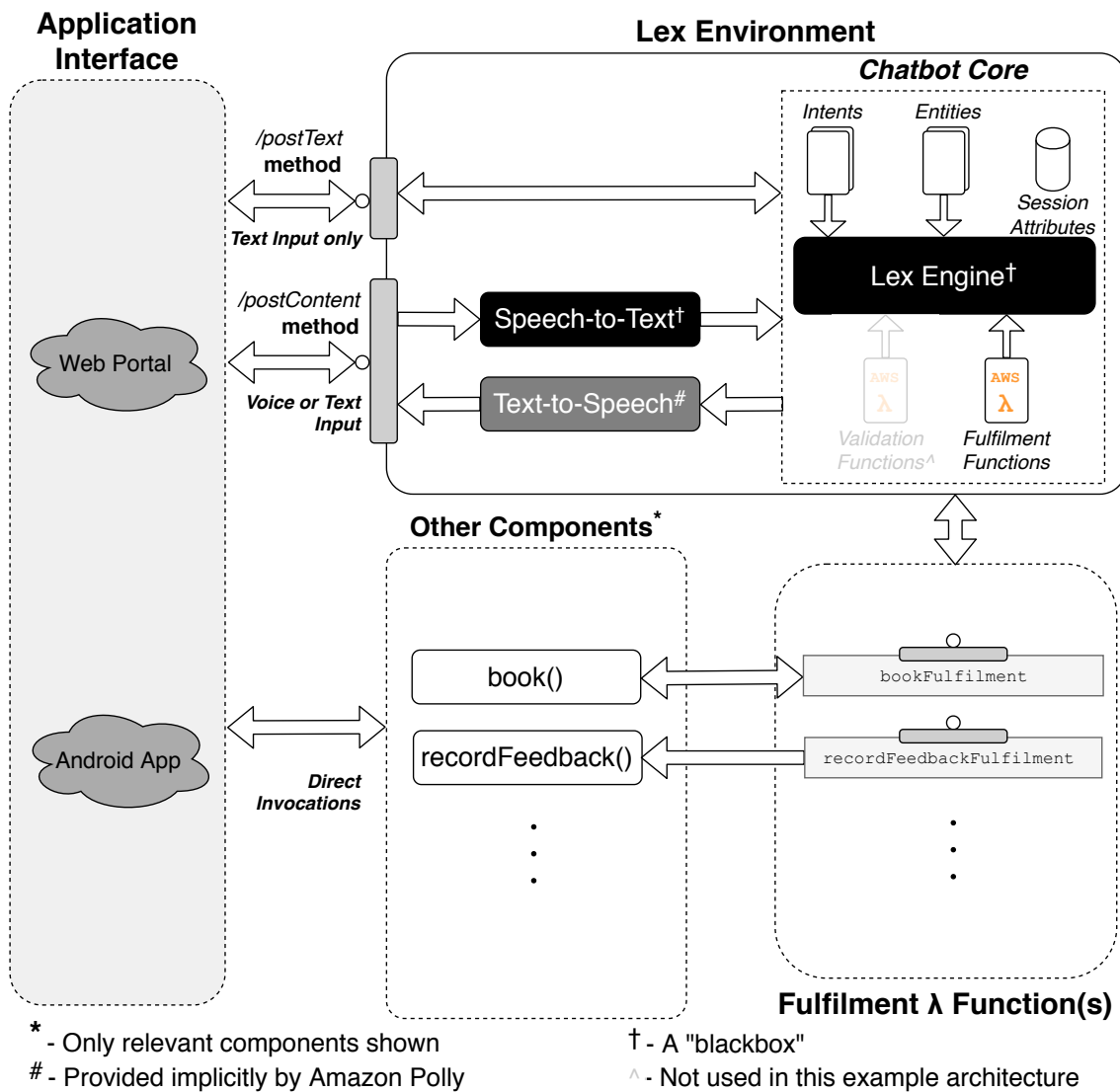


Figure 2.7: Example Concrete Architecture for a Containing system when Lex is used for building the Conversational subsystem

This means that many articles and blogs may become outdated fairly quickly. It is, therefore, important to read the current documentation and release notes on the platforms to know their current capabilities.

There are speech recognition services such as Google Cloud Speech [94], Sonix [95], Amazon Transcribe [96], Go Transcribe [97] and IBM Watson Speech to Text [79]. There are some open-source initiatives as well such as CMUSphinx [98], Kaldi [99] and Mozilla DeepSpeech [100]. Some services which can be used for generating speech are Google Text-to-Speech [101], Azure Text to Speech [102], Amazon Polly [103] and IBM Watson Text to Speech [80]. Open source alternatives include Mozilla TTS [104], CMU Flite [105] and eSpeakNG [106].

A step-by-step process of how an Intent Classifier can be made from scratch using deep learning techniques is shown in [107]. Another article which discusses the Intent Classification process in RASA is [108]. This too can be useful if the developers wish to implement the chatbot from scratch.

Building a Response Generator from scratch would mainly require three elements. First, an underlying storage mechanism, such as XML or JSON files to keep a map of Events, mapped to their respective Intents. Second, building or employing a Template Engine [109]. A list of some open-source Template Engines can be found at [110]. Third, there needs to be a mechanism in place to invoke external business logic, for example, using cURL [111].

Building a Flow Manager is a challenging problem, mostly because it has to be done on a per use case basis. Even the chatbot-building platforms provide limited support for Digressions. IBM Watson Assistant [21] uses a Dialog Tree to act like a Flow Manager. On platforms like TARS [112] or ManyChat [113], which provide a visual editor, the Flow Manager is the tree-like decision structure that the user defines. Some hints about the custom implementation of a Flow Manager can be extracted from the discussion in [114].

A good starting point to get a general overview of the chatbot-building process can be found in [115]. There are a number of articles like [116], [117] and [118] which discuss the building of simple chatbots using NLTK [119]. Articles such as [120] and [121] as well as [122] and [123] show how a chatbot can be composed with the help of RASA.

2.5 Summary

In this chapter, we started the discussions related to architecting chatbots as well as the Containing systems that house them. We started by looking at the major elements that a typical chatbot comprises. These elements include:

- A set of utilities called the Voice Utils, which convert speech to text and vice versa.
- An Intent Classifier element, which maps a given user query to one of the pre-defined types.
- A Parameter Extractor element, which extracts instances of useful information from the conversations with the user.
- A Response Generator element that invokes a processing pipeline, and relays a response for every user query.
- A Flow Manager element, responsible for the discourse management during a conversation session with the user.

We then discussed relevant elements of the Containing system's architecture. We discussed how Fulfilments are the connections that a chatbot uses to off-load the processing of queries to external business logic, and Actions are specific business operations which are part of the overall environment. We discussed how the system's interface could be made up of channels on popular communication mediums or, through custom websites and apps. The built chatbot, thus, may be required to have the ability to process inputs from different sources and different input formats.

We then presented a Reference Architecture for building an end-to-end application containing a conversational interface. We showed three possible Concrete Architectures, derived from the presented Reference Architecture, for a simple set of use cases. These architectures were drawn, keeping in mind that a platform was used to build the chatbot. We showed how using three different platforms can yield three different Concrete Architectures, which differ from each other significantly.

Chapter 3

Chatbot-building Platforms

In Chapter 2, we initiated the discussion on building chatbots. We discussed the architectural building blocks of a chatbot, as well as its Containing system. In this thesis, the focus is on building the chatbot using a platform. Therefore, in this chapter, we further our discussion of the process by attempting to answer the following two questions:

- What are the different types of chatbot-building platforms?
- What are the important features that a chatbot-building platform offers to the developers?
- How can we evaluate a set of candidate platforms for a given chatbot project?

Even though we present a perspective which is closer to a developer, throughout this chapter, we assume that the design decisions are either taken by a *Software Architect*, or taken with the Software Architect in the loop. It is because the choice to pick a platform is a design decision, which is usually made by the Software Architect.

This chapter is organised as follows; Section 3.1 provides a brief discussion on reasons which may force developers to avoid using chatbot-building platforms. This discussion is included for the sake of completeness. We otherwise focus on cases where a platform is used in the development process. Section 3.2 discusses the different types of chatbot-building platforms. Section 3.3 provides a deeper insight into the dashboards of chatbot-building platforms. Section 3.4 present a list of

desirable features that a chatbot-building platform should ideally have. Section 3.5 presents the details of the framework that can be applied to evaluate a candidate chatbot-building platform with respect to a particular set of use cases. In Section 3.6, we present some Case Studies which can provide valuable hints towards the process of choosing a platform for use. Section 3.7 presents some related works which can be pursued for further reading. Finally, we summarise the chapter in Section 3.8.

3.1 Reasons for Custom Development

Although in this thesis, we concentrate on building chatbots with the help of a platform, we briefly discuss the reasons for which custom development may be better. The developer need not build the chatbot from scratch. Instead, custom development may be done at different levels, to varying extents. There are two major reasons which can push the developer towards more custom development:

- **Privacy Concerns:**

Platforms require the developers to provide some training data to build the chatbot. The most vital data that a platform needs is a set of possible user utterances, i.e. example queries that a chatbot may receive once it is deployed (we discuss this in detail in Chapter 4). The best place to get this data would be from the transcripts of conversations between users and the Technical Support Team. However, the company may be bound to honour certain Privacy commitments, or legal frameworks (e.g. GDPR [124]). This prohibition may render the data useless unless it is anonymised or a synthetic dataset is created, which closely resembles the variations of the original data.

When a platform is used for building a chatbot, it becomes essential to find out details about the data they need, the data they store and the ownership of the data while it resides on their servers. Other details, such as the security measures they employ to avoid accidental leakage of data or thwart attacks, may also be crucial. For some domains, such as Healthcare, Privacy may have even more importance in the architecture of the chatbot [125]. To the best of our knowledge, there are no articles or work which compare Privacy

Policies of chatbot-building platforms. It may be because Privacy Policies are often drafted in legal terms, which may not be easy for everyone to read and understand. Some platforms do provide a simplified version of their Privacy Policy, but in general, there are no solutions. Appendix A contains links to Privacy Policy resources for selected platforms.

In addition to training data, another possible Privacy problem may arise if the user inadvertently reveals any personal information while conversing with the chatbot. Most of the platforms maintain logs of conversations of the chatbot. It means that the personal information of the user has entered a domain that she did not intend to (as she assumed that the chatbot operates in the realm of the company). Handling such instances require precise NLP models to detect such information at the runtime, and either replace the same with synthetic data or delete the utterance completely. Achieving this level of customisation over a platform is usually not easy. It is another strong reason to consider building the chatbot from scratch, in-house.

- **Flexibility Issues:**

The chatbot-building platforms provide features to cover common use cases. Even though some platforms allow more flexibility over others to the developers, there can be some use cases which cannot be implemented on any candidate platform. An example of such a scenario is dealing with queries that can be associated with more than one Intents. It is extremely difficult to build chatbots which can handle queries like:

```
I would like to tell you that your service is horrible,  
please refund my money immediately !!
```

The above query can be associated with two Intents - the Intent of providing feedback as well as the Intent of returning a product. It is not uncommon to receive such complex queries from users. If the chatbot is designed in a way that it can only process simpler queries, i.e. those which can be associated with at most one Intent, then it is not feasible to handle the query shown above. There can be other scenarios too, specific to a particular use case, which cannot be mapped to the definition pattern that the chatbot provides.

In addition to these issues, there may be certain constraints which can increase the developer's custom development efforts. Some of these constraints for which developers may not find any suitable candidate platform are listed as follows:

- **Supported Languages:**

Although English seems to be supported by almost all platforms, a chatbot may have to cater to people who may not be able to converse in English. The major issue with supporting multiple languages is that the NLP tasks may not be performed in the same fashion for each one of them. It means that for every supported language, the platforms may have to manage a different *language model* [126]. Although some methods have been studied for building “general language models” (e.g. [127], even large-scale deep learning efforts like GPT-3 find it hard to deal with issues pertaining to language modelling [128]. It is not straightforward to find chatbot-building platforms which support languages other than English. There are some platform comparison articles which cover the supported languages attribute as well (e.g. [129] and [22]), but usually it is not discussed in blogs and articles.

- **Pricing:**

Probably the most stringent constraint on any software project is Cost. The advent of cloud computing allowed businesses to pick a pay-as-you-go pricing model. It allows businesses to try out solutions, often free for a limited period, before making a heavy investment. Whether these prices are within the project's budget or not, differs for every case, and a detailed analysis is required to see if any candidate platform fits the requirements. Since pricing information is dynamic, the best places to look for it are recent articles on the Internet such as [130] and [131]. There are some websites and blogs which compare chatbot-building platforms with each other regularly, such as [132], [133] and [134], which also provide updated information about pricing models of popular platforms.

- **Region of Deployment:**

Some chatbots may have a hard constraint on their response time, i.e. there may be a hard limit on the maximum amount of time the chatbot can take

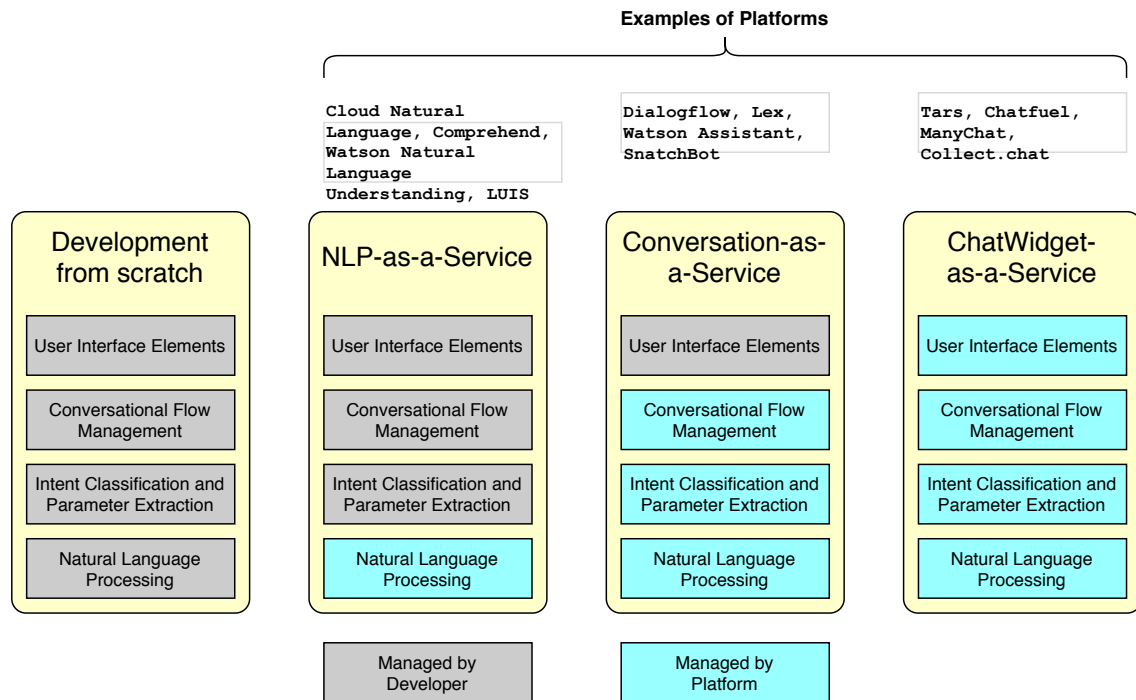


Figure 3.1: **Solution stacks offered by different Chatbot-building Platforms along with some examples**

to respond to a user query. In such cases, there may be a strong need to be able to select the geographical region where the platform deploys the chatbot. Some chatbot-building platforms (such as Watson Assistant [21] and Lex [28]) allow developers to choose a region where the built chatbot will reside. Other platforms do not provide this liberty, and the actual location of the chatbot cannot be custom picked by developers.

As mentioned earlier, we concentrate on platform-aided development in this thesis. This discussion was aimed at covering certain cases where significant custom development may be required (in addition to, or without the use of a platform).

3.2 Types of Chatbot-building Platforms

We begin the discussion about chatbot-building platforms by examining common offering stacks offered by commercial platforms. It allows us to differentiate them

with custom development, as well as with each other. Figure 3.1 shows the categories in which these offerings can be classified, along with some examples for each type. The leftmost stack is not a category, but represents a complete custom development scenario. The other stacks compare the responsibilities of the developer, in case a platform from that category is used to aid the development. The NLP tasks are the foundation for any chatbot and are thus placed at the bottom of the stack. The User Interface of the chatbot (e.g. a chat widget), placed at the top of this maturity stack, interacts with the user, hiding all the details of implementation. The two intermediate levels are chatbot-specific tasks. We now describe these three categories in brief.

3.2.1 NLP-as-a-Service Platforms

The platforms which provide NLP-as-a-Service (NLPaaS) aim to a wider audience - not necessarily those who are building chatbots. Chatbot developers may use these platforms to compose functions which perform tasks that are specific for a chatbot. For example, these platforms provide basic features such as Entity Extraction and Text Categorisation, which can be used to build the Parameter Extractor and the Intent Classifier elements of a chatbot. Usually, these NLP services have ready-to-use NLP models for multiple Natural Languages. In case these models are not helpful, some services also allow training custom NLP models. A major issue with the use of these services could be price. Each service has a different pricing model, and it may enforce cost constraints over its usage. Some of these services include Microsoft LUIS [135], Amazon Comprehend [136], Google Cloud Natural Language [137] and IBM Watson Natural Language Understanding [138].

While the use of these services can allow developers to compose a more flexible version of the chatbot, they may still not alleviate all issues pertaining to the Privacy of data. To augment the use of these services, the developers may have to look for some Privacy-preserving techniques [139], e.g. those related to Data Anonymisation [140].

In addition to cloud-based services, there are a few options which can qualify under NLPaaS; however, they are not provided in ready-to-use forms. A more apt term for these options would be NLP libraries or frameworks. Nevertheless, they

provide similar support to developers, as the NLPaaS platforms. The most popular open-source framework for building chatbots is Rasa [141]. Rasa is a dedicated open-source framework, specifically for building, and integrating chatbots with popular messaging platforms. Other, more general NLP libraries can also be used, albeit with more coding effort. The examples for such libraries include NLTK [119], spaCy [142] and Gensim [143]. The use of the TensorFlow [144] to train and use models is almost omnipresent with all AI activities today, including those involving NLP tasks.

All the libraries discussed above are in Python. A free framework for performing chatbot-specific NLP tasks in Java is the BotsCrew Bot Framework [145]. Some available options to perform core NLP tasks in Java are Apache OpenNLP [146], Apache UIMA [147] and Stanford NLP [148].

3.2.2 Conversation-as-a-Service Platforms

The Conversation-as-a-Service (CaaS) platforms are those which are dedicated towards chatbot development. Often, the providers of such offerings have their own dedicated layer of NLP services which power their abilities to build chatbots. There are two important aspects in which CaaS platforms help the developer. First, they provide support for matching queries to their Intents and extracting any Parameter values from user utterances. Second, they allow the developer to control the discourse during a conversation with the user to make the overall process seem more natural. These offerings give ample flexibility to the developers for implementing their use cases, and at the same time, reduce the effort required to do the same (we discuss how they do so in Chapter 4). It makes these platforms the ideal class to study from an architectural point-of-view, since their usage provides scope for some interesting designing decisions. Thus, in this thesis, we focus largely on chatbot development over CaaS platforms. **From here on, whenever we use the term “chatbot-building platforms”, we essentially mean “CaaS platforms”, unless otherwise specified.** Some examples are Google Dialogflow [27], IBM Watson Assistant [21], Amazon Lex [28] and SnatchBot [149].

CaaS platforms expect the developers to provide some training data, with the help of which, they build models for performing the NLP tasks such as Intent Classi-

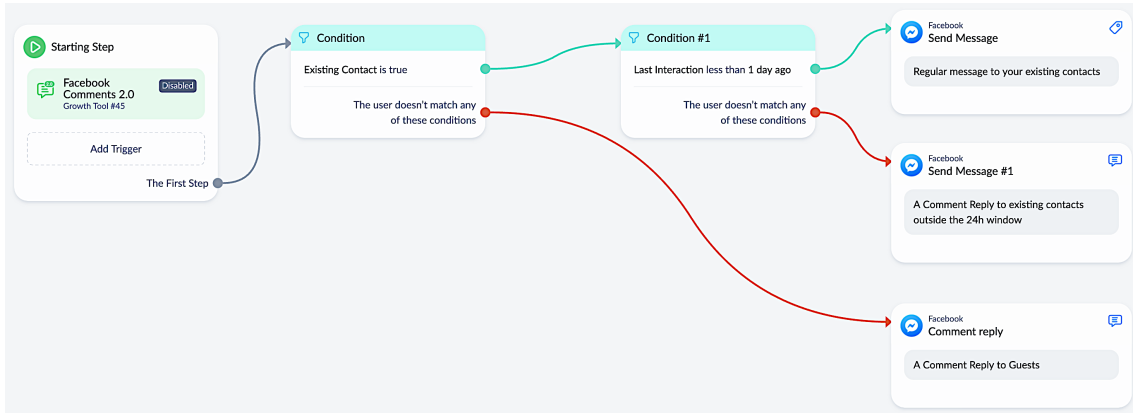


Figure 3.2: A view of the ManyChat Visual Editor, clipped from an image at [1]

fication and Parameter Extraction. These platforms provide multiple **Integrations**, through which, the built chatbot can be deployed over a wide array of mediums and communicate with downstream processes. These platforms expect that the developers define the use cases associated with the chatbot in a particular definition pattern (we discuss this in detail in Chapter 4). This pattern enforces a restriction on the capability of the built chatbot. Any use case that cannot be expressed in terms of this pattern cannot be served by the chatbot built with these platforms. While this does take away some flexibility from the hands of the developers, they save the efforts related to training and fine-tuning models for the NLP tasks. For many use cases, this trade-off is acceptable.

3.2.3 ChatWidget-as-a-Service Platforms

The platforms that we call as the ChatWidget-as-a-Service (CWaaS) platforms are often also known as “no-code” platforms for building chatbots. Almost all of them provide a visual, drag-and-drop editor for composing chatbots. It is in contrast to the CaaS platforms where the developer has to provide textual data with multiple annotations.

An example of such a visual chart from the ManyChat platform [113] is shown in Figure 3.2. ManyChat specialises in building chatbots which can be deployed on Messenger [72]. It does not mean that these platforms do not provide any mechanisms to write code. It only indicates that the amount of (and scope for) coding

on these platforms is usually restricted to simple code fragments. If we refer to the elements of a chatbot discussed in Section 2.1, these platforms usually provide a Flow Manager, a Response Generator and mechanisms to invoke Fulfillments, without any (or with very simple) Intent Classifier and Parameter Extractor elements. These platforms usually provide visual tools for creating a flowchart-like structure. The built chatbot strictly follows this chart, including nodes meant for conditions where the chatbot receives an input that it cannot handle explicitly. An example of a Messenger [72] chatbot is the Lufthansa Airlines Chatbot [2], which is shown in Figure 3.3. These platforms often build chatbots for specific mediums (such as Messenger) and harness the features of the medium to build the chatbot (such as a multiple-choice answering interface restricting or suggesting answers to a question from the chatbot).

CWaaS platforms may not be able to provide the required flexibility that a developer may wish to have. To overcome this handicap, they often offer pre-built chatbot templates for common use cases. For more sophisticated scenarios, these platforms usually provide mechanisms to use Intent Classification and Parameter Extraction facilities of a CaaS platform such as Dialogflow.

3.3 Dashboards on Chatbot-building Platforms

The dashboard of a chatbot-building platform is the canvas where chatbots are defined. In this section we contrast the dashboards of the CWaaS platforms with those on the CaaS platforms.

3.3.1 CWaaS Platforms

CWaaS platforms usually provide a drag-and-drop graphical editor to define the chatbot's "flow". With respect to our discussion related to the elements of a chatbot (Section 2.1), this editor roughly corresponds to the *Flow Manager* (Section 2.1.5). As an example, we presented a snapshot of a flow graph, built on the dashboard of the Manychat platform [113] in Figure 3.2. We now take the dashboard of another popular top-tier platform, Chatfuel [150], to provide a general overview of the dashboards of such platforms.

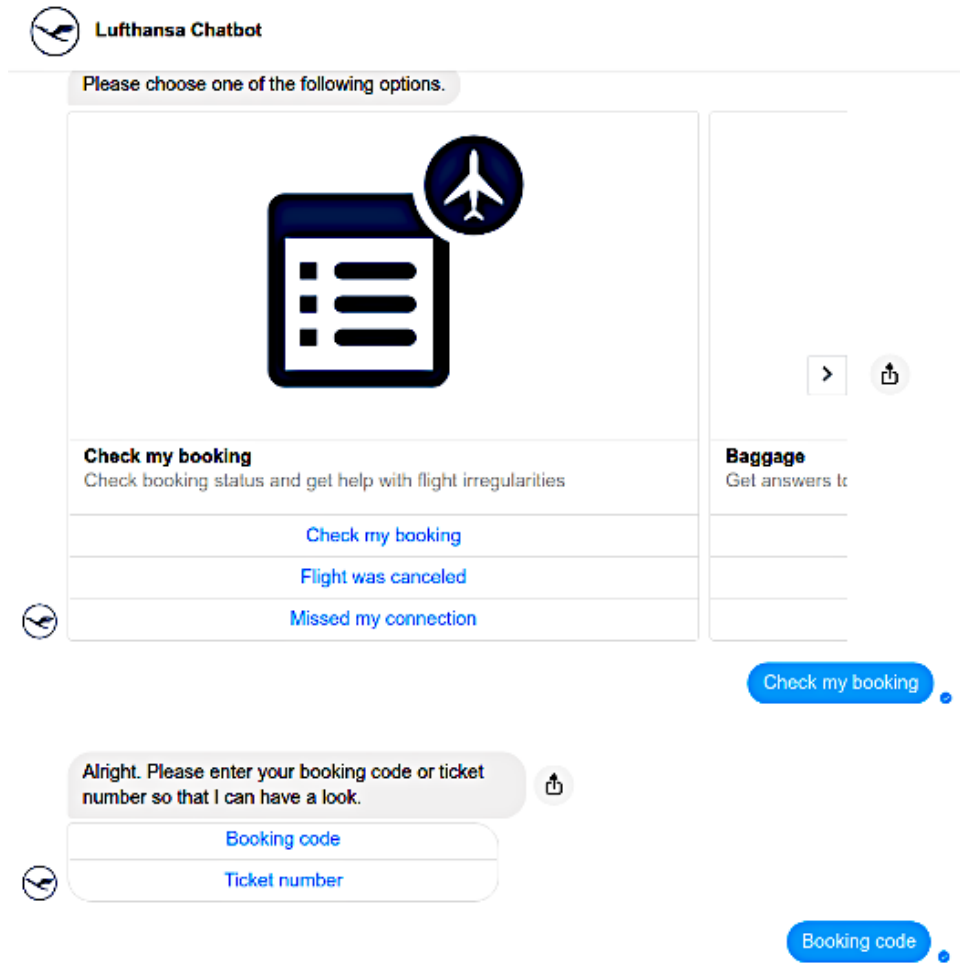
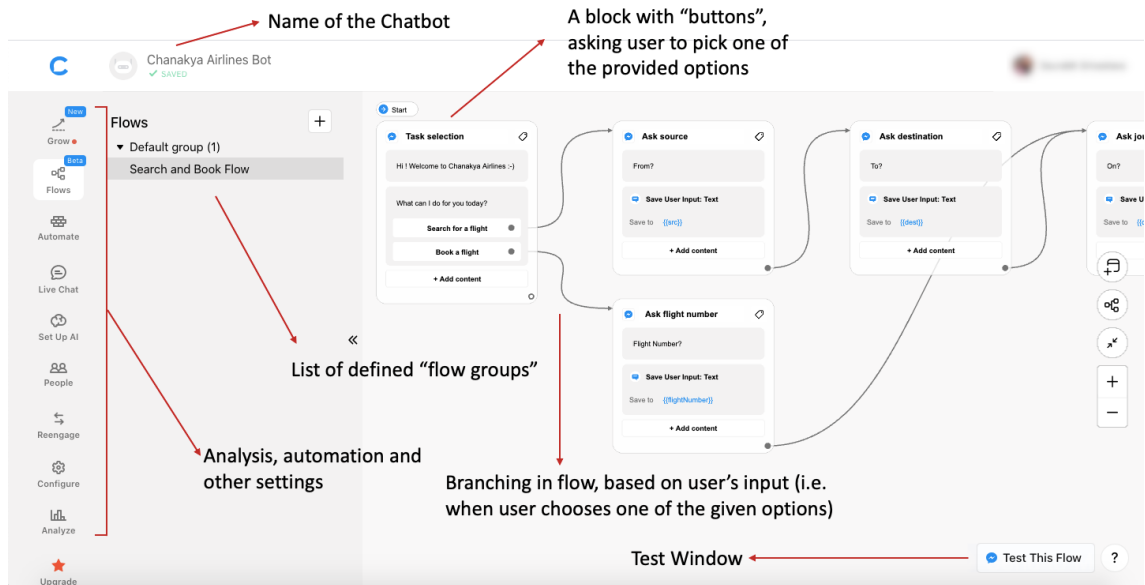
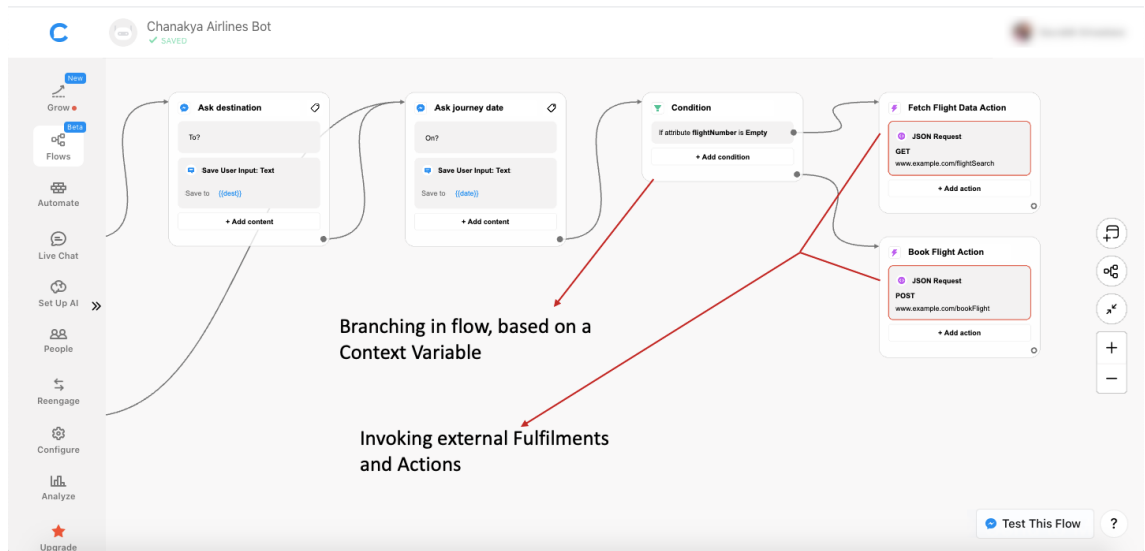


Figure 3.3: A sample interaction with Lufthansa Airlines Chatbot [2] on Messenger

Figure 3.4 shows two screenshots of the Chatfuel dashboard, with descriptions. Figure 3.4(a) shows the options available on the left pane. These include settings related to analytics and other chatbot settings on the platforms. Chatbots are defined through a group of flows. A sample flow is shown in the two screenshots. Each flow has a *starting block*. It is where the chatbot enters the flow. From that point, based on the defined flow, the chatbot can take any of the branches described in the flow. Branches in the flow can happen because of multiple reasons. It could be because the user clicked on one of the given buttons (as shown in the example of Lufthansa Airlines chatbot in Figure 3.3), or, it could be based on the value of some Context Variables, as shown in Figure 3.4(b). Figure 3.4(b) also shows two *Action*



(a) The starting (block on the top-left) and consequent blocks of the flow



(b) A block with conditional branching and two nodes to invoke external logic (on the right)

Figure 3.4: Screenshots from the Chatfuel dashboard

blocks. On Chatfuel, an Action block can perform some basic processing, such as setting or unsetting values of some Context Variables or forwarding the request to an external Fulfilment source.

We summarise the common highlights associated with the dashboards of CWaaS platforms as below:

- The CWaaS platforms usually provide a drag-and-drop editor to define states and transitions in the conversation.
- The CWaaS platforms often provide mechanisms to either restrict or suggest user inputs, by providing buttons which the user can (or must) click to move forward in the conversation.
- Since buttons are a part of chatbot's front-end, CWaaS platforms usually have higher coupling with specific mediums. For example, both Manychat and Chatfuel are meant to build chatbots for the Messenger platform only.
- Most CWaaS platforms either have limited or no support for NLP tasks. For that, they usually allow integrations with other platforms for NLP support. For example, both Manychat and Chatfuel provide mechanisms to connect the chatbots with Dialogflow [151] [152].
- The CWaaS platforms usually compensate for their lack of NLP support (and hence support for limited chatbot use cases) by providing other features, such as better analytics and insights. It is possible because they are tightly coupled with the mediums, and hence, can get a lot of data related to the users, directly from the underlying medium.
- The CWaaS platforms, do provide mechanisms to invoke external processing pipelines through REST API calls [76].

The CWaaS platforms usually pick higher *Usability* with low *Flexibility* as their primary tenets. Although CWaaS platforms are trying to catch-up on the Flexibility part - such as the newly introduced features for AI on Chatfuel [153] - they are still limited in their capability. From an architectural perspective, they do not provide enough decision-making instances in the chatbot-building journey to study.

3.3.2 CaaS Platforms

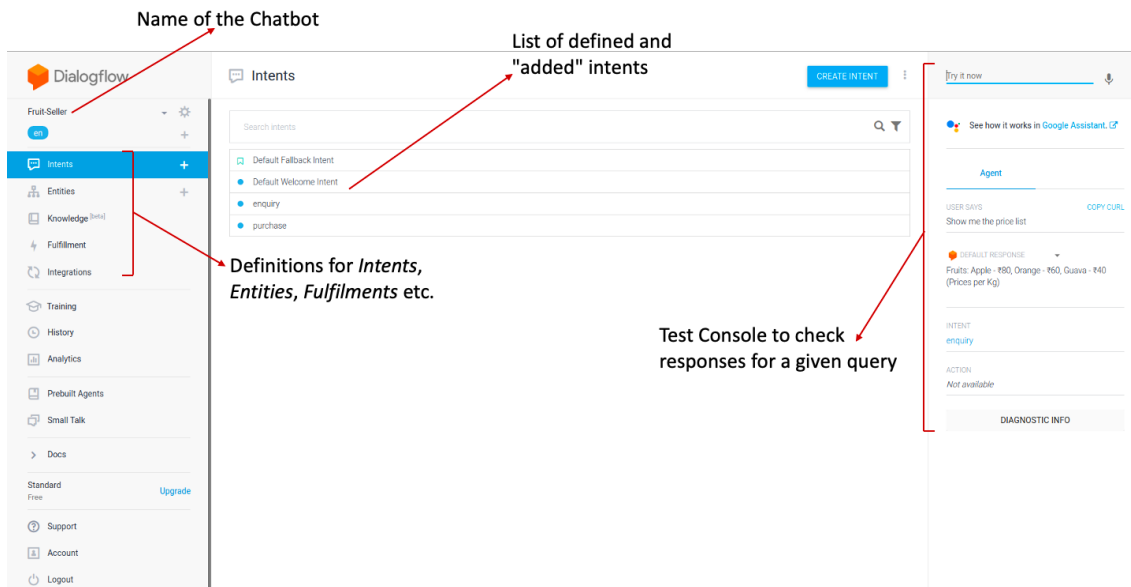
CaaS platforms require the definition of multiple elements, which the platform combines together to build a working chatbot. The two most common elements that a developer has to define on a CaaS platform are Intents and Entities. In order to discuss the dashboards of these platforms, we pick two example platforms - Google Dialogflow [27] and IBM Watson Assistant [21].

Some screenshots of the Dialogflow platform’s dashboard are shown in Figure 3.5. As shown in Figure 3.5(a), the dashboard is different as compared to that of Chatfuel. Instead of a visual editor, the dashboard shows the list of Intents associated with the chatbot. On the left, there are links to manage the other elements of the chatbot, such as Entities and Fulfilments. The right side of the dashboard hosts a Test Console, where queries can be tested for their response. Figure 3.5(b) shows the definition page of an Intent, whereas Figure 3.5(c) shows the definition page of an Entity. We will discuss these details in Section 4.2.

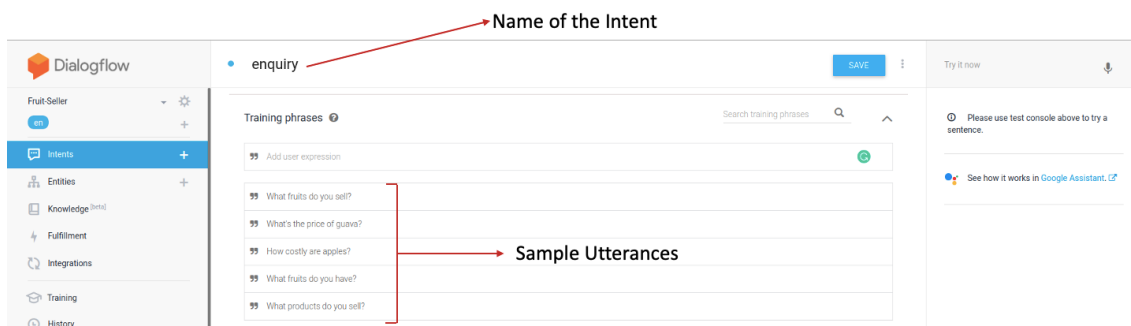
Figure 3.6 shows two screenshots of the dashboard of Watson Assistant. They show an interesting discourse management tool - a Dialog Tree for managing the flow of Conversations. The Dialog Tree acts like an explicit Flow Manager, where developers can define fairly complex flow scenarios. Figure 3.6(a) shows the nodes in the tree. Nodes are evaluated from top to bottom, and parent to child. Each node can represent some Flow Logic. An example of how the flow logic can be supplied is shown in Figure 3.6(b). In essence, the developers can define a condition, on which the node is “triggered” during evaluation. The condition could be the detection of a particular Intent in the last user input or any other boolean expression based on Context variables. In the next step, the developer can define some Context variables to hold any needed Slot values. These variables can then be used to process a Response. Finally, the developers can use a feature similar to *Goto statements* [154], to take the processing to any other node in the tree.

We summarise the common highlights associated with the dashboards of CaaS platforms as below:

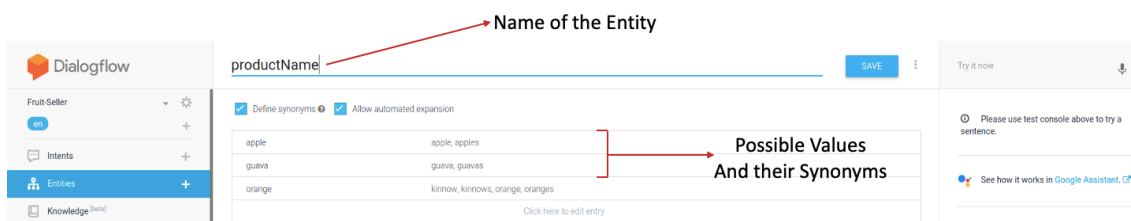
- The CaaS platform dashboards provide features to define Intents and Entities, often including the ability to tag instances of specific Entities in the user utterances associated with an Intent.



(a) A general overview of the dashboard

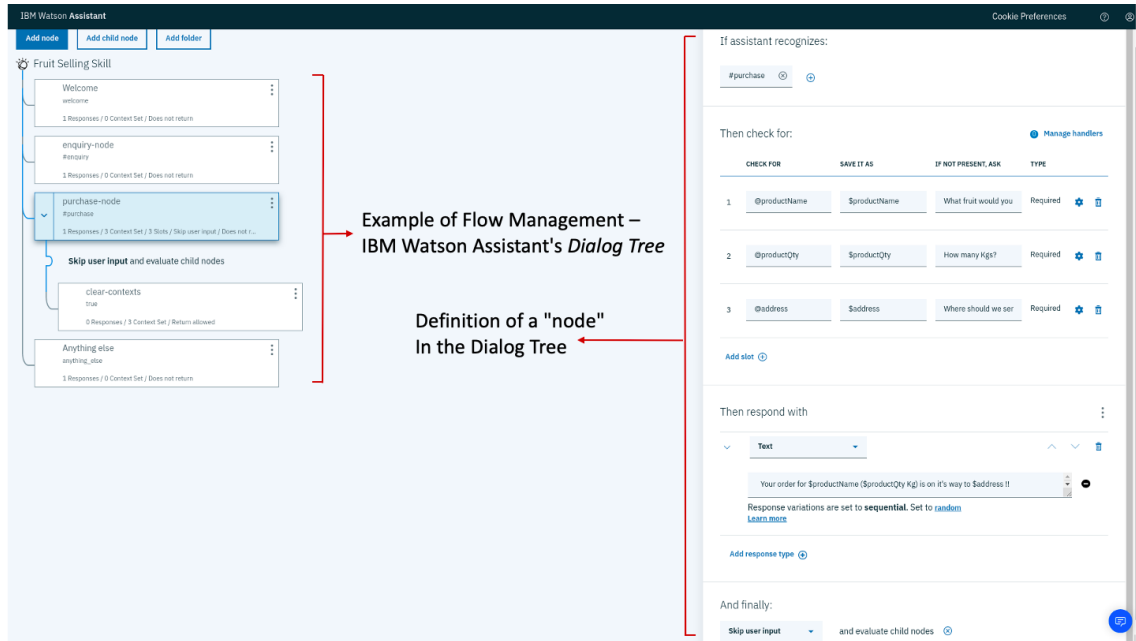


(b) Definition page for an Intent



(c) Definition page for an Entity

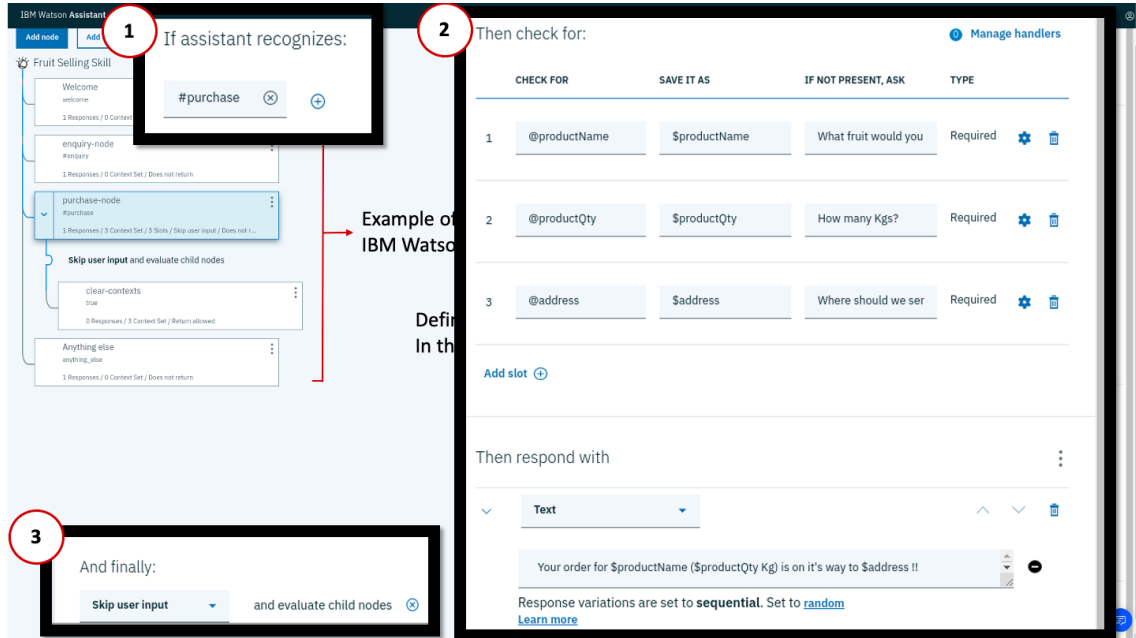
Figure 3.5: Screenshots from the Dialogflow dashboard



Example of Flow Management – IBM Watson Assistant's *Dialog Tree*

Definition of a "node" In the Dialog Tree

(a) The node structure; the nodes are evaluated from top to bottom



(b) An example of how flow is controlled

Figure 3.6: Screenshots of the Watson Assistant Dialog Tree

- While the CaaS platforms do provide easier integrations with a vast number of solutions, they are usually more generic in nature, i.e. they are not centred towards specific mediums.
- The CaaS platforms which do not have an explicit Flow Manager, usually have other features that can help in Flow Management. For example, Dialogflow offers a feature called *Follow-up Intents* [155], which in collaboration with the *Input and Output Contexts* [156] can provide support for some elementary Flow Management [157].
- The CaaS platforms, in addition to mechanisms for invoking external processing pipelines, often provide either inline coding support (such as Dialogflow’s inline Node.js editor [158]) or support on allied coding environments (such as IBM Cloud Functions [81] on Watson Assistant and AWS Lambda Functions [82] on Lex).

The CaaS platforms aim to provide more *Flexibility* to the developers for defining a variety of use cases. They provide out-of-the-box NLP features, specific to the process of building chatbots, and allow developers to use them to compose their chatbots. From an architectural perspective, they present several design choices. We will discuss one such choice in great detail in Chapter 5, called Intent Sets.

3.4 Desirable Features of a Chatbot-building Platform

We now present a hierarchical list of features that a chatbot-building platform should ideally expose, in order to cater to a wide range of chatbot projects. Although these features are fairly close to implementation (and hence, mostly affect the developers), their knowledge can be helpful for the Software Architect in analysing platforms. The Software Architect can also involve the developers in the decision-making process (we discuss examples of this scenario in Section 3.5). Figure 3.7 shows the five top-level categories of these features. Each category is either divided into sub-categories or contain a list of desirable features. As mentioned before, we assume

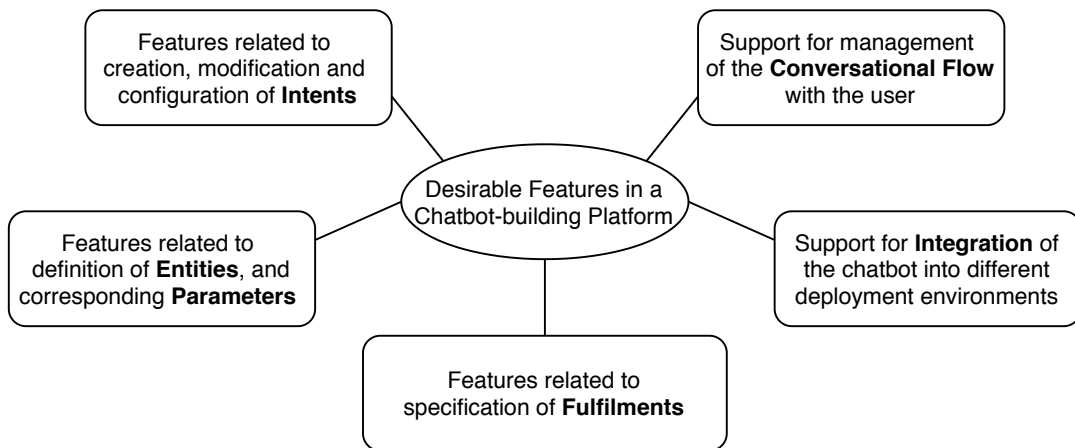


Figure 3.7: Top-level categories of desired platform features

that the platforms in consideration are the CaaS platforms. The features are based on the discussions in Section 2.1 and Section 2.3.

3.4.1 Features for Intents Management

The features in this category are associated with creation, deletion and modification of Intents on the platform. Table 3.1 shows these features. We discuss them in brief:

- **Create Intents:**

The platform should provide mechanisms to create one or more Intents that the chatbot has to serve. In particular, it should offer the following:

- Provide mechanisms to *associate Parameters with the Intent*. For example, if the developer wishes to define an Intent called `order query` to cater to any product ordering queries in an e-commerce environment, she should be able to associate Parameters like `Product Name` and `Shipping Address` with the Intent to capture this information from the user query.
- Provide mechanisms to *associate example user utterances with the Intent*. For example, for the `order query` Intent, probable examples could be: `Please order a Samsung Galaxy Note to 221B, Baker Street`.
- Provide mechanisms to *tag locations of Parameter values in user utterances for the Intent*. For example, the above example could also be

Table 3.1: Intents Management Features

Desired Platform Features for Intents Management
<p><i>Create Intents</i></p> <ul style="list-style-type: none"> Add Parameters Add training examples Tag Parameter occurrences in training examples Map Parameter occurrences to specific values
<p><i>Update Intents</i></p> <ul style="list-style-type: none"> Provide new training examples Edit existing training examples Remove existing training examples
<p><i>Delete Intents</i></p>
<p><i>Default Intent</i></p> <ul style="list-style-type: none"> Provide default response Set a minimum confidence threshold to trigger any Intent Provide counterexamples – to trigger Default Intent explicitly

provided as: Please order a *Product Name* to *Shipping Address*.

- Provide mechanisms to *map instances of Parameter values, in example user utterances for the Intent*. For example, in the example above, the developer should be able to tag instances for `Product Name` and `Shipping Address`, like: Please order a Samsung Galaxy Note:Product Name to 221B, Baker Street:Shipping Address.

Note that the last feature is a “superset” of the feature before, i.e. if the platform allows mapping of Parameter values, it implicitly provides the feature of tagging the location of the Parameter values as well. However, a platform can provide flexibility to the user to pick either of the two options - i.e. provide specific values (such as `Samsung Galaxy Note`) and map them to the respective Parameter (such as `Product Name`), or just provide a placeholder in the example to show the location of the Parameter. Watson Assistant [21] offers both the features. Dialogflow [27] offers the latter, but not the former. Lex [28] offers the former, but not the latter.

- **Update Intents:**

The platform should provide mechanisms to update the details associated with defined Intents for the chatbot. In particular, it should offer the following:

- Provide mechanisms to *add additional example user utterances* associated with an Intent.
- Provide mechanisms to *modify existing user utterances* associated with an Intent.
- Provide mechanisms to *remove one or more existing utterances* associated with an Intent.

- **Delete Intents:**

The platform should provide mechanisms to delete one or more Intents defined for the chatbot.

- **Default Intent:**

Default Intents play an important role in the working of a chatbot. We discussed the uncertainties associated with the chatbots in Section 1.2. One of the uncertainties that the chatbot has to handle is the imperfection of the Intent Classifier element. It is possible that the user utterance cannot be mapped with high confidence to any of the defines Intents. It could be because the user asked something that is out of the scope of the chatbot, or that the chatbot was not trained with a similar user utterance. In any case, platforms should provide a safety net to catch these instances and invoke a backup plan. A query is classified under the Default Intent if it cannot be associated with any defined Intent of the chatbot. A platform should provide the following features to the developer, to make the best use of the Default Intent:

- Provide mechanisms to *configure one or more default responses* for the chatbot. For example, the default response for a chatbot could be:
Sorry, can you rephrase your query?
- Provide mechanisms to *configure the minimum confidence threshold for triggering a non-Default Intent* for the chatbot. For example, if this threshold is set to 0.8, then in case the chatbot cannot associate the

Table 3.2: Entities Management Features

Desired Platform Features for Entities Management
<p><i>Create Parameters</i></p> <ul style="list-style-type: none"> Provide possible values for the Parameter Provide synonymous occurrences for provided values Allow the extension of possible values list automatically Disallow extension of possible values list automatically
<p><i>Update Parameters</i></p> <ul style="list-style-type: none"> Add more possible values for the parameter Edit a provided value Delete synonymous occurrences Remove existing values for the parameter
<p><i>Delete Parameters</i></p>

query with any non-Default Intent with at least a probability of 80%, then the Default Intent is triggered.

- Provide mechanisms to *associate example user utterances with the Default Intent* of the chatbot. These examples are essentially counter-examples for the chatbot, to signify that these set of queries are not meant to be served by the chatbot. For example, if a chatbot in an e-commerce environment is not supposed to process returns, then the following query could be configured as a counter-example: **I want to return the phone I bought from you !!**

3.4.2 Features for Entities Management

The features in this category are associated with creation, deletion and modification of Entities on the platform. Entity is another term used to denote Parameters. Table 3.2 shows these features. We discuss them in brief:

- **Create Parameters:**

The platform should provide mechanisms to create one or more Parameters that the chatbot will encounter in user queries. In particular, it should offer the following:

- Provide mechanisms to *define possible values of the Parameter*. For example, possible values for a Parameter called **Customer Age** could be defined as: **less than 18**, **18-35**, **36-50** and **more than 50**.
- Provide mechanisms to *define synonyms for any possible value of the Parameter*. For example, for the **less than 18** value of the **Customer Age** Parameter, the developer could define a synonym called **minor**.
- Allow the developer to *choose that the possible values of the Parameter can be extended by the platform automatically*, if a relevant query containing the same is encountered. For example, if a Parameter called **Fruit Name** is defined to have values **apples**, **oranges** and **grapes**, the developer could configure the platform to add, say **pears** to the list, if it encounters a query where the term “pears” was found at a location usually associated with the **Fruit Name**, e.g.: **I would like to buy 2 Kgs of fresh pears**.
- Allow the developer to *restrict the possible values of the Parameter to the explicitly defined values only*, even if the platform encounters a likely new value of the Parameter. For the above example, in such a scenario, the platform will ignore the value “pears” and behave as if no value has been specified for the **Fruit Name** Parameter.

Note that the last two features are “opposites” of each other. For some Parameters, the developer may wish to have the former feature, while for others, the developers may want the latter. An example of the former case is, as mentioned above, the name of fruits. The developer would like that the platform keeps on adding new fruit names to the list, since it may be too difficult to provide the names of all possible fruits explicitly. An example of the latter is the type of shipping - normal or express. Since the available types of shipping are limited in number (and can be defined explicitly), the developer would like to restrict its possible values to the defined set only.

- **Update Parameters:**

The platform should provide mechanisms to update the details associated with defined Parameters for the chatbot. In particular, it should offer the following:

- Provide mechanisms to *add more possible values* of a Parameter.
- Provide mechanisms to *modify any possible values* of a Parameter.
- Provide mechanisms to *remove one or more synonyms for any possible value* of a Parameter.
- Provide mechanisms to *remove one or more possible values* of a Parameter.

- **Delete Parameters:**

The platform should provide mechanisms to delete one or more Parameters defined for the chatbot.

3.4.3 Features for Defining Fulfilments

The features in this category are associated with the management of Fulfilments. Table 3.3 shows these features. We discuss them in brief:

- **Slot filling:**

The platform should provide mechanisms to fill values of *necessary* Slots associated with the Intents of the chatbot. As mentioned in Section 2.1.3, a Slot represents the mapping between an Intent and a Parameter. Values for necessary Slots are required for processing queries of the respective Intent. In particular, the platform should offer the following:

- Provide *a special type of response to the user, called a prompt*, seeking values for a particular Slot. For example, if the user has not provided value for a Slot called **Shipping Address**, the prompt may look like:
I need a **Shipping Address** to process your order !!
- Show the user *a set of fixed options* to pick a value for a particular Slot. For example, for fetching the value for a Slot called **Shipping Type**, the platform may provide the user two options - **Normal** and **Express** - and the user is expected to pick anyone out of these two options.

- **Static Responses:**

The platform should be able to provide a static response for queries associated with certain Intents. In particular, the platform should offer the following:

Table 3.3: Fulfilments Management Features

Desired Platform Features for Managing Fulfilments
<p><i>Slot filling</i></p> <p>Provide prompts to receive inputs from the user during conversation, for filling a required Slot</p> <p>Show options (one or more buttons for the user to click, out of a limited set)</p>
<p><i>Provide a static response for some queries</i></p> <p>Textual Response</p> <p>Multimedia Response (Image/Audio/Video etc.)</p>
<p><i>Provide a dynamically crafted response for some queries</i></p> <p>Allow the response to have placeholders for Slot values and Context variables</p> <p>Allow the response to have placeholders for secondary information, e.g. Intent classification confidence or computed values</p> <p>Dynamic response based on variable values (e.g. using conditional operators or if-else ladders)</p>
<p><i>Trigger external events</i></p>
<p><i>Control Slot values interpretation</i></p> <p>Get the supplied value</p> <p>Get the reference value</p>

- Provide mechanisms to configure a *fixed textual response* for queries associated with any particular Intent. For example, for an Intent called **greeting**, the developer may fix the response as:
Hello !! how are you doing today?
- Provide mechanisms to configure a *fixed multimedia response* for queries associated with any particular Intent. It may include a picture, a video, or any other custom response.

- **Dynamic Responses:**

The platform should be able to provide a dynamic response for queries associated with certain Intents. In particular, the platform should offer the following:

- Provide mechanisms to *craft dynamic responses with placeholders pointing to Slot values or Context variables*. For example, for the **greeting** Intent, the response could be made dynamic, by using the user's first name from the Contexts. This is possible if the platform allows responses like:
Hello *\$FirstName* !! how are you doing today?
- Provide mechanisms to *craft dynamic responses with placeholders containing computed values or any other value, not a part of the Contexts*. For example, the platform has mechanisms to build a response having a placeholder for user's age, derived from the value of a Slot associated with user's Date of Birth.
- Provide mechanisms to *craft dynamic responses based on conditions*. For example, for the above scenario, the platform has mechanisms to provide different responses, based on different age groups.

- **Trigger External Events:**

The platform should provide mechanisms to trigger any Events, which in turn, initiate a processing pipeline, external to the chatbot. The most common means to do so is providing the developers with the option to configure an external URL. The URL is invoked by the chatbot when a specific Event occurs, such as the user requesting the creation of a new order. The platforms

usually put a timeout for the call to complete, so that the chatbot does not enter an indefinite wait state.

- **Parameter Values Interpretation:**

The platform should provide options to the developers to choose which value for a particular Slot should be supplied for downstream processing. For example, for a Slot that accepts the name of a sport, the developers may define a possible value as `Football`, with the synonym `Soccer` (they are two names for the same sport). When a user supplies this value during a conversation, she may write either of the two terms. Based on how the Fulfilment logic is written, the developer may want the platform to replace the term `Soccer` with `Football`, before triggering any Events. In this case, `Football` is the *reference value*, whereas `Soccer` is the *supplied value* for the Slot. In particular, the platform should offer the following:

- Provide a mechanism to the developer to configure the chatbot to *use the reference value* of a particular Slot value, instead of the supplied value.
- Provide a mechanism to the developer to configure the chatbot to *use the supplied value* of a particular Slot value, instead of the reference value.

3.4.4 Features Related to Integrations

The features in this category are associated with the management of Integrations. As mentioned in Section 3.2.2, Integrations are the means with which the chatbot interacts with its environment. This category includes features related to integrating the chatbot with the Application’s user interface, linking it to the Application’s business backend and accessing it through programmable means such as APIs. Table 3.4 shows these features. We discuss them in brief:

- **Invoking public URLs for Fulfilment:**

This is the most common means provided by platforms to support the *Trigger External Events* feature discussed in Section 3.4.3. It means allowing developers to configure external URLs to invoke on the triggering of some Event. Usually, the platforms only support POST calls [159]. This option potentially

Table 3.4: Integrations Management Features

Desired Platform Features for Managing Integrations
<i>Send requests to a public URL and expect a response back</i>
<i>Send request to a restricted source (a URL with domain restrictions or a function that can be invoked internally by the platform) and expect a response back</i>
<i>Interfaces to integrate with existing platforms (e.g. Facebook Messenger, Slack, Telegram etc.)</i>
<i>API to create, modify, train, use and delete the Chatbot (e.g. REST APIs)</i>

allows integration of the chatbot with any arbitrary system at the back for processing queries. The only requirement from the backend operation is to expose a publicly accessible URI.

- **Invoking restricted, secure URLs for Fulfilment:**

The major issue with invoking a public URL is security. For example, if the public URL points to a business API of the company, it may require authentication. Platforms usually allow configuring authentication information as well along with the URL, such as username and password pairs or API keys and secrets. Nevertheless, authentication complicates the process flow, and can even slow it down significantly. Some platforms allow the specification of business logic on a restricted, secure location, within the ambit of the platform's domain. The common example for such sources are IBM Cloud Function [81] and AWS Lambda Functions [82]. The advantage of these sources is that they usually have faster response time, and use implicit authentication mechanisms, not requiring the developers to worry about passwords or API secrets.

- **Integrating with popular communication mediums:**

Assisting developers with connecting the chatbot to a popular communication medium is often touted by the chatbot-building platforms as a significant advantage over their peers. Some platforms even restrict themselves to building chatbots that are tailor-made for specific communication mediums like Messenger [72] or Slack [73]. Nevertheless, providing integration support for as many communication mediums as possible is a critical point of competition between chatbot-building platforms today.

Table 3.5: Conversational Flow Management Features

Desired Platform Features for Managing Conversational Flow
<i>Pass contextual information from the currently triggered Intent to any subsequently triggered Intents in the conversation flow</i>
<i>Handle follow-ups (e.g. Confirmations such as Yes or No, Next or Previous etc.)</i>
<i>Handle digressions</i>
<i>Process multiple-Intents queries</i>

- **API access:**

While most of the platforms provide a dashboard to the developers to create, train, use and delete the chatbots, it is an added advantage if the same can be done via programmable means. Some platforms provide an API suite for these purposes. The most common API access is provided through a set of REST API endpoints.

3.4.5 Features for Managing Conversational Flow

The features in this category are associated with the discourse management during an interaction session with the user. In simpler terms, the features in this category are essential for managing the state of a conversation with the user, and responding accordingly. Table 3.5 shows these features. We discuss them in brief:

- **Storing and Retrieving Contexts during a Conversation:**

The platform should provide mechanisms to store and retrieve values of Context variables during the processing of a query. This feature is crucial for provide the feature of crafting *Dynamic Responses*, as discussed in Section 3.4.3. Context variables or Contexts in short, as discussed in Section 2.1.5, provides a kind of temporary storage for a particular conversational session with the user. This feature allows information to flow from one query-response cycle to another. For example, consider the conversational session shown in Figure 3.8. Here, the chatbot uses the value of the Slots containing the product's name and shipping address provided by the user in previous responses, to place an order and craft a response dynamically containing the details. For this, the

U1	User	I want to buy a Samsung Galaxy Note
C1	Chatbot	Understood, you want to buy one Samsung Galaxy Note.
C2	Chatbot	Would you like Normal shipping or Express shipping?
U2	User	Normal
C3	Chatbot	Understood, Normal shipping it is.
C4	Chatbot	Can you provide me the Address for shipping?
U3	User	221B, Baker Street
C5	Chatbot	Ok, shall I place the order?
U4	User	Please make it Express shipping
C6	Chatbot	Alright, I've changed the shipping to Express.
C7	Chatbot	Ok, shall I place the order?
U5	User	Yes
C8	Chatbot	Great !! A Samsung Galaxy Note coming to 221B, Baker Street via Express shipping soon.

Figure 3.8: A sample conversational session between a chatbot and a user

platforms should provide a mechanism to store Slot values as well as any other information that the chatbot requires, as Context variables.

- **Follow-ups in a Conversation:**

The platform should provide a mechanism to provide simple follow-up prompts to the user, such as those asking for some kind of confirmation. Figure 3.8 shows how the chatbot seeks approval from the user, twice, before going ahead with the placement of the order. Another possible follow-up prompts could be the user asking to go to the next task in the sequence (such as playing the next track in a playlist). Platforms usually provide this feature by offering a set of specific Intents to seek such inputs from the user.

- **Digressions:**

A digression, as discussed in Section 2.1.5, is a scenario when the user temporarily switches the topic of conversation and returns. An example of digression can be seen in Figure 3.8. In C5, the chatbot essentially asks a Follow-up question for confirmation. The user, instead of answering in “Yes” or “No”,

instead talks about changing the shipping speed in U4. The chatbot understands this digression, modifies the shipping type accordingly, and comes back to the same question in C7. A platform should provide mechanisms to the developer to handle such scenarios.

- **Multiple-Intents queries:**

Discussed as a Flexibility issue in Section 3.1, providing mechanisms for handling complex user queries which can be associated with more than one Intent, is not easy. It is an area where most of the chatbot-building platforms fail. On some platforms (like Watson Assistant [21]), it may be possible to do so with some jugglery with the available features (for instance, as discussed in [160]). However, in general, the support for this feature is still not widely available in chatbot-building platforms.

3.5 Hospitality of a Chatbot-building Platform

In this section, we discuss the evaluation of a few candidate platforms, against a given set of use cases. The assumption is that the Software Architect can come up with a set of Quality goals to achieve. We use an architectural framework, called the *Hospitality framework*. The framework is specifically useful for evaluating the usefulness of one or more platforms. The Software Architect can apply the framework to pick one platform out of a small set of candidate platforms, by critically examining the use cases that the chatbot has to serve.

3.5.1 Understanding Hospitality

From a pure Software Architectural perspective, chatbots are just another kind of software component. They too have inherent *Quality issues*, similar to any other component. What differentiates them from the “conventional” software components is the embedded “uncertainty” involved in their operation as discussed in Section 1.2. The Hospitality framework is a general framework, which can be applied on a set of platforms, to evaluate their support towards achieving quality in an application.

The term *Hospitality* has been defined in the context of cloud platforms before in [161]. The authors defined the term as “the support provided by the underlying

cloud platform towards building quality applications”. We generalise this definition of Hospitality as following:

Hospitality of a platform is defined by the native support of the platform towards achieving any quality goals. The goal could be realising an architectural tactic or exhibiting a Quality Attribute.

Hospitality is a Quality Attribute (QA) of a platform (not an application). This attribute can be discussed with respect to specific quality goals. For example, a platform might be “more hospitable” towards the *Interoperability* QA, whereas it may be “less hospitable” towards the *Security* QA. Any application built or deployed over the platform, with certain Quality requirements, would require lesser efforts to achieve its Quality goals if the platform has higher Hospitality for the same. Hospitality can also be analysed at the level of an *Architectural Tactic* or Tactic in short [162]. For example, a platform that provides seamless integration between modules of an application may be considered more hospitable towards the *Split Module* Tactic for the *Modifiability* QA. In this thesis, we apply the concept of Hospitality to chatbot-building platforms. In Section 3.4, we discussed desirable features that a chatbot-building platform should offer. In this section, we show how the features provided by a platform, can be linked to their support towards achieving Quality goals associated with the chatbot. While our focus always remains around the Quality issues with the chatbot, certain issues may also require analysing its relationship with the Containing system.

We start by tailoring the definition of Hospitality for chatbot-building platforms as following:

The Hospitality of a chatbot-building platform can be defined as the support provided by the platform towards building and deploying a chatbot component with expected Quality.

The overall process of evaluating the Hospitality of a platform is summarised below:

1. The Software Architect analyses the use cases that the chatbot has to serve. Besides, she may also consider some of the Constraints related to the project.
2. The Software Architect prepares a list of Quality goals that the chatbot has to achieve, by examining the use cases and Constraints.

3. The Software Architect maps the Quality goals to a set of QAs. These QAs must be in focus throughout the development, integration and operation of the chatbot.
4. For each QA, the Software Architect examines the Software Architecture Body of Knowledge, to find out relevant Architectural Tactics associated with the QA. The Architect may also come up with new Tactics to suit the scenario.
5. The Software Architect should map each Tactic to a list of desirable features, that the platform should expose.
6. For each candidate platform, the Software Architect examines the availability of the desired features.
7. Hospitality can now be calculated for a particular Tactic or a particular QA by assigning different weights to the features, and calculating an index using the weighted sum approach. We call these values the *Hospitality Index* of the platform towards the specific QA or Tactic.

To remove any confusions, we summarise the three contexts in which we will use the term “Hospitality” before we move ahead:

- *Hospitality* is a QA. It is a QA of a “platform”, not the application built on top of it. Hospitality is always associated with another QA or Tactic. To put simply, it is the contribution of a platform, through its offered features, towards achieving a Quality goal.
- *Hospitality Index* is the metric to measure Hospitality of a particular platform towards a specific Quality goal. It only makes sense, when it is “compared” with the Hospitality Index of another platform towards the same Quality goal, with the same set of weights.
- The *Hospitality framework* is a framework which defines a phase-by-phase description for calculating Hospitality Indices for one or more platforms towards one or more Quality goals.

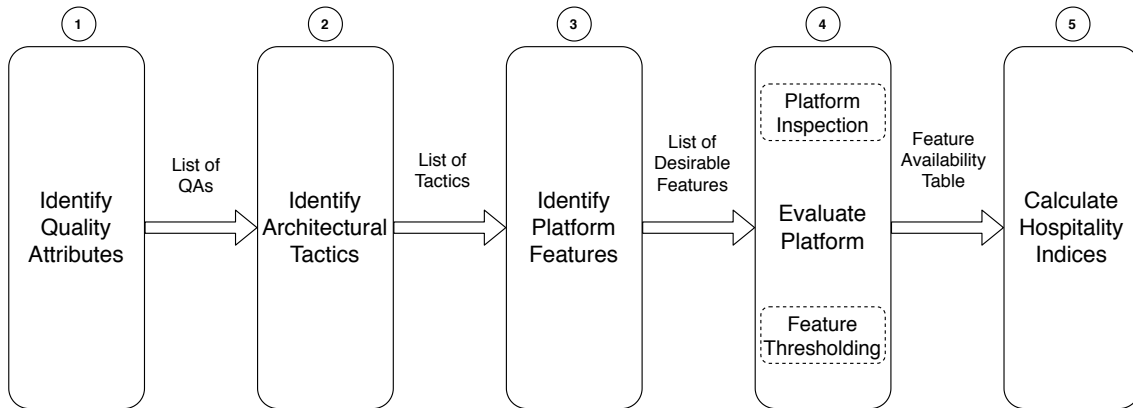


Figure 3.9: Different phases in application of the Hospitality Framework

3.5.2 Using Hospitality

Figure 3.9 shows the phases of the Hospitality framework. We describe these phases in brief. We present a case study in Section 3.6.1 to show the application of these steps over for a sample chatbot project.

- **Phase 1 - Identify Quality Attributes:**

In the first phase, the Software Architect analyses the user requirements. The requirements are usually divided into the categories of *functional* and *non-functional*. This categorisation may not always be crisp, and these requirements may not be independent of each other either. The art of carving out these distinctions is in the hands of the Software Architect. We do not detail this step as it involves a non-trivial process involving the application of existing Architectural knowledge and a substantial amount of experience.

- **Phase 2 - Identify Architectural Tactics:**

The output of the first phase of the framework is a set of QAs to achieve. The next step is to delve deeper for each QA and come up with a set of Tactics to meet them in practice. There are existing bodies of knowledge that attempt to catalogue these Tactics or, discuss the Architectural Styles which employ them (e.g. [163], [164], [165], [166] and [167]). Referencing such catalogues could be the first step towards coming up with a set of useful Tactics. It must be kept in mind that many times, Tactics presented in catalogues may be too abstract to apply directly for a particular use case. In such cases, a Software

Architect may come up with a set of custom Tactics also. The overall idea is to create a list of feasible, actionable points that can be implemented to achieve the QAs in the picture.

- **Phase 3 - Identify Platform Features:**

By the end of the second phase, the Software Architect has a list of actionable points, in the form of Architectural Tactics that are supposed to be baked in the chatbot. The next step is to figure out how a chatbot-building platform can ease the job of the developers and reduce the overall implementation time. This phase involves a *shallow* literature survey of the candidate chatbot platforms. The idea is to get an overview of their capabilities. Gathering opinions of developers with prior chatbot-building experience (if there are any in the team) can also be helpful in this phase. These platforms are still in initial stages, undergoing rapid changes to incorporate new features. Thus, the best resources for this phase are technical blogs (e.g. [168], [92] and [93]). There are also dedicated websites, where practitioners building chatbots, share their experiences (e.g. [169] and [170]). Browsing through these resources can be a good exercise to understand the general capabilities of the candidate platforms.

Although these resources may provide the latest updates, and discuss the state-of-the-art features in the arena, they almost always have inherent biases in favour or against specific platforms. Caution must be taken to make sure that these biases are not transferred in the process, affecting objectivity. The Software Architect, therefore, should refrain from studying a platform in greater detail at this step. A shallow reading to absorb the essence is enough. After surveying these platforms, the Software Architect can come up with a list of desirable features expected from a chatbot-building platform. The lists presented in Section 3.4 can also be helpful at this stage. These features facilitate the implementation of one or more Tactics that the Software Architect had listed down at the end of the previous phase.

- **Phase 4 - Evaluate Platforms:**

In the previous phase, a list of desirable features was produced. In the best-case scenario, there may be one or more platforms, which offer all the features in the above list. However, in most cases, platforms would differ in their feature-

Feature	Ability to externalise response generation
Platform	Lex
Status	Limited to AWS Lambda Functions [82]
Criteria	The platform should allow direct invocation of business logic present at a remote location, accessible via a webhook.
Decision	“✗” (Not available)
Reason	An external webhook can be invoked via an HTTP call from a Lambda function (e.g. using cURL [111]), however, it cannot be called directly. This implies additional, undesirable overhead.

(a)

Feature	Ability to provide default values for slots
Platform	Watson Assistant
Status	Cannot be set at either Parameter, or Intent level
Criteria	The platform should allow setting of default values for certain parameters, and use them for response generation instead of prompting the user.
Decision	“✓” (Available)
Reason	Watson Assistant provides a tree-like flow graph to process custom business logic. Default values for certain parameters can be set in ancestor nodes, and response can be processed in descendant nodes.

(b)

Figure 3.10: Examples of *Feature Cards* for two different features and platforms

set, and this decision may not be straightforward. In this phase, the Software Architect inspects the platforms for the presence or absence of a feature. In case, a feature is only partially available, or available but its effectiveness is not up to the satisfaction level, the feature requires some more introspection. Thus, this phase has two activities, which run in parallel - Platform Inspection and Feature Thresholding.

- **Platform Inspection:** The major objective of this phase is to put a “✓” or a “✗” against every feature in the list of desired features, for every candidate platform, indicating “available” or “not available” respectively. At this stage, a deeper analysis of these platforms is required. It involves referring to documentation and tutorials, going through developer forums and even using out-of-the-box techniques such as checking posts on trou-

bleshooting websites like **Stack Overflow** [171] for common user queries related to the platform. It must be noted that the idea should not be reading tutorials or documentation pages serially. Instead, the goal is to dig inside to find if a feature is available or not.

- **Feature Thresholding:** The decision to put a “✓” or a “✗” against all the features in the list may not always be boolean in nature. It is possible that the feature is provided with some limitations. In such cases, the issue that the Software Architect has to resolve is whether the support of the feature is helpful for the project or not? We suggest creating *Feature Cards* for such cases, similar to those shown in Figure 3.10. The Software Architect can then pass these cards to other stakeholders of the project, such as developers and testers, to collect their opinions. The Software Architect can make a final call after analysing all the cards.

- **Phase 5 - Calculate Hospitality Indices:**

It brings us to the final phase of the framework, where the Hospitality indices are calculated. As mentioned in Section 3.5.1, Hospitality Indices can be calculated at either the Tactic level or the QA level. To explain the process of calculation, we make use of the following notations:

- q_i : A QA in consideration, for $i=1,2,\dots,n$
- $\{T_i\}$: List of Tactics in consideration, for $i=1,2,\dots,n$; where $T_i = \{t_j\}$; for $j=1,2,\dots,r$ represent Tactics for achieving q_i
- $\{f_i^j\}$: List of all the Features associated with a Tactic, where f_i^j represents features desired by the Tactic t_j of the QA q_i
- $\{p_k\}$: List of candidate platforms in consideration; for $k = 1,2,\dots,m$
- $\{af_{ik}^j\}$: List of features marked with “✓”, meaning “available”, where af_{ik}^j represents the list of such features available in the platform p_k from the list f_i^j

Hospitality Index for a particular Tactic, H_{ik}^j , is defined as:

$$H_{ik}^j = |af_{ik}^j| / |f_i^j|$$

To calculate Hospitality Index for a specific QA, we can assign relative weights to the Tactics associated with it. Let

$TW_i = \{tw_j\}$; where tw_j represent the relative weight of Tactic t_j from list T_i , associated with q_i .

The Hospitality Index for platform p_k , for the QA q_i , represented as H_{ik} , can be given by:

$$H_{ik} = \sum_{\forall t_j \in T_i} (tw_j * H_{ik}^j)$$

If we assume equal weight for all the Tactics, then H_{ik} becomes:

$$H_{ik} = \sum_{\forall t_j \in T_i} (H_{ik}^j) / |T_i|$$

Figure 3.11 shows an overview of the calculation process. There is one point worth mentioning here - the same feature may contribute towards the realisation of more than one Tactic.

We now show one possible way of using the Hospitality Indices, to rank the candidate platforms. The field of Multi-criteria Decision Analysis or MCDA[172] in short, attempts to resolve problems of this nature. One such technique from the area is called *TOPSIS* (Technique for Order of Preference by Similarity to Ideal Solution)[173]. TOPSIS requires the problem to be formulated in terms of a set of alternatives, and a set of criteria. The process described below can be used for selecting a platform from some candidates (the set of alternatives), given the Hospitality Indices for the quality attributes in consideration (the set of criteria) and relative weights for preferring one attribute over the other. We summarise the steps as following:

- Step 1: If there are m candidate platforms, and n concerned QAs, generate a **Hospitality Matrix** as:

$$HM = \{hm_{ij}\}_{m \times n}; \text{ for } i = 1..m \text{ and } j = 1..n; \text{ where } hm_{ij} = H_{ji}$$

(H_{ji} being the Hospitality Index of platform i towards QA j)

- Step 2: Normalize the **Hospitality Matrix** as:

$$R = \{r_{ij}\}_{m \times n}; \text{ where } r_{ij} = hm_{ij} / \max_{\forall i} (hm_{ij})$$

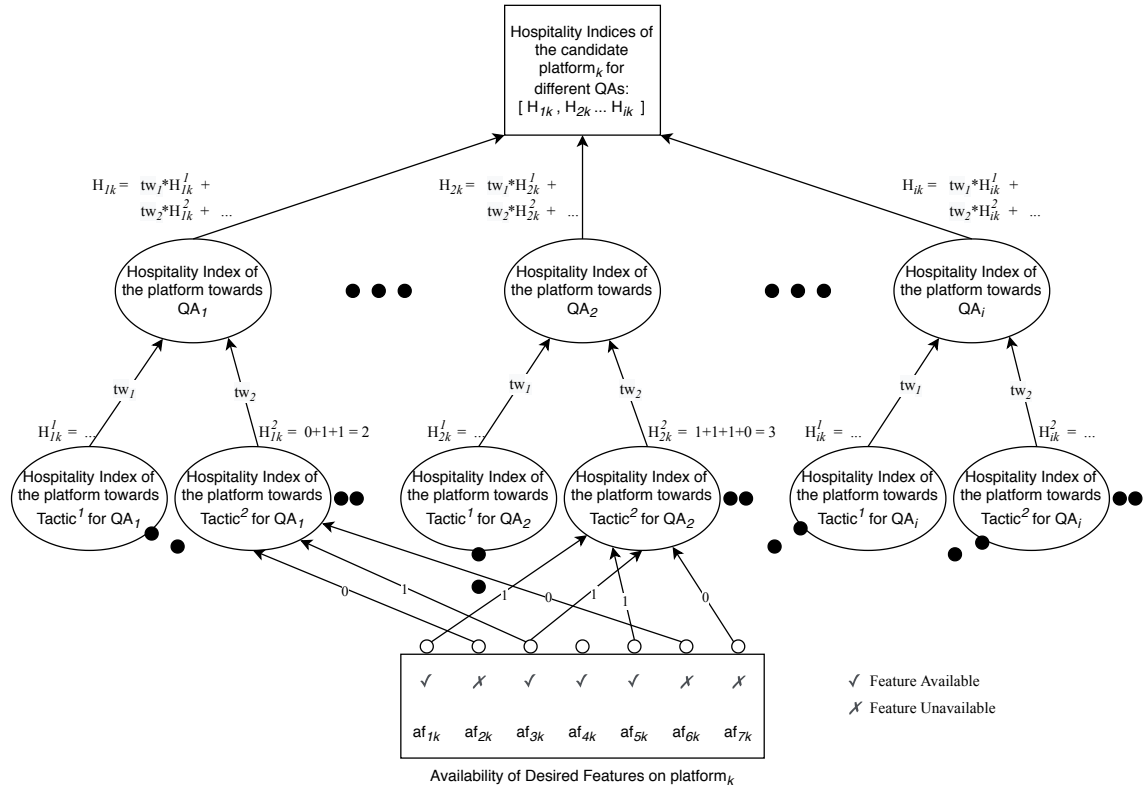


Figure 3.11: An overview of the Hospitality Indices Calculation process

- Step 3: Generate a weighted, normalized matrix, T :

$\{w_i\}$: List of weights, where, w_i is the relative weight associated with QA q_i
 $T = \{t_{ij}\}_{m \times n}$; where $t_{ij} = w_j \times r_{ij}$

The weights here are normalised. Weights can be normalised by dividing each weight by the sum of all.

- Step 4: Next, the best and the worst possible alternatives are found by:

Best possible, $A_b = \{ (\max_{\forall i=1..m}(t_{ij}) | j = 1..n) \}$

Worst possible $A_w = \{ (\min_{\forall i=1..m}(t_{ij}) | j = 1..n) \}$

- Step 5: Now, find the “distance” of candidate platform i , from the best, and the worst alternative:

$$d_{ib} = \sqrt{\sum_{\forall j} (t_{ij} - t_{bj})^2}$$

$$d_{iw} = \sqrt{\sum_{\forall j} (t_{ij} - t_{wj})^2}$$

- Step 6: Finally, find the similarity with the best possible alternative of candidate platform i as:

$$S_{ib} = d_{iw} / (d_{ib} + d_{iw})$$

The candidate platforms can now be ranked in non-decreasing order of S_{ib} values. The candidate with the highest value for S_{ib} is the recommended platform.

3.6 Case Studies

We also present two case studies that are relevant for understanding the importance of chatbot-building platforms. The first case study applies the Hospitality framework to three chatbot platforms, and show the implementation of all the phases discussed in Section 3.5.2, for a sample use case. The second case study takes the same three platforms, and report their current support for the list of desired features covered in Section 3.4.

3.6.1 Sample application of the Hospitality Framework

Consider the example of a store that sells a variety of fruits. The store plans to launch a website, as well as a mobile app for its customers. On the website, a popup window can engage users in a text chat, where users can enquire, browse and buy fruits. In the mobile app, the users can do the same, either through a text chat, or a voice chat (the user speaks her query, and response from the system is spoken back). The project involves building a chatbot for this use case.

It is clear that some components involved in these applications are relatively standard, with which software practitioners are well versed. We focus on the need to build the chatbot component, which consumes the user's inputs and produces appropriate outputs. There might be other components, too, based on different requirements.

3.6.1.1 Phase 1: Identify Quality Attributes

For the sample use case of a fruit store, let us assume that the Software Architect noted down the requirements, as shown in Figure 3.12. We focus our attention

Functional Requirements
<ul style="list-style-type: none"> - Need an app as well as a website - User could browse through fruits available in the inventory - Answer user queries about fruit prices, availability etc. - User should be able to buy fruits - If the user needs directions to the store, provide guidance - Allow typed text/spoken queries on the app
(a)
Non-functional Requirements
<ul style="list-style-type: none"> - Keep chat transcripts onsite (privacy concerns) - Chatbot component needs access from multiple locations (app/website) - Will have access to inventory, require some authentication - Need voice-to-text/text-to-voice capability, if the chatbot cannot handle it implicitly - Keep the bot simple; don't attempt to answer queries with low confidence - Add counter-examples to avoid responding to queries like "How's the weather"? - Validate the bot before deployment, check behaviour for common user utterances - Response strings and prompts might change/customised
(b)

Figure 3.12: Software Architect's notes, representing the requirements for the Fruit Store applications

mostly on the non-functional requirements, as they provide hints towards the prominent QAs associated with the application. For our purpose, we are interested only in the QAs that the chatbot component is expected to exhibit, and the issues related to its integration with the Containing system.

Suppose, the Software Architect delineates the following QAs after inspecting the requirements:

- Since the responses provided to the user, and the paraphrasing of questions could be altered; it should be easy to modify the built chatbot later, without cascading changes. It makes *Modifiability* an important aspect of the chatbot component.

- Keeping control over the chat data is important, making *Privacy* a prominent concern. Also, protecting data from any kind of eavesdropping means *Security* concerns should not be neglected either.
- The chatbot needs to be integrated with the UI as well as with the inventory. It needs to be accessed via the app, as well as the website. Better *Interoperability* of the chatbot component is probably the most critical aspect.
- It is clear that the chatbot won't work in all cases, but it should try to cater to common queries and natural digressions. It should also avoid getting into awkward situations arising due to confusion in understanding the user's intentions. *Reliability*, therefore, is important too.

It must be noted that there may be other QAs involved in the example. However, to highlight the subsequent phases of the framework, we stick to the above QAs. This culminates the first phase of the Hospitality framework.

3.6.1.2 Phase 2: Identify Architectural Tactics

Finding relevant Architectural Tactics involves browsing through multiple catalogues. However, Tactics presented in catalogues may be too abstract to apply directly for a particular use case. In such cases, a Software Architect may come up with a set of custom Tactics also. The overall idea is to create a list of possible actions that can be implemented to achieve the QAs in the picture. For the sample use case that we have been tracking, Table 3.6 shows some Architectural Tactics that may be useful in achieving the QAs mentioned in Section 3.6.1.1.

These Tactics are derived based on available Architectural knowledge, as well as considerations for the particular use case. For example, *Abstract Common Services*, *Defer Binding* and *Split Module* are standard Modifiability Tactics available in the Software Architectural Body of Knowledge[163]. On the other hand, *Manage Interfaces* and *Support multiple Data Formats* are the Tactics which are derived from existing knowledge, to suit the particular use case in consideration. Either way, Tactics can be considered as a handle for clubbing related platform features, which can be of interest to the Software Architect. At the end of this phase, the Software

Table 3.6: Finding Tactics for Quality Attributes

QA	Tactic	Reason
Modifiability	Abstract Common Services	Keeping intents, parameters and flow logic separate allows adding or modifying them independently.
	Defer Binding	Allows tailored responses based on user inputs.
	Split Module	Separates the intent matching from business logic.
Security & Privacy	Authenticate Communication	Prevents the chatbot from unauthorized access (superfluous calls to platform may incur additional cost).
	Protect Data at Rest	Keeps the conversations between users and the store private.
	Protect Data in Motion	Prevents breaches due to eavesdropping (e.g. Man-in-the-middle attacks).
Interoperability	Manage Interfaces	Require both ingress and egress capabilities, to and from the chatbot (e.g. API access).
	Support multiple Data Formats	Chatbot needs to take queries (and send responses) in both text and audio formats.
Reliability	Validate common use cases	Verifies that expected user utterances are properly processed by the chatbot.
	Prevent Failures	Restricts the chatbot from responding with low confidence.
	Recover from Failures	Handles known nuances of common conversation (e.g. assuming defaults for missing information).

Table 3.7: A comparison between possible conversations for the Fruit Seller chatbot, with and without digression support

With <i>digressions</i> support	Without <i>digressions</i> support
Bot: <i>What can I do for you?</i>	Bot: <i>What can I do for you?</i>
User: <i>I want to order fruits</i>	User: <i>I want to order fruits</i>
Bot: <i>Which fruit?</i>	Bot: <i>Which fruit?</i>
User: <i>What do you have now?</i>	User: <i>What do you have now?</i>
Bot: <i>Bananas and Apples.</i>	Bot: <i>Sorry, we don't have that!</i>
Bot: <i>Which fruit?</i>	Bot: <i>What can I do for you?</i>
User: <i>Apples</i>	User: <i>What fruits do you have?</i>
Bot: <i>Ok. Ordering Apples...</i>	Bot: <i>Bananas and Apples.</i>
	Bot: <i>What can I do for you?</i>
	User: <i>I want to order apples</i>
	Bot: <i>Ok. Ordering Apples...</i>

Architect has an action plan, i.e. a set of Tactics, which are expected to be achieved when the chatbot is implemented.

3.6.1.3 Phase 3: Identify Platform Features

Table 3.8 shows some features that the Software Architect may narrow down for the tactics shown in Table 3.6. An example of a useful feature exposed by a platform would be the ability to handle *digressions* implicitly. Digressions are natural discourses in a slightly different direction when two human beings converse. We discussed digressions earlier in Section 3.4.5. An example of a conversation which shows the importance of handling digressions was shown in Figure 3.8. For the Fruit Store chatbot, a sample conversation with the user, with and without handling digressions, is shown in Table 3.7. One can see that while the user could still achieve what she wanted (ordering apples), the two chatbots differ from each other in understanding a natural digression in the conversation. The latter could anticipate and prevent a potential failure in understanding the user's intention, making it more reliable than the former.

As another example, to keep the chatbot component easily modifiable, it should be desirable to keep the `intents` and `parameters` independent from each other. In simpler terms, it should be possible to create an “intent without any parameters”

as well as a “parameter without any intent”. This abstraction can help to change them separately, without a cascading effect on the other. This phase ends up with a list of desirable features that the software architect would like to see in the chatbot building platforms. Table 3.8 shows the examples of these features.

3.6.1.4 Phase 4: Evaluate Platforms

The next phase of the framework involves putting “✓” and “✗” against the features shown in Table 3.8, for each candidate platform. For the case study, we chose three popular chatbot-building platforms for evaluation - Google Dialogflow [27], IBM Watson Assistant [21] and Amazon Lex [28]. For all the platforms, we assumed the role of a Software Architect, and performed the two tasks involved in the evaluation - Platform Inspection and Feature Thresholding - as discussed in Section 3.5.2. Table 3.9 shows the results of this phase. It must be noted that these “✓” and “✗” may represent a subjective analysis (involving Feature Thresholding).

3.6.1.5 Phase 5: Calculate Hospitality Indices

The last phase of the framework involves calculation of different Hospitality Indices. The Hospitality Indices at the Tactic level are shown in Table 3.10. The Hospitality Indices at the QA level, with equal weightage to all tactics, are shown in Table 3.11. The process followed for calculating these Indices is detailed in Section 3.5.2. The values for Tactics are calculated by assigning a value of 1 to a “✓” and a value of 0 to a “✗”, and then dividing the sum of the values for its features by the number of features associated with the Tactic. For example, the value of 0.66 for the Tactic *Abstract Common Services* for Dialogflow, is calculated from Table 3.8 and 3.9 as $(1 + 1 + 0) / 3 = 0.66$. In words, this means that two out of the three features desired from the platform to achieve the Tactic are provided. Similarly, Hospitality Index at QA level, is calculated by multiplying the weight of each Tactic to its Hospitality Index. For example, the Hospitality Index for the QA *Modifiability* for Dialogflow, is calculated from Table 3.8, 3.9 and 3.10, assuming equal weights for all Tactics, as $(0.66 + 0.66 + 1) / 3 = 0.773$.

Table 3.8: Using Platform Features for calculating Hospitality Index, at Tactic level

QA	Tactics	Useful Platform Features
Modifiability	Abstract Common Services	Ability to create intents independently
		Ability to create parameters independently
		Ability to manage conversation flow independently
	Defer Binding	Ability to externalise response generation
		Allow placeholders in response to fill parameter values
		Allow conditional responses
	Split Module	Ability to externalise parameter validation
Ability to externalise response generation		
Security & Privacy	Authenticate Communication	Ability to create and verify credentials for accessing the chatbot
		Ability to supply credentials to an external source
	Protect Data at Rest	Ability to create and verify credentials for accessing chat data
		Ability to keep chat transcripts onsite
	Protect Data in Motion	Use secured channels only for communication (e.g. allow <code>https</code> and block <code>http</code>)
Interoperability	Manage Interfaces	Allow API access for intent classification
		Allow API access for slot filling
		Ability to trigger external events
	Support multiple Data Formats	Ability to receive voice input
		Provide transcribed text from speech
		Ability to send voice output
Reliability	Validate common use cases	Provide Test Console to observe chatbot response for specific inputs
		Provide Test Console to observe the debug information for specific inputs
	Prevent Failures	Ability to set confidence threshold for intent classification
		Ability to provide counter-examples
		Ability to digress and return
	Recover from Failures	Ability to provide default conversation flow
		Ability to provide default values for slots

Table 3.9: Availability of useful platform features in three candidate platforms

Useful Platform Features	Feature Available		
	WA	DF	LX
Ability to create intents independently	✓	✓	✓
Ability to create parameters independently	✓	✓	✓
Ability to manage conversation flow independently	✓	✗	✗
Ability to externalise response generation	✓	✓	✗
Allow placeholders in response to fill parameter values	✓	✓	✓
Allow conditional responses	✓	✗	✗
Ability to externalise parameter validation	✓	✓	✗
Ability to create and verify credentials for accessing the chatbot	✓	✓	✓
Ability to supply credentials to an external source	✓	✓	✗
Ability to create and verify credentials for accessing chat data	✓	✓	✓
Ability to keep chat transcripts onsite	✓	✗	✗
Use secured channels only for communication (e.g. allow <code>https</code> and block <code>http</code>)	✓	✓	✓
Allow API access for intent classification	✓	✓	✓
Allow API access for slot filling	✓	✓	✓
Ability to trigger external events	✓	✓	✗
Ability to receive voice input	✗	✓	✓
Provide transcribed text from speech	✗	✓	✓
Ability to send voice output	✗	✓	✓
Provide Test Console to observe chatbot response for specific inputs	✓	✓	✓
Provide Test Console to observe the debug information for specific inputs	✓	✓	✓
Ability to set confidence threshold for intent classification	✓	✓	✗
Ability to provide counter-examples	✓	✓	✗
Ability to digress and return	✓	✗	✗
Ability to provide default conversation flow	✓	✓	✓
Ability to provide default values for slots	✓	✓	✗

WA - Watson Assistant

DF - Dialogflow

LX - Lex

Table 3.10: Hospitality Indices at Tactic level for the three candidate platforms

Tactic	Hospitality Index		
	Watson Assistant	Dialogflow	Lex
Abstract Common Services	1	0.66	0.66
Defer Binding	1	0.66	0.33
Split Module	1	1	0
Authenticate Communication	1	1	0.5
Protect Data at Rest	1	0.5	0.5
Protect Data in Motion	1	1	1
Manage Interfaces	1	1	0.66
Support multiple Data Formats	0	1	1
Validate common use cases	1	1	1
Prevent Failures	1	0.66	0
Recover from Failures	1	1	0.5

Table 3.11: Hospitality Indices at QA level for the three candidate platforms

Quality Attribute	Hospitality Index		
	Watson Assistant	Dialogflow	Lex
Modifiability	1.000	0.773	0.330
Security & Privacy	1.000	0.833	0.667
Interoperability	0.500	1.000	0.830
Reliability	1.000	0.887	0.500

3.6.2 Support of Desired Features in Three Platforms

In Section 3.6.2, we introduced a set of desirable features, distributed across five categories, which a chatbot-building platform should expose. In this case study, we present the results of our analysis over three popular chatbot-building platforms, from the perspective of their support for these features. We picked the same three platforms that we discussed in Section 3.6.1, i.e. IBM Watson Assistant [21], Google Dialogflow [27] and Amazon Lex [28]. Table 3.12 shows a relative comparison between the support of the desirable features on these platforms. The ranks are on a scale of 0 to 1. A rank of 1 means that the feature is supported. A rank of 0

means the feature is not supported. A number in between implies that the feature is supported by the platform, but not up to satisfactory levels (in comparison with the other platforms). The values for the middle and top tiers of the hierarchy are calculated by taking a weighted-sums of their constituent features. It means that each feature in the bottom tier has the same weight - an assumption which is made to consider the most general case (i.e. where these features do not have more or less value for a chatbot project). These values, thus, should only be compared with other values in the same row, and not with any other value across rows or columns. A row, here, represents the “relative” support of the feature (or categories of features) by the three platforms.

We also provide a short description for some entries in Table 3.12 in Appendix B. The capital letters shown in Table 3.12 (e.g. (A) or (B)) are keys to these explanations.

3.7 Related Work and Further Reading

Evaluating Software Architectures prior to their implementation has been studied by multiple researchers, yet there is no single method or framework that is uniformly accepted by the Software Architects. The Software Architecture Analysis Method or SAAM [174] considered an Architecture to have three perspectives - functionality, structure and allocation. When SAAM came out, one of the goals it had was to promote common terminology for describing Software Architecture. It was one of the earliest works at analysing Software Architectures. The Architecture Tradeoff Analysis Method or ATAM [175], built upon learnings from SAAM, was one of the first attempts at analysis Software Architectures through scenarios, a process commonly known as Scenario-based Software Architecture Evaluation. ATAM is an iterative analysis model, which starts by collecting the scenarios and constraints associated with the project, and compares probable Architectural alternatives against multiple Quality Attributes. A more recent evaluation method is the Cost Benefit Analysis Method or CBAM [176]. The approach analyses the benefits or the costs of a particular design decision. CBMA tries to account for costs and benefits “in future” by accounting them appropriately, and guides the design of the Architecture accordingly. There have been efforts dedicated towards tooling for Architectures evaluation (such as [177], [178] and [179]), but tool support for the process is still rather lim-

Table 3.12: Relative support of the Desired Features on three platforms

Desired Platform Features	DF	WA	LX
<i>Intent Management</i>	<i>1</i>	<i>0.9583</i>	<i>0.8177</i>
<i>Create Intents</i>	<i>1</i>	<i>1</i>	<i>0.9375</i>
Add Parameters	1	1	1
Add training examples	1	1	0.75 (A)
Tag Parameter occurrences in training examples	1	1	1 (A)
Map Parameter occurrences to specific values	1	1	0 (A)
<i>Update Intents</i>	<i>1</i>	<i>1</i>	<i>1</i>
Provide new training examples	1	1	1
Edit existing training examples	1	1	1
Remove existing training examples	1	1	1
<i>Delete Intents</i>	<i>1</i>	<i>1</i>	<i>1</i>
<i>Default Intent</i>	<i>1</i>	<i>0.8333</i>	<i>0.3333</i>
Provide default response	1	1	1
Set a minimum confidence threshold to trigger any Intent	1	0.75 (B)	0 (B)
Provide counterexamples – to trigger Default Intent explicitly	1	0.75 (C)	0 (C)
<i>Entity & Parameter Management</i>	<i>1</i>	<i>0.9167</i>	<i>0.8833</i>
<i>Create Parameters</i>	<i>1</i>	<i>0.75</i>	<i>1</i>
Provide possible values for the Parameter	1	1	1
Provide synonymous occurrences for provided values	1	1	1
Allow the extension of possible values list automatically	1	1	1
Disallow extension of possible values list automatically	1	0 (D)	1

DF - Dialogflow

WA - Watson Assistant

LX - Lex

Table 3.12 – continued from previous page

Desired Platform Features	DF	WA	LX
<i>Update Parameters</i>	<i>1</i>	<i>1</i>	<i>0.65</i>
Add more possible values for the parameter	1	1	0.75 (E)
Edit a provided value	1	1	0.75 (E)
Change synonymous occurrences	1	1	0.25 (E)
Delete synonymous occurrences	1	1	0.75 (E)
Remove existing values for the parameter	1	1	0.75 (E)
<i>Delete Parameters</i>	<i>1</i>	<i>1</i>	<i>1</i>
<i>Fulfilment Management</i>	<i>0.8417</i>	<i>0.9833</i>	<i>0.7667</i>
<i>Slot filling</i>	<i>1</i>	<i>1</i>	<i>1</i>
Provide prompts to receive inputs from the user during conversation, for filling a required Slot	1	1	1
Show options (one or more buttons for the user to click, out of a limited set)	1	1	1
<i>Provide a static response for some queries</i>	<i>0.875</i>	<i>1</i>	<i>0.75</i>
Textual Response	1	1	0.75 (F)
Multimedia Response (Image/Audio/Video etc.)	0.75 (G)	1	0.75 (F)
<i>Provide a dynamically crafted response for some queries</i>	<i>0.3333</i>	<i>0.9167</i>	<i>0.3333</i>
Allow the response to have placeholders for Slot values and Context variables	1	1	1
Allow the response to have placeholders for secondary information, e.g. Intent classification confidence or computed values	0 (H)	1	0 (H)
Dynamic response based on variable values (e.g. using conditional operators or <code>if-else</code> ladders)	0 (H)	0.75 (H)	0 (H)

DF - Dialogflow

WA - Watson Assistant

LX - Lex

Table 3.12 – continued from previous page

Desired Platform Features	DF	WA	LX
<i>Trigger external events</i>	1	1	1
<i>Control Slot values interpretation</i>	1	1	0.75
Get the supplied value	1	1	0.75 (I)
Get the reference value	1	1	0.75 (I)
<i>Integration</i>	0.75	0.9375	0.6875
<i>Send requests to a public URL and expect a response back</i>	0.75 (J)	1 (J)	0.25 (J)
<i>Send request to a restricted source (a URL with domain restrictions or a function that can be invoked internally by the platform) and expect a response back</i>	0.25 (K)	1 (K)	1 (K)
<i>Interfaces to integrate with existing platforms (e.g. Facebook Messenger, Slack, Telegram etc.)</i>	1 (L)	0.75 (L)	0.5 (L)
<i>API to create, modify, train, use and delete the Chatbot (e.g. REST APIs)</i>	1 (M)	1 (M)	1 (M)
<i>Conversation Flow Management</i>	0.5625	0.6875	0.3125
<i>Pass contextual information from the currently triggered Intent to any subsequently triggered Intents in the conversation flow</i>	1	1	0.5 (N)
<i>Handle follow-ups (e.g. Confirmations such as Yes or No, Next or Previous etc.)</i>	1 (O)	0.5 (O)	(O)
<i>Handle digressions</i>	0.25 (P)	1 (P)	0 (P)
<i>Process multiple-Intents queries</i>	0 (Q)	0.25 (Q)	0 (Q)

DF - Dialogflow

WA - Watson Assistant

LX - Lex

ited. There are a number of resources which discuss and compare Scenario-based Architecture evaluation, such as [180], [181] and [182].

The work in this chapter is mostly inspired by the work of Agrawal et al. [161]. To the best of our knowledge, the work stands out on its own. The Hospitality Quality Attribute defined by them significantly differs from the other Quality Attributes. First, it is an attribute of the platform used to build a product or service, instead of the product or service itself (or its Containing environment). Second, the Quality Attribute is defined along with a metric - the Hospitality Index, and a framework to evaluate the metric - the Hospitality framework. This triad makes a good combination for applying Hospitality-related concepts to platforms from any domain. While the original work applied it on the cloud platforms, we applied the same on chatbot-building platforms. The major hurdle in the application of the framework to a new domain is to enlist the desired features of the platforms of the specific domain and map them to Tactics and QAs. The hierarchical list of desired features in a chatbot-building platform can help do so.

Several researchers have compared the chatbot platforms from the perspective of the accuracy for matching intents (e.g. [183] and [184]) using standard NLP methods [185]. Berger et al. [186] presented chatbots categorised by properties such as forms of communication, knowledge domain and goal-based. Canonico et al. [187] have presented a taxonomy focusing on 13 facets of chatbot platforms such as usability for developers, language support, linkable intents and price. Braun et al. [183] presented a reference architecture for chatbot platforms, which provides a handle to compare features. Peras [188] presented five perspectives for chatbot evaluation - user experience, information retrieval, linguistics, technology, and business for comparison. Braun et al. [189] also showcased a framework for comparison using features such as Input/Output, Timing, Flow, Platform and Understanding. Daniel et al. [190] highlighted the need for attributes such as *Productivity*, *Maintainability*, *Reusability* and *Interoperability* as motivation for their multi-platform chatbot modelling framework - Jarvis.

For further reading, checking out platform comparison articles (e.g. [129], [130], [133], [93] and [191]) is a good starting point. Almost all chatbot companies operate blogs that cover latest developments in chatbot-related technology (e.g. [169], [192], [193] and [170]). However, these sources are almost always biased towards one platform (e.g. as admitted by the author in a Disclaimer on [92]). Therefore, these articles may not always provide a balanced view of the capabilities of other

platforms. However, since the chatbot-building platforms are constantly evolving, with new entrants joining the field every year, research papers may not be as helpful as recent articles on the Internet. Another quick reference option is to check out the Release Notes on the platforms. They usually mention, in short, the features recently added to the platform, and can be helpful while inspecting the platform for desired features.

3.8 Summary

This chapter was dedicated to describing the importance of chatbot-building platforms. We first discussed some common Constraints associated with chatbot development, which may force the developers to do more custom development, rather than using a platform. These issues included problems with Privacy, Developmental Flexibility, Natural Language support (especially for languages other than English), Pricing (related to the development and post-deployment operation) and Geographical proximity of the deployment servers.

We then presented a categorisation of chatbot-building platforms based on the services they offer. The NLP-as-a-service (NLPaaS) platforms provide support for basic NLP tasks such as Entity Extraction and Text Categorisation, which can be used to compose a chatbot. The Conversation-as-a-Service (CaaS) platforms provide support for tasks that are specific to a chatbot, such as matching queries to Intents, parsing Parameter values from user utterances and managing the flow of the conversation. The ChatWidget-as-a-Service (CWaaS) platforms offer visual interfaces to build chatbots, often using a drag-and-drop mechanism. The chatbots built on CWaaS platforms are usually tightly coupled with a deployment environment since they provide a chat widget that can be deployed straight away on these platforms.

We also presented a contrasting description of the dashboards of CaaS and CWaaS platforms. While the dashboards of CaaS platforms are built to take textual input from the developers, the CWaaS platforms provide an interface similar to that used for making flowcharts.

Next, we presented a list of desired features that a chatbot-building platform should expose to support a wide range of chatbot use cases. We arranged these features in five major categories, which are based on the discussions in Chapter 2.

These categories included; features related to managing Intents, features related to managing Parameters, features dealing with Conversational flow, features dealing with Fulfilments and features relating to chatbot's Integrations with the rest of the environment. This list can also be helpful for selecting platforms based on their versatility.

We then presented an application of the Hospitality framework over the chatbot-building platforms. The framework allows evaluation of support that a particular platform provides towards achieving Quality goals. These goals may be implementing an Architectural Tactic or maximising a Quality Attribute. The framework starts with a set of Quality Attributes. A collection of Architectural Tactics are then found to achieve them. For every Tactic, the Software Architect has to come up with a list of features which can be helpful in implementation of the Tactic. The candidate platforms are then evaluated, and the availability of these features are compared against some threshold, to consider them useful or useless. The Hospitality Indices can then be computed using a weighted sum approach, for either a Tactic or a Quality Attribute. A Multi-criteria Decision Analysis method, such as TOPSIS, can then generate a ranked list of the platforms, provided that the Software Architect can provide relative weights to the Quality Attribute in consideration.

We then presented two case studies. The first showed the application of the Hospitality framework on a simple set of chatbot use cases. The second one involved an in-depth analysis of three chatbot-building platforms, and inspect their relative support for the proposed desired features in the chapter.

Chapter 4

Contextual Reactive Pattern

In Chapter 2, we discussed the architecture of a chatbot and its Containing system. Chapter 3 was dedicated towards discussion of chatbot-building platforms. In this chapter, we move ahead to the next problem of defining a chatbot on a chatbot-building platform. In particular, this chapter attempts to answer the following two question:

- Given a set of use cases to be served by a chatbot, how can we express them on a chatbot-building platform?
- Given that most software development projects today are built in iterations, how can we model the definition of the chatbot, such that it can be easily modified and improved over multiple iterations?

This chapter is organised differently from the rest of the chapters since it presents a pattern called the *Contextual Reactive* chatbot definition pattern. The organisation is loosely based on the document titled “How to write a pattern?” by Tim Wellhausen and Andreas Fießer [194].

This chapter is organised as follows; In Section 4.1, we provide an overview of the *Contextual Reactive* pattern, describing its scope and the challenges that it attempts to mitigate. Next, in Section 4.2, we describe the solution that this pattern presents. We then discuss some examples of how this pattern is adopted by real-world platforms in Section 4.3. Section 4.4 presents a case study, where we show the details of describing a chatbot’s use cases on a platform. We then discuss

the consequences of this definition pattern, including its benefits and limitations, in Section 4.5. Finally, we summarise the chapter in Section 4.6.

4.1 Pattern Overview

Building a chatbot with an iterative development process poses certain challenges for the chatbot developer. The developer is expected to produce a deployable version of the chatbot at the end of a short development cycle. In Agile Software Development, the counterpart of use cases are *user stories* [195]. Every iteration should incrementally increase the capability of the chatbot and implement a subset of overall user stories based upon a priority list, similar to any other project developed using iterative development. To do so, for every query the chatbot is expected to answer, the developer must evaluate the intention of the user. Based on the intention, the query must be processed differently, which may involve the execution of some business logic. Besides, the processing of the query may require specific data items which the user must supply as part of the conversation with the chatbot. Thus, the chatbot is defined by supplying a “context” that it may encounter, and the “reaction” that must take place when the context is observed. We call this pattern, the *Contextual Reactive* pattern for chatbot definition.

4.1.1 Context

The Context of a pattern (not to be confused with the term *Context* in *Contextual Reactive* pattern) is the overall picture where the problem and the solution to the problem exist. The Context for the pattern is as follows:

- Organisations are rapidly adding chatbots to their business operations to improve customer experience [3].
- Chatbots are being built to serve a wide variety of use cases, such as E-commerce, Customer Service, Information Retrieval and Travel Assistance [196].
- Any Iterative software development process (e.g. Scrum) attempts to build

components, including chatbots, in an incremental fashion (e.g. in Sprints) [197] [198].

- There is an organisation which wants to deploy a chatbot to act like a conversational interface towards its business processes. They would like to start with simple and non-essential business operations, and slowly move towards more complex and essential business operations.

4.1.2 Problem

The Problem section of a pattern highlights the major issues which the pattern addresses. The pattern may provide a comprehensive solution for these issues, or, act as a step towards its resolution. The problem that this pattern attempts to resolve is summarised as follows:

- To build a chatbot to cater to a set of user queries, where the chatbot acts like a conversational interface towards accessing a set of business operations.
- To be able to produce a deployable chatbot in a short time span, albeit with minimal features, and update it over multiple iterations to cater to all the user queries.
- To be able to handle the imperfections in Natural NLP tasks [199] [200] associated with the chatbot, in a graceful manner.
- To build the chatbot with no or minimal changes in business operations code-base.

4.1.3 Forces

Forces related to a pattern are reasons that motivate the use of the solution provided by the chatbot. This section also provides arguments in favour of this particular solution, in case the problem can be solved via other methods as well. For the *Contextual Reactive* pattern, the other major solution to the issues described in Section 4.1.2 is to go for custom chatbot development. The Forces section, thus, delineates the major motivating factors to use the chatbot-building platforms, and

hence, use the definition pattern to model the chatbot use cases. The Forces for the pattern are summarised as follows:

- An iterative development process, such as Scrum, provides a framework to build software components over multiple iterations, with each iteration building upon or improving the artefact produced in previous iterations.
- Chatbot-building platforms provide crucial NLP support which reduces the time for each iteration. They also provide better error handling mechanisms to tackle known issues and imperfections in these NLP tasks.
- Rewriting business operations specifically to be connected to a chatbot may increase the load on the chatbot developer, and maybe undesirable for other reasons as well (such as preventing duplication of business logic).

4.2 Solution

In a nutshell, the problem for which this pattern proposes a solution is to define a chatbot over a chatbot-building platform, keeping in mind that the development may go through multiple iterations. Another aspect of the problem is that the business operations that the chatbot has to invoke should remain as decoupled as possible from the chatbot so that no or minimal changes are required in them. We now describe the solution formally, in more detail:

Figure 4.1 shows the solution's outline. The core idea is to pick a chatbot building platform and an iterative development process. The features to be implemented in each iteration must be mapped to a certain format and supplied to the platform, which in turn, creates a deployable chatbot which can be used for operations.

The details of the solution are highlighted as follows:

- **Pick a chatbot-building platform to perform the NLP tasks.** The chatbot building platform provides two vital services:
 - For every query that the chatbot receives, the platform attempts to guess the user's intention. The chatbot developer predefines a set of possible intentions, and the platform maps the query to one (and only one) of those intentions.

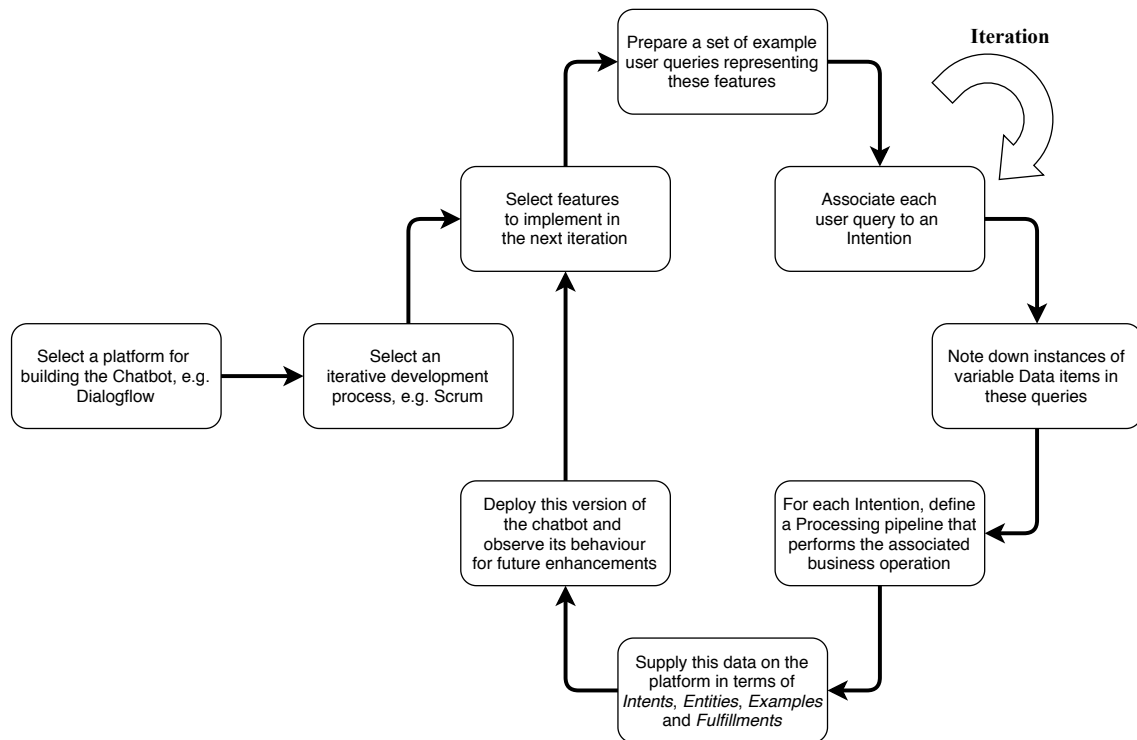


Figure 4.1: Building Chatbots using a platform with the *Contextual Reactive* pattern

- A user query may provide implicit hints or explicit inputs required for fulfilling their intention. The platform identifies these inputs, extracts them and supplies them downstream to process the query.
- **Select the set of user stories to be served by the chatbot during the current iteration.** The factors affecting this decision are specific for a particular project, but the most important aspect is the priority of user stories based on the business operation that they represent.
- **Agree upon a set of user intentions to cater.** The platform expects that every user query must be associated with a pre-configured intention. We refer to these intentions as *Intents*. The idea is to assign an appropriate processing pipeline to each user query and generate a suitable response at the end of the processing pipeline. From an implementation perspective, if there are two queries, where the processing logic and the response generation method are exactly the same, it makes sense to categorise them under the same Intent.

- **Figure out the required and optional data items that may be present in a user query.** The platform allows the chatbot developer to define these data items in a fashion similar to defining variables in a program, i.e. they have specific types (e.g. Numeric or Ordinal) and can take values from a finite or infinite input set. We refer to these data items as *Entities*.
- **Work out the relationships between the defined Intents and Entities.** Certain Entities may only be present in queries belonging to certain Intents. We say that an Intent has a *Slot* for an Entity if the values for the Entity can appear in queries associated with the Intent. A Slot may be “optional”, i.e. the query can still be processed if the user does not provide a value for it, or, it may be “required”, meaning without that particular detail, processing pipeline cannot be invoked. For the Slots of the latter type, platforms usually have a mechanism to produce a temporary response, in the form of a question, asking the user to supply a value explicitly.
- **Prepare a collection of possible user utterances.** The platform requires training data to build models for performing NLP tasks. This data is supplied in terms of sample user queries. We refer to these queries as *Examples*. The chatbot developer provides a few Examples for each defined Intent and tags any instances of values associated with any defined Slots within these queries.
- **Connect the processing pipelines and response generation logic to respective intentions.** The platform maps a user query to its associated Intent. It also attempts to parse instances of any Slot data supplied in the query. Then the platform invokes a pre-configured processing pipeline, which we term as the *Fulfillment* for the mapped Intent. This invocation involves supplying the parsed values of Slots as well as any additional contextual information and expecting a generated Natural Language response which is relayed to the user. Two common examples of processing pipelines are serverless functions [201] and REST API endpoints [202] implementing specific business operations.
- **Handle the user stories to serve in next iterations.** In addition to the defined Intents, another Intent can be defined to cater to the user stories which will be served by the chatbot in future iterations. We refer to such an Intent

as the `others` Intent. It involves providing Examples for such user stories, but not connecting them to a processing pipeline. Instead, this Intent could be configured to output static responses (e.g. “We currently do not support this feature”).

- **Handle imperfections in NLP tasks.** In addition to the above Intents, define another Intent to handle the “irrelevant” or “ill-formed” queries. Platforms usually have a mechanism to invoke a special processing pipeline for cases where the query could not be mapped to any defined Intent. Theoretically, this can be seen as the presence of a “default” or “fallback” Intent. Typically, the “default” Intent can either be handled via a special processing pipeline (e.g. transferring the chat session to a human) or responded with static responses (e.g. “Sorry, can you rephrase the query?”).
- **Deploy the chatbot and collect usage data.** A detailed analysis of the chat data can provide important insights. In particular, an important dimension of inspection is the proportion of the queries that matched the `others` Intent. A high percentage indicates one of the two possibilities:
 - The current set of user stories catered by the chatbot represent only a small fraction of the overall use cases that the user expects the chatbot to serve.
 - The platform is not able to perform a good job of mapping a user query to its correct intention. It indicates that the platform needs more Examples for training, or the quality of Examples used previously were not up to the required quality mark.

This data may provide crucial inputs for planning the next iteration.

- **Plan the next iteration.** This involves catering to other user stories and refining the definitions or Examples for the existing user stories. As more Intents are added iteratively, the number of user stories handled by the `others` Intent keeps reducing every iteration, and finally, the Intent is removed.

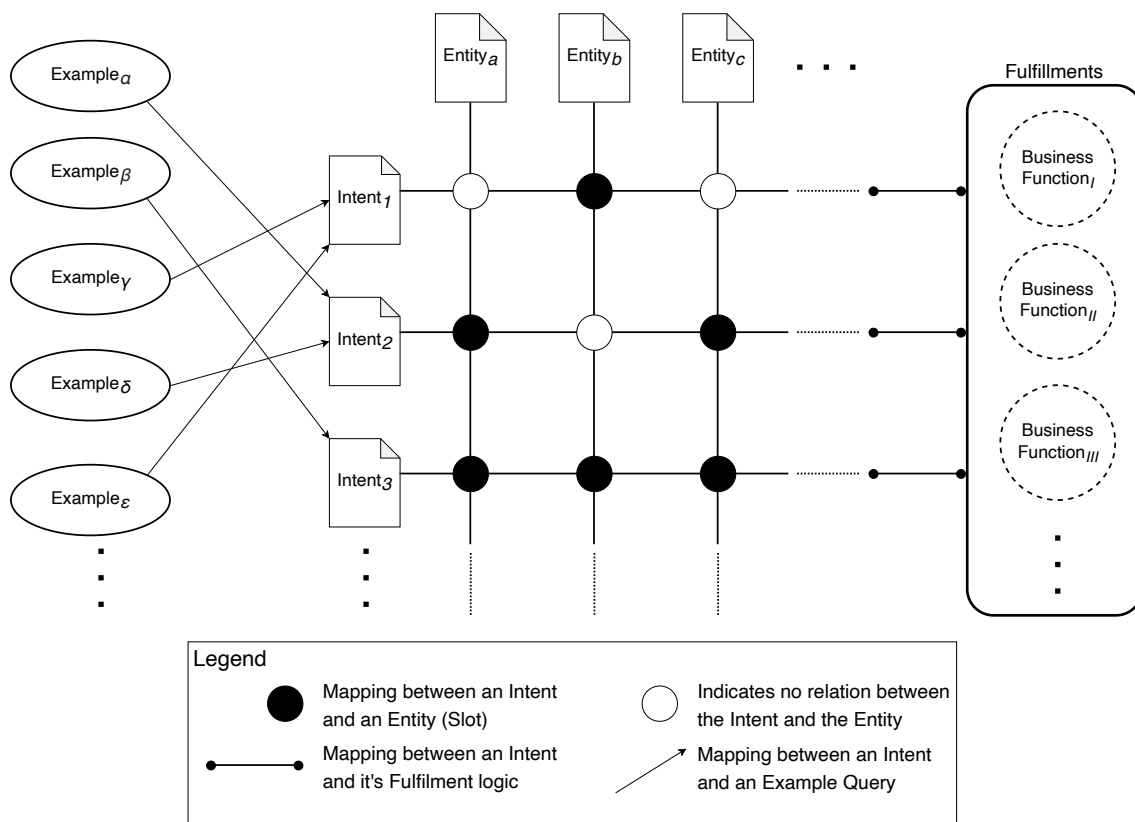


Figure 4.2: **Solution Structure** : How Entities, Intents, Examples and Fulfillments interrelate to each other in an application.

4.2.1 Structure

The Structure subsection of the Solution section of a pattern discusses the relationships between the different elements of the solution. For the *Contextual Reactive* pattern, the structure shows how Intents, Entities, Examples and Fulfillments relate to each other in a chatbot's definition.

Figure 4.2 shows the elements of the chatbot definition and their relationships. A summary of the elements is as follows:

- **Intents:** An Intent represents a collection of related user queries, associated with a particular user story. The set of Intents for a chatbot partition all possible queries that the chatbot may receive into mutually exclusive categories.
- **Entities:** Entities represent real-world objects supplied implicitly in a partic-

ular query. Each Entity associated with a chatbot can accept a value from a finite or infinite set, predefined at the time of chatbot definition.

- **Examples:** Examples are sample user queries that a user may fire at the chatbot during a conversation. An Example is mapped to one and only one Intent and may contain values associated with zero or more Entities.
- **Fulfilments:** Fulfilments represent the processing pipeline that must be executed to prepare a response for any query. In some cases, where the response is a static message, the platform usually provides a mechanism to output the same to the user without making a remote function call. Fulfilments can also be gateways to external API endpoints which can be invoked to prepare the response (as well as perform any actions) for the given query. A fulfilment is associated with a respective Intent and is invoked every time a query is classified to have the said Intent.

In addition, **Slot** is a term that represents the instantiation of an Entity with respect to a particular Intent. An analogy would be - Slots are to Entities, what Objects are to Classes. A Slot thus represents the mapping between an Intent and an Entity.

4.2.2 Dynamics

The Dynamics subsection of the Solution section of a pattern discusses how the elements come together in solving the problem when in action. For the *Contextual Reactive* pattern, the details of the dynamics are usually hidden, since it is performed by the platform. In any case, a simple representation of the process that platforms employ is shown in Figure 4.3.

The platform collects vital pieces of the supplied information, namely the Intents, Entities and Examples, and builds some NLP models. These models are used later by the chatbot. Any changes - addition, deletion or modification - in any one of these components require retraining. Some platforms, like Dialogflow [27] and Watson Assistant [21], do this automatically on detection of a change, while some others like Lex [28] may force the developers to do this explicitly. The training process usually does not take more than a few minutes, hence allowing seamless modification to the above information, as and when required.

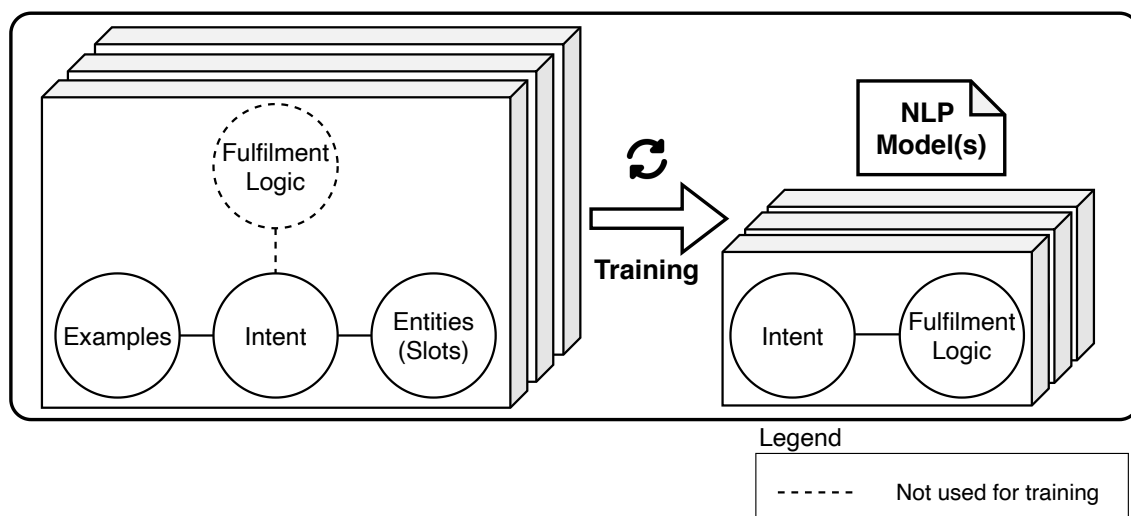


Figure 4.3: **Solution Dynamics** : How a platform uses Entities, Intents, Examples and Fulfilments for building the chatbot.

4.2.3 Deployed Chatbot's Sketch

At the end of each development iteration, a deployable chatbot is produced. The models that were built after training are connected by the platform to perform two critical tasks - Intent Classification and Parameter Extraction (also called Slot Filling). The Fulfilments supplied by the developer are used as processing pipelines for the queries that the chatbot receives. Figure 4.4 shows an overview of the deployed chatbot.

After deployment, the chatbot performs some crucial NLP tasks and follows a workflow. A typical workflow is shown in Figure 4.5. A summary of the workflow is provided below:

1. The user asks a query. The query may have implicit values associated with one or more Entities.
2. The chatbot performs two NLP tasks - guessing the Intent associated with the query and finding out if there are values associated with any defined Entities, supplied within the query.
3. For the detected Intent, the chatbot looks at the Slot details (recall that a Slot represents a relationship between an Entity and an Intent). If there are parsed

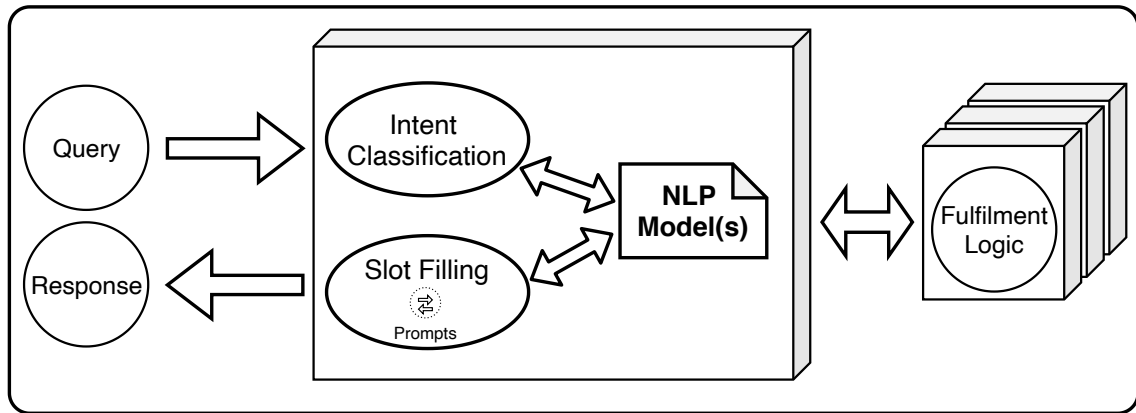


Figure 4.4: **Deployed Chatbot's Sketch** : How the platform uses the built NLP models along with the defined Fulfilments to construct the chatbot.

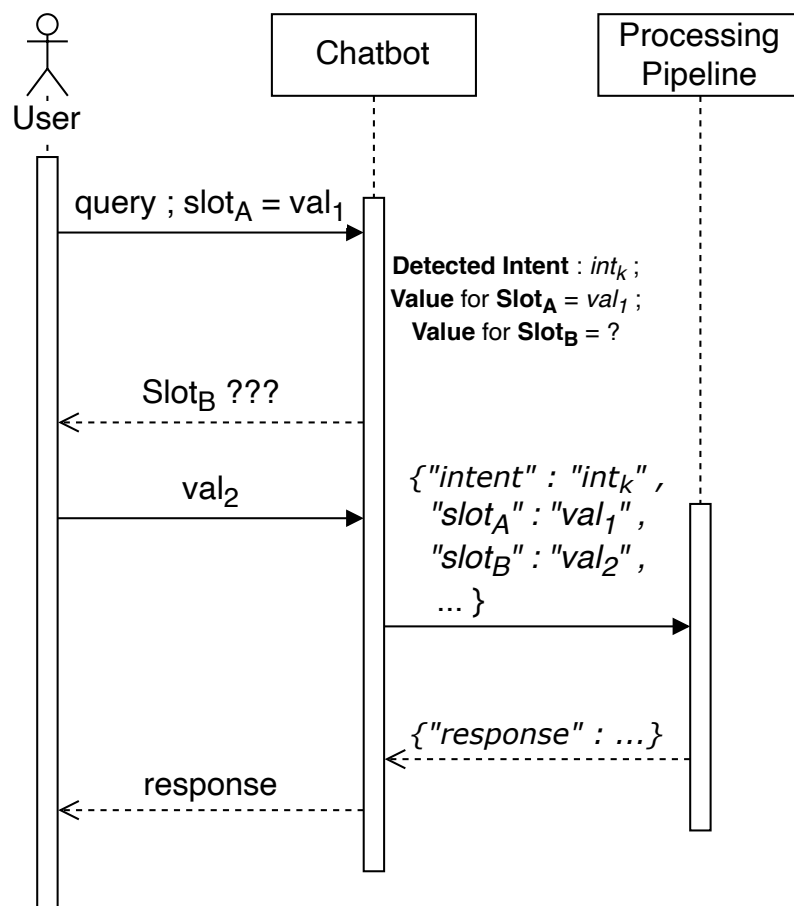


Figure 4.5: **Deployed Chatbot's Sketch** : Typical workflow observed during the operation of a chatbot.

values which belong to one or more Slots, they are stored temporarily. Next, the chatbot checks if there are any Slots defined as “required”. If present, the chatbot checks if any values for these Slots are present in the temporary storage or not. If not, the chatbot starts producing a series of responses, essentially prompting the user to supply these values. This process may continue till values for all the “required” Slots are gathered, or the chatbot may give up after a certain number of attempts, providing a specific response configured for such cases (the exact behaviour may vary from one platform to another).

4. Once the chatbot has values associated with all “required” Slots (and any “optional” Slots, if provided), the chatbot passes this information to the processing pipeline. **This is the step which decouples logic associated with business operations from the intricacies of the chatbot implementation.** The processing pipeline can reside inside an external system or maybe part of a legacy system already operational. The query may trigger an idempotent operation such as a lookup in a database, or, it may perform actions in the business domain, such as creating a new booking. The details of the operation are abstracted from the chatbot. The chatbot only expects a response, generated at the completion of the processing.
5. The response received after the processing is relayed back to the user. The chatbot now expects another query from the user, and the cycle is repeated.

4.3 Pattern Examples

There are many platforms which provide support for building chatbots. The *Contextual Reactive* pattern for chatbot definition can be observed in almost all of them with minor variations. We discuss three such platforms in brief - Dialogflow [27], Watson Assistant [21] and Lex [28].

4.3.1 Dialogflow

- **Intents:** Each Intent is defined in the dashboard on a separate page. Typical information provided on the definition page is Intent’s name and some

“Training Phrases”. If the Intent has any associated Slots, the same needs to be configured on this page itself. In Dialogflow, Slots are called “Parameters”. Prompts can be configured for one or more Parameters on the same page. Dialogflow provides a set of common Intents which can be added to any chatbot (e.g. Intents for casual conversation with a user). It also adds a `Welcome` and a `Default Fallback` Intent by default to every chatbot to greet and handle irrelevant user queries respectively.

- **Entities:** Each Entity is defined in the dashboard on a separate page. There are some System Entities, which Dialogflow provides for usage directly, for instance, Geographic Locations or instances of Time and Date. However, for most user stories, one would need to define custom Entities, which are nominal in nature. For each Entity, a set of possible values are defined. For each value, synonyms can be defined. Dialogflow offers the option to intelligently extend possible values for an Entity, in case it is not feasible to list down all of them beforehand.
- **Examples:** Examples are called “Training Phrases” in Dialogflow. They are provided directly on the definition page of the associated Intent. Dialogflow expects that instances of any Entity are tagged in the Example, to help the Slot Filling task. Negative Examples can be configured on the page of the `Default Fallback` Intent. These Examples provide explicit hints to trap “irrelevant” user queries.
- **Fulfilments:** Fulfilments can be defined in two different ways in Dialogflow. For Intents, where the response is fixed, the response, along with placeholders for Parameters, can be directly supplied on the Intent definition page. For Intents requiring complex Fulfilment Logic, one has to define an Action - a unique tag associated with the Intent - and provide a common Fulfilment webhook on Fulfilments page. For such Intents, the chatbot sends a POST HTTP request to the configured URL with the name of the Action and other inputs, such as parsed values of the Slots. The webhook must provide a JSON reply with a fixed schema. The response is then relayed back to the user.

4.3.2 Watson Assistant

- **Intents:** Each Intent is defined in the dashboard on a separate page. Typical information provided on the definition page is Intent’s name and some “User examples”. The Intent name starts with a # (e.g. #schedule-query). Slot details are not provided on the Intent definition page. Instead, they are configured using the “Dialog Tree”, a tree-like Flow Management structure that provides more Flexibility in defining Slots and Prompts. Watson Assistant provides a Content Catalog for common use cases, from where Intents can be imported and added to the skill. Watson Assistant does not have a Fallback Intent. Negative examples must be marked as “irrelevant” on the Test Console to be considered while training.
- **Entities:** Each Entity is defined in the dashboard on a separate page. The Entity name starts with a @ (e.g. @FlightNumber). For each Entity, a set of possible values are defined. For each value, synonyms can be defined. Watson Assistant offers the option to intelligently extend possible values for an Entity, in case it is not feasible to list down all of them beforehand. There is an *Annotations* tab on the definition page, which shows Examples (across all Intents) where some value of the Entity appears.
- **Examples:** Examples are called “User Examples” in Watson Assistant. They are provided directly on the definition page of the associated Intent. Watson Assistant expects that instances of any Entity are tagged in the Example, to help the Slot Filling task. Negative Examples cannot be configured directly. To provide them one has to use their Test Console and mark a query as “irrelevant”. The other way could be exporting the Skill configuration as a JSON file, provide these Examples under the “counterexamples” attribute, and import the same.
- **Fulfilments:** Watson Assistant provides multiple options to define Fulfilment Logic. It provides a tree-like structure called the Dialog Tree, where Fulfilments can either be defined inline or, calls can be made to IBM Cloud Functions [81]. Calls to an external Fulfilment webhook can also be configured. Watson Assistant’s Dialog Tree allows more business logic to be added to pro-

cess any pre-conditions before or post-conditions after the Fulfilment Logic is executed.

4.3.3 Lex

- **Intents:** Each Intent is defined in the dashboard on a separate page. Typical information provided on the definition page is Intent’s name and some “Sample utterances”. If the Intent has any associated Slots, the same needs to be configured on this page itself. Prompts can be configured for one or more Slots on the same page. Lex provides a set of common Intents which can be added to any chatbot, albeit, these Intents are more useful for chatbots that work over audio content (some of the examples of built-in Intents are the `PauseIntent`, `ShuffleOnIntent`, `LoopOffIntent` etc.). Lex does not provide any Welcome or Fallback Intents, although there is a section for configuring “Error Handling”, which is loosely the same as a default Intent.
- **Entities:** Entities cannot be defined separately in the dashboard. In Lex, Entities are called “Slot Types”. Slot Types cannot exist independently unless they are associated with some Intent. However, an already defined Slot Type can be reused in another Intent. For each Slot Type, a set of possible values are defined. For each value, synonyms can be defined. Every time a change is made to a Slot Type, it is saved as a new version. Different Intents can use different versions of the same Slot Type.
- **Examples:** Examples are called “User utterances” in Lex. They are provided directly on the definition page of the associated Intent. Lex expects that positions of any Entities in an Example are explicitly mentioned, to help the Slot Filling task. Lex only asks the position of the Entity in the Example and not the value. There is no mechanism in Lex to provide Negative Examples.
- **Fulfilments:** Fulfilments can be defined in two different ways in Lex. For Intents, where the response is fixed, the response, along with placeholders for Parameters, can be directly supplied on the Intent definition page. For Intents requiring complex Fulfilment Logic, Lex allows configuring an AWS Lambda Function [82] where the Fulfilment Logic can be implemented in a serverless

environment. For such Intents, the chatbot invokes the configured Lambda Function with standard input and expects a standard output from which the response is relayed back to the user.

Appendix C provides a comparison between the three platforms in terms of their alignment with the *Contextual Reactive* pattern.

4.4 Case Study

As an example, we show how a chatbot can be built with a platform using the *Contextual Reactive* pattern. Consider a fictitious airline called Chanakya Airlines. Assume that they operate a limited number of flights daily between a few cities¹. The airline already has a website through which users can perform basic business operations such as searching for an appropriate flight, booking a ticket and cancelling a booked ticket. The airline, in an attempt to provide better user experience, wishes to deploy a chatbot on their website, which can help users perform the same operations through a conversational interface. After a few sprint planning meetings, it was decided that the chatbot will be built using the Dialogflow [27] platform. A set of user stories were also finalised. The operations team of the airlines are open to sharing a REST endpoint with the chatbot developer, which can be invoked to perform the business operations. The endpoint takes a JSON object as input and returns another JSON object as the response after the execution of the operation.

4.4.1 User Stories

The chatbot developer has to implement certain user stories in the priority of their business importance. The user stories, in decreasing order of their priority, are as follows (the format used to express these stories is loosely based on the format discussed by Mike Cohn [203]):

¹For the demo, we picked only three cities - Delhi, Mumbai and Bengaluru

1. Search for a flight

- Description:

As a customer of Chanakya Airlines, I want to search for a flight between two cities on a particular date.

- Example:

As a customer of Chanakya Airlines, I want to know about all the flights, if there are any, between Delhi and Mumbai on coming Friday.

- Conditions of Satisfaction:

- (a) The user shall be prompted to supply a source, a destination and a date.
- (b) A list of applicable flights (if any) shall be shown.
- (c) *Nice to have* - show only day or night flights (as per user's preference).

2. Book tickets on a flight

- Description:

As a customer of Chanakya Airlines, I want to book one or more tickets on a particular flight for a particular date.

- Example:

As a customer of Chanakya Airlines, I want to book a ticket on the flight EX-101 for tomorrow.

- Conditions of Satisfaction:

- (a) The user shall be prompted to supply an email and the date of booking.
- (b) On success, the user shall be given a Booking Id for future reference.
- (c) *Nice to have* - multiple tickets could be booked in a single booking (if the user wishes to book more than one ticket on the same flight on the same date).

3. Show previously made bookings

- Description:

As a customer of Chanakya Airlines, I want to find out all the previous bookings I made with Chanakya Airlines (that are not cancelled).

- Example:
As a customer of Chanakya Airline, I want to know the details (such as Flight Number, Source, Destination, etc.) of my previous bookings.
- Conditions of Satisfaction:
 - (a) The user shall be prompted to supply the email used for the bookings.
 - (b) A list of all bookings (if any) made with the email (that are not cancelled) shall be shown.
 - (c) *Nice to have* - show bookings for a particular travel date only (as per user's request).

4. Cancel a booking

- Description:
As a customer of Chanakya Airlines, I want to cancel a previous booking made with Chanakya Airlines.
- Example:
As a customer of Chanakya Airlines, I want to cancel a booking with Booking Id 1001.
- Conditions of Satisfaction:
 - (a) The user shall be prompted to supply a Booking Id and the email used for making the booking.
 - (b) On success, the user shall be informed about the cancellation, and the booking details must be removed from the bookings database.

4.4.2 The First Sprint

In the first sprint, the chatbot developer picks the top two user stories to implement. In addition, the developer has decided to achieve only the core conditions of satisfaction - keeping the *Nice to have* aspects for the second sprint. The chatbot developer maps the user stories to the following elements on Dialogflow:

- **Intents:**
 - The `search` Intent caters to inquiries about flights.

- The `book` Intent handles requests for booking flight tickets.
- The `others` Intent handles requests related to user stories to be handled in the next sprint(s).
- Dialogflow provides a `Default Welcome` Intent and a `Default Fallback` Intent automatically. The former can cater to mundane greeting queries (e.g. `Hi` or `Hello`), whereas the latter produces a response to cater to queries that may be “irrelevant” for the chatbot (e.g. `How’s the weather today?`).

- **Entities:**

- The `FlightNumber` Entity is a *regex* Entity, which can take values in the form `EX-ddd`, where *d* represents a digit. It represents the flight number of a particular Chanakya Airlines flight.
- The `Source` Entity is a *system*¹ Entity which can take a value of any major city such as Delhi or Mumbai. It represents the origin of a particular Chanakya Airlines flight.
- The `Destination` Entity is another *system* Entity similar to the `Source` Entity and can take a value of any major city. It represents the destination of a particular Chanakya Airlines flight.
- The `Date` Entity is a *system* Entity which can take a value of valid date. Dialogflow also maps relative utterances such as `tomorrow` or `yesterday` to a date string in ISO 8601 format [204], e.g. `2020-05-01T12:00:00+05:30` (representing 1st May 2020 in Indian Standard Time). It represents the date of departure of a particular Chanakya Airlines flight.
- The `Email` Entity is a *system* Entity which can take a value of a valid email address. It represents the email used to make a booking with Chanakya Airlines.

¹Dialogflow provides predefined system entities to capture cities, dates, emails, numbers etc.

- **Slots:**

- For search Intent:

Entity	Parameter Name	Required
Source	src	✓
Destination	dest	✓
Date	date	✓

- For book Intent:

Entity	Parameter Name	Required
FlightNumber	flightNumber	✓
Date	date	✓
Email	email	✓

For each required slot, a set of *prompts* were also provided. For example, if the email for the booking is not supplied, the chatbot may respond with:

Please provide the email to use for the booking...

- **Fulfilments:**

The fulfilment logic is available externally. A REST endpoint is invoked with different inputs to execute different business operations. Sample inputs for the two Intents are as follows¹:

- For search Intent:

```
{
  "operation": "search",
  "parameters": {
    "src": "delhi",
    "dest": "mumbai",
    "date": "2020-05-01T12:00:00+05:30"
  }
}
```

¹The response received from the REST API looked like: { "fulfillmentText": *response* }

– For book Intent:

```
{
  "operation": "book",
  "parameters": {
    "flightNumber": "EX-101",
    "date": "2020-05-01T12:00:00+05:30",
    "email": "ssri@cse.iitk.ac.in"
  }
}
```

For the `others` Intent, some static responses are configured, meaning if the `others` Intent is triggered, the chatbot may respond with a response like:

We request you to connect to our Customer Care
for assisting you with this issue ...

- **Examples:**

A set of examples for both search and book Intents were provided:

– For search Intent:

Training phrases ? Q ^

” Add user expression
” show flights from delhi to mumbai for 20th April
” show flights from delhi to mumbai
” show flights for 20th April
” show flights

– For book Intent:

Training phrases ⓘ Search training phrases 🔍 ^

” Add user expression
” book a ticket on EX-101 for tomorrow with email ssri@cse.iitk.ac.in
” book a ticket on EX-101 for tomorrow
” book a ticket on EX-101
” book a ticket
” book a flight

– For others Intent:

Training phrases ⓘ Search training phrases 🔍 ^

” Add user expression
” cancel booking with booking id 1001 and email ssri@cse.iitk.ac.in
” cancel booking with booking id 1001
” cancel my booking
” cancel booking
” show bookings with email ssri@cse.iitk.ac.in for 20th April
” show bookings with email ssri@cse.iitk.ac.in
” show my bookings
” show bookings

We refer to the deliverable built after the first sprint as **Chanakya-Airlines-Bot-v1**.

4.4.3 The Second Sprint

The second sprint involved adding functionality to the chatbot built in the first sprint and improving it by adding the *Nice to have* features. The main agenda of the second sprint is summarised as follows:

1. Implement the two remaining user stories and remove the `others` Intent created to handle queries related to them.
2. Implement the *Nice to have* features related to each user story.

Thus, the second sprint showcases two major tasks. First, it upgrades the existing Intents to cater to more complex scenarios. Second, it adds more Intents to upgrade the overall capability of the chatbot. Both these tasks help us showcase how flexible the pattern is towards rapidly changing chatbots.

We only discuss the details that were added in the second sprint:

- **Intents:**

- The `show` Intent caters to inquiries about existing bookings (those that were not cancelled).
- The `cancel` Intent handles requests for cancelling existing bookings.

The `others` Intent is no longer required, since we now have Intents related to all user stories. Thus, it is removed in the second sprint.

- **Entities:**

- The `Time` Entity is a *nominal* entity, which can take two values - “`day`” or “`night`”. We define a few synonyms for both values, such as “`morning`” for “`day`” and “`late`”¹ for “`night`”.
- The `NumberOfTickets` Entity is a *system* entity that can take any integer as value. It represents the number of tickets to book as part of a single booking.
- The `BookingId` Entity is a *system* entity that can take any integer as value. It represents the Booking Id for a particular Chanakya Airlines booking.

¹The synonyms are contextual - e.g. “night flight” and “late flight” mean the same here

- **Slots:**

The slots added to existing Intents in this sprint are shown *differently*:

– For search Intent:

Entity	Parameter Name	Required
Source	src	✓
Destination	dest	✓
Date	date	✓
<i>Time</i>	<i>time</i>	<i>✗</i>

– For book Intent:

Entity	Parameter Name	Required
FlightNumber	flightNumber	✓
Date	date	✓
Email	email	✓
<i>NumberOfTickets</i>	<i>numberOfTickets</i>	<i>✗</i>

– For show Intent:

Entity	Parameter Name	Required
Email	email	✓
Date	date	<i>✗</i>

– For cancel Intent:

Entity	Parameter Name	Required
Email	email	✓
BookingId	bookingId	✓

- **Fulfilments:**

Sample inputs for the two new intents are shown as follows:

- For show Intent:

```
{
  "operation": "show",
  "parameters": {
    "email": "ssri@iitk.ac.in"
  }
}
```

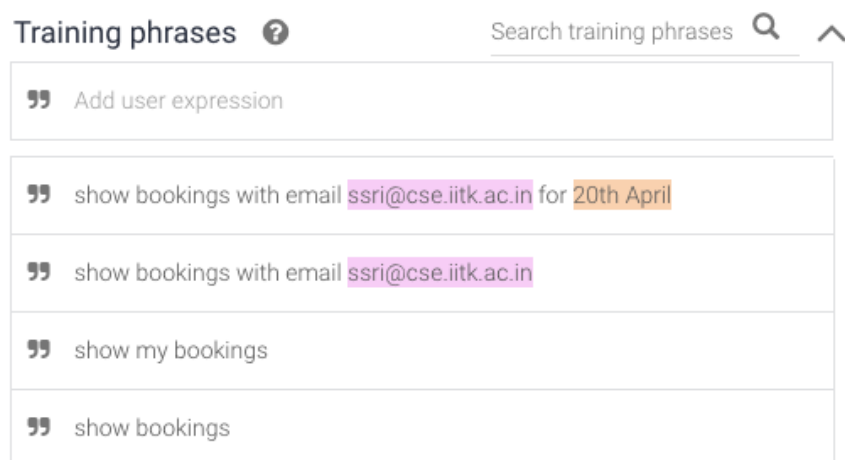
- For cancel Intent:

```
{
  "operation": "cancel",
  "parameters": {
    "bookingId": "1001",
    "email": "ssri@iitk.ac.in"
  }
}
```

- **Examples:**

The set of examples for both `show` and `cancel` intents are shown below:

- For show Intent:



The screenshot shows a 'Training phrases' interface. At the top, there is a search bar with the text 'Search training phrases' and a magnifying glass icon. Below the search bar is a list of phrases, each preceded by a double quote icon. The phrases are:

- ” Add user expression
- ” show bookings with email `ssri@cse.iitk.ac.in` for 20th April
- ” show bookings with email `ssri@cse.iitk.ac.in`
- ” show my bookings
- ” show bookings

– For cancel Intent:

Training phrases ? Search training phrases Q ^

- ” Add user expression
- ” cancel booking with booking id **1001** and email **ssri@cse.iitk.ac.in**
- ” cancel booking with booking id **1001**
- ” cancel my booking
- ” cancel booking

For both sprints, the coloured examples, show *tagged* values of Entities in user utterances. An example of values are tagged in Dialogflow is shown below:

Training phrases ? Search training phrases Q ^

” book a ticket on **EX-101** for **tomorrow** with email **ssri@cse.iitk.ac.in**

PARAMETER NAME	ENTITY	RESOLVED VALUE	
flightNumber	@FlightNumber	EX-101	X
date	@sys.date	tomorrow	X
email	@sys.email	ssri@cse.iitk.ac.in	X

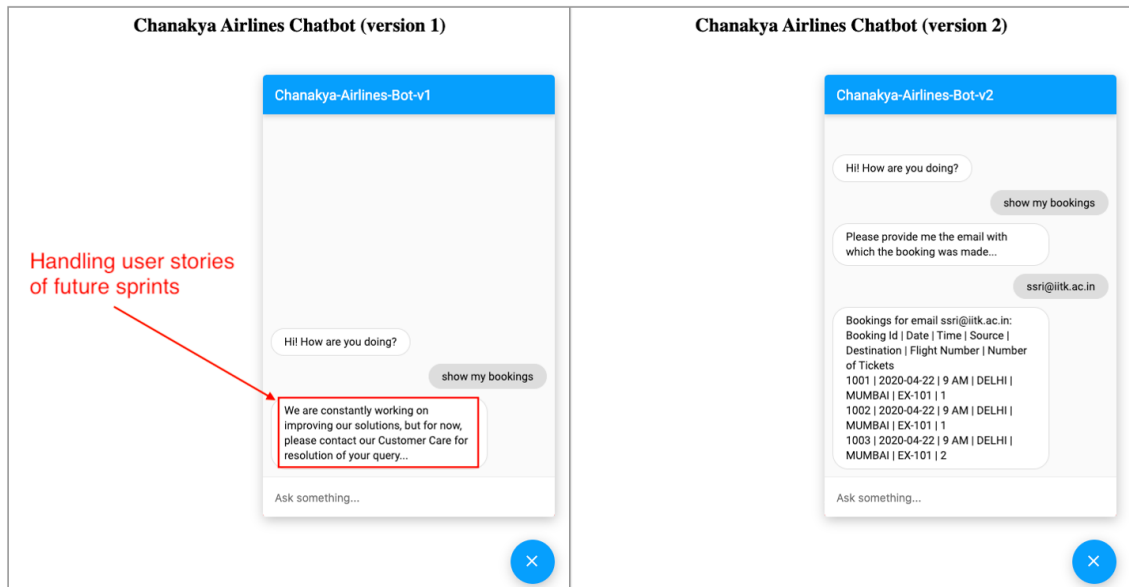
We refer to the deliverable built after the first sprint as *Chanakya-Airlines-Bot-v2*.

4.4.4 Comparison of Sprint Deliverables

Figure 4.6 and 4.7 show screenshots of the two versions of the Chanakya Airlines chatbot. The screenshots show the response of the two versions of the chatbot, on providing the same user queries.

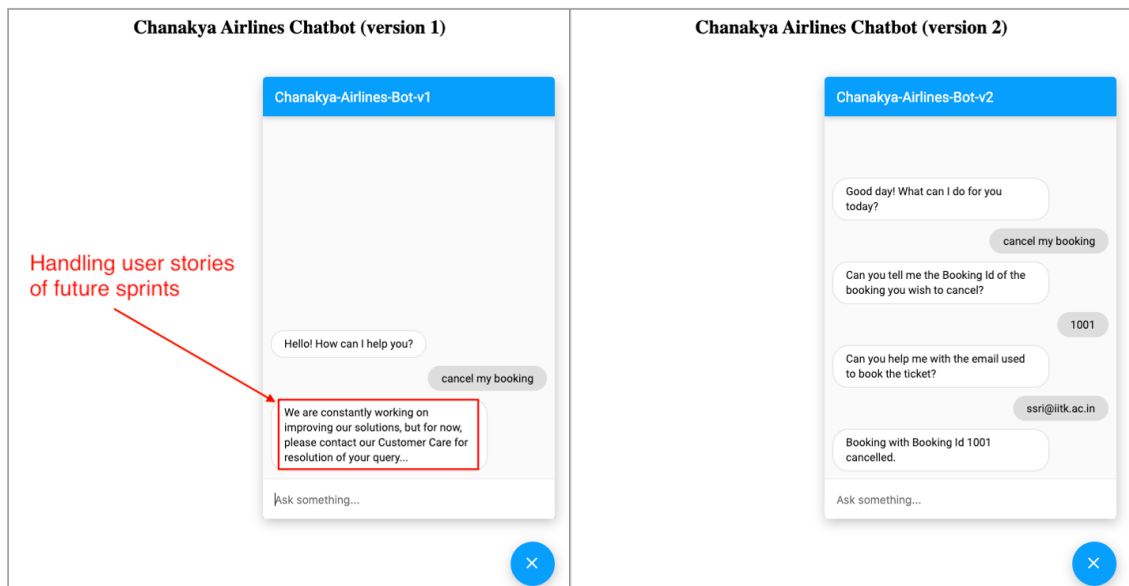
In particular, the differences are on two fronts:

Welcome to Chanakya Airlines



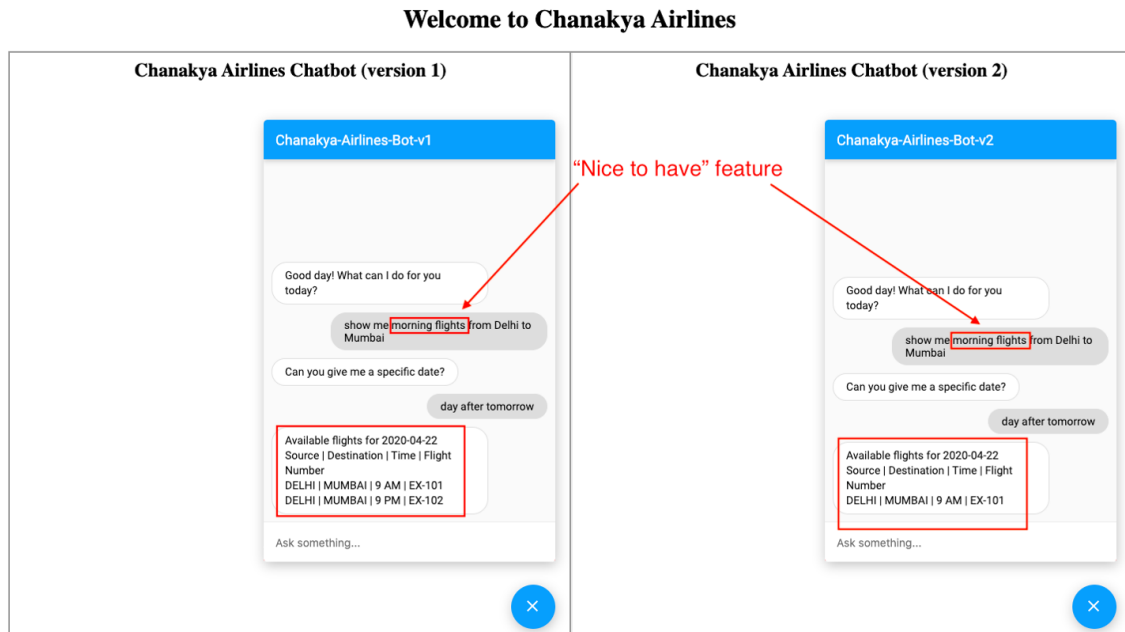
(a) Handling the queries related to the show Intent

Welcome to Chanakya Airlines

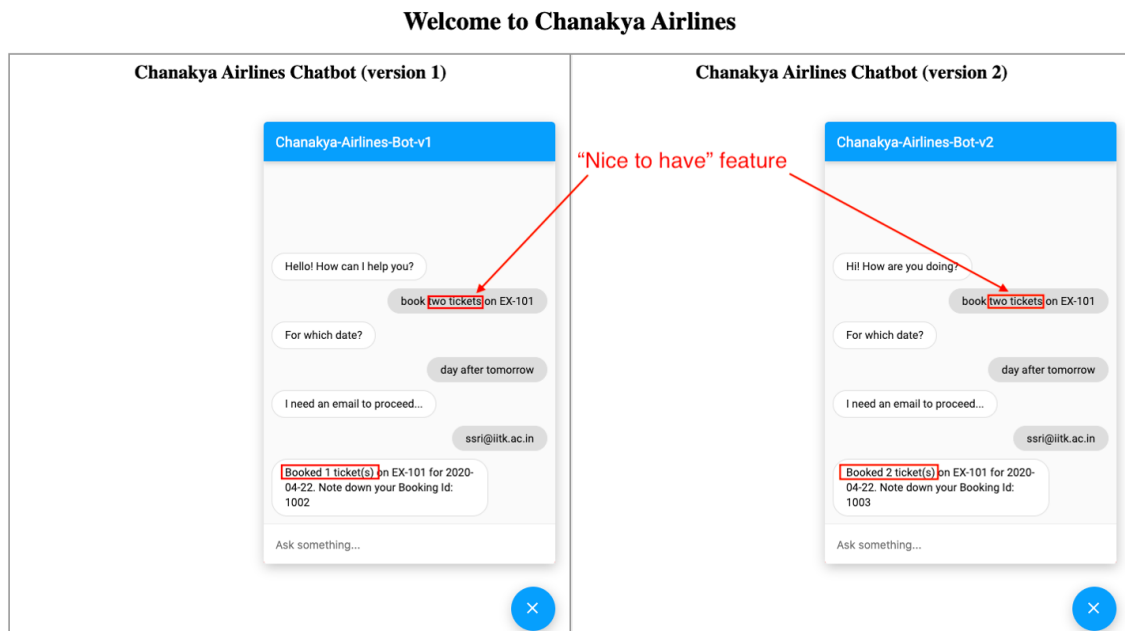


(b) Handling the queries related to the cancel Intent

Figure 4.6: Chanakya Airlines Chatbot: Handling use cases of future iterations



(a) Flights filtered on the basis of user's preference (day or night)



(b) More than one ticket can be booked as part of the same booking

Figure 4.7: Chanakya Airlines Chatbot: Updating implemented use cases

1. **Handling use cases of future iterations:**

The `others` Intent acts as a graceful handler for user stories that are not handled in the first sprint. It is meant to handle queries of both, `show` (Figure 4.6(a)) as well as `cancel` (Figure 4.6(b)) Intents.

2. **Updating existing use cases:**

Adding new features to existing Intents involve adding, removing or modifying Slot details, and providing more Examples, if required, to cover the set of queries that should be associated with the Intent. Figure 4.7(a) shows the updated `search` Intent, while Figure 4.7(b) shows the updated `book` Intent.

The configuration files for creating the Chanakya Airlines chatbot on the Dialogflow platform is available at [205]. The code that imitates the backend operations of Chanakya Airlines can be found at [206]. Both repositories are licensed under the MIT license [207].

4.5 Consequences

The Consequences section of a pattern discusses the outcomes of applying the pattern over the problem. In particular, it discusses the benefits of applying the pattern (say over any other possible solutions) as well as any limitations that cannot be mitigated by (or introduced by) its application.

- **Benefits:**

1. The *others* Intent can cover a large number of user stories in initial iterations, and its role in the chatbot operation decreases as new Intents are added in future iterations. The *default* Intent can provide graceful responses to cover imperfections in NLP tasks.
2. The chatbot-building platform requires only a limited number of real-world examples to train its models, and can extrapolate in multiple directions as required.
3. The logic associated with business operations can be executed as part of the processing pipeline. The details of the logic are abstracted and decoupled from the core NLP tasks performed by the chatbot.

- **Liabilities:**

1. The chatbot associates each query with one and only one Intent. A complex query which can be associated with more than one Intent cannot be handled.
2. The chatbot supplies the inputs to the processing pipeline and expects the response in a platform-specific format. It may require creating wrappers over existing cloud functions or REST API endpoints.

4.6 Summary

In this chapter, we provided insight into using chatbot-building platforms for creating chatbots. We aimed at coupling the development with common project constraints, such as multiple-iteration deliveries and fast-changing use cases.

We presented the *Contextual Reactive* pattern for defining chatbots. The pattern is used with some variations on most of the CaaS platforms. The idea is to define the chatbot in terms of Intents, Entities, Examples and Fulfilments.

- Intents group together queries of the same type.
- Entities are instances of real-world data that appear in conversations with a user.
- Examples are sample utterances that a user may say while interacting with the chatbot.
- Fulfilments are processing pipelines, which process a user query, and prepare a response to be relayed back to the user.

To handle the “uncertainty” associated with NLP tasks, two special Intents, the *others* Intent and the *default* Intent can be useful. The *default* Intent is used to trap queries that are “irrelevant” for a chatbot, i.e. the chatbot does not have an Intent which can cater to these queries. The *others* Intent can be configured to cater to any queries that are relevant for the chatbot but are not yet served (but maybe served in future iterations).

We then presented the form in which the pattern can be seen on three chatbot-building platforms - Google Dialogflow, IBM Watson Assistant and Amazon Lex. We also discussed a detailed case study of building a chatbot with the help of a chatbot-building platform, over two iterations. The code and configuration for the case study are available under the MIT license.

Chapter 5

Intent Sets

Chapter 2 was dedicated to introducing core elements of a chatbot, and its Containing system. In Chapter 3, we discussed the chatbot-building platforms in detail. We then described, in Chapter 4, the overall process of mapping a chatbot's use cases to a particular format, through which they can be expressed on a platform. In this chapter, we focus on another design choice related to building a chatbot, over a platform. We present the concept of *Intent Sets*. When a chatbot is defined on a platform, the developer essentially defines it using an Intent Set. This definition, however, is not unique in many cases. In other words, the same set of use cases can be defined in more than one way on a platform. Each of these ways, essentially represent an Intent Set for the problem at hand. Particularly, this chapter provides answers to the following two questions:

- Is the process of mapping a chatbot's use cases into the *Contextual Reactive* definition pattern unique? In other words, can the same set of chatbot use cases be mapped in two or more ways on a chatbot-building platform?
- If there are more than one ways to do so, are these options equivalent to each other, or do they affect the built chatbot significantly?

This chapter is organised as follows; We first present the concept of *Intent Sets* in Section 5.1. We then discuss the case study of a sample chatbot in Section 5.2, to show the difference between two Intent Sets. Section 5.3 discusses our observations related to a set of experiments conducted over the sample chatbot discussed in

Model	Manufacturer	Price	Available Colours
6i	Realme	175.58	White, Green, Blue
Galaxy M21	Samsung	222.40	Blue, Black
iPhone 11	Apple	672.58	Black, Green, Yellow, Purple, Red, White
A6 Plus	Samsung	219.99	Black, Gold, Blue, Lavender
Redmi 9	Xiaomi	119	Grey, Purple, Green, Pink/Blue
C15	Realme	169.99	Blue, Silver
S20	Samsung	679.99	Grey, Blue, Pink, White, Red
iPhone X	Apple	437.98	Grey, Silver

Table 5.1: An Example Table containing data related to phones in an inventory

Section 5.2. Section 5.4 presents some related works which can be pursued for further reading. Finally, we summarise the chapter in Section 5.5.

5.1 Intent Sets

As discussed in Chapter 4, the use cases that a chatbot has to serve, must be mapped to elements of a specific definition pattern, called the Contextual Reactive Pattern. Figure 4.1 shows an overview of the definition process. The process shown in the figure expects the developer to come up with a “set of Intentions”, which are roughly mapped to an Intent each on the platform. However, in general, this step of dividing the user queries into a set of Intents is not unique.

As an example, consider some data shown in Table 5.1. Assume that a chatbot is to be built to answer queries about the data shown in the table. We call it the *phoneBot*. What could be the Intents that could cover this scenario? For doing so, we should first collect a set of user utterances, that the chatbot may receive, e.g.

1. Do you have any phones for less than two hundred dollars?
2. Show me all the blue phones you have...
3. What Samsung phone do you have in stock right now?

The example queries, when seen in isolation, hint towards different directions to create Intents for the chatbot. Table 5.2 shows these hints and a possible set of

Query#	Hint	Possible Intents
1	“less than two hundred”	sub-200-query, upto-300-query, upto-500-query, beyond-500-query
2	“blue”	white-query, black-query, blue-query, other-colour-query
3	“Samsung”	apple-query, realme-query, xiaomi-query, samsung-query

Table 5.2: Possible Intents for the *phoneBot*

Intents that can be formulated with them.

For example, the query containing the phrase “less than two hundred” hints that the user may like to know about phones in specific price ranges. This can be done easily if the queries are classified into different intentions, based on the price range. One possible way is to create the Intents are shown in the **Possible Intents** column of the table. While it may be tempting to create the Intents based on the Price column, providing examples for training the chatbot may become tricky. For instance, there are theoretically, infinite number of (or even practically, a lot of) possible values that a user may put in her query when it comes to Price. It is, therefore, better to use Price as a Parameter, which can be instantiated as a Slot for some Intent, where it can take any numeric value as a legal input from the user.

Another possibility for creating Intents is given by the hint “blue”. This query highlights the importance of the Color column in data. Thus, Intents can be designed to classify a query based on the user’s choice of colour as well. While it is undoubtedly feasible to note down all the colours with which we have a phone, since Manufacturers often launch new colours, it may mean continuously adding more Intents. Another option could be creating Intents for popular colours, such as blue or black, and create an Intent for classifying queries of all other colours.

Compared to the other two options, the third query hints towards a more practical choice - the Manufacturer. Categorising queries based on the Manufacturer seems better than categorising them based on Price or Colour. First, Manufacturer is a nominal attribute, i.e. unlike Price, it only has a small set of fixed possible values. Second, since new Manufacturers do not enter markets fairly regularly, adding or removing Intents may not have to be done regularly.

The example we chose is relatively simple. Here, the background processing

<i>Intent Set_a</i>		<i>Intent Set_b</i>	
Intents	<ol style="list-style-type: none"> 1. apple-query 2. realme-query 3. xiaomi-query 4. samsung-query 5. any-other-query 	Intents	<ol style="list-style-type: none"> 1. white-query 2. black-query 3. blue-query 4. other-colours-query 4. any-other-query
Entities	<ol style="list-style-type: none"> 1. Phone-Colour 2. Phone-Price-Min 3. Phone-Price-Max 	Entities	<ol style="list-style-type: none"> 1. Phone-Manufacturer 2. Phone-Price-Min 3. Phone-Price-Max

Table 5.3: Two possible Intent Sets for *phoneBot*

required to answer a query is limited to doing a lookup in a table and using the result to craft a response. A chatbot which performs such a task is a type of Information Retrieval (IR) chatbot. We will present a case study of an IR chatbot in detail, in Section 5.2.

IR chatbots provide an intuitive understanding of the choices that a developer has when it comes to picking Intents for a chatbot. However, these choices are a part of every chatbot project, i.e. all the queries that a chatbot may encounter must be classified under some Intent. The set of Intents, thus, collectively cover all the use cases that a chatbot has to serve.

We can now provide a semi-formal definition for Intent Sets as:

An Intent Set is a collection of Intents and their associated Entities, which can collectively cover all possible queries that a chatbot may encounter.

An Intent Set, thus, is a collection of Intents and Entities, which can together cover all possible queries that can be fired at the chatbot. From the outset, it may seem a challenging task to come up with these sets. However, as discussed in Chapter 4, it is possible to define a chatbot through multiple iterations (with the help of an **others** Intent).

For example, two possible Intent Sets for the *phoneBot* are shown in Table 5.3. *Intent Set_a* classifies the queries into different sets based on the Manufacturer of the phones. If the user asks a query that doesn't mention any Manufacturer, the query is categorised in the set **any-other-query**. Any "additional" information that the user provides, such as a particular colour or price range of the phone, are captured

through Entities (which are instantiated as Slots with all the Intents). *Intent Set_b* classifies the queries into different sets based on the colour of the phone. Similar to *Intent Set_a*, if the user does not mention any colour in the query, it is categorised in the set `any-other-query`.

5.1.1 Properties of Intent Sets

We now list some properties of Intent Sets:

- *If the platform maps every query to the correct Intent, and parses all the Slot values correctly, then the response produced by any Intent Set for a given query is the same.*

The response produced by a chatbot depends on the processing pipeline. If the correct pipeline is invoked, with correct parameter values, the output should be correct as well. For the *phoneBot*, if we can correctly determine the choice of Colour, Manufacturer and Price Range from the user query, the response should also be correct (provided that the process which selects data from Table 5.1 does not have any bugs). However, the real challenge and “uncertainty” aspect that is added to a chatbot’s operation is due to the imperfection in recognising the correct processing pipeline for a given query. In other words, if the chatbot makes a mistake in identifying the associated Intent with a query, or fails to extract correct values for some Slots, the response will be incorrect as well.

- *An Intent Set divides the real-world into a finite number of disjoint scenarios, the union of which, constitutes all possibilities that the chatbot can handle.*

Intent Sets essentially partition the set of possible queries that a chatbot can receive, into *disjoint* sets. As discussed in Section 4.5, a limitation of the Contextual Reactive pattern is that it cannot handle queries belonging to multiple Intents. Intent Sets have the same limitation. Every possible query can only be part of one of the disjoint sets. There are no general rules-of-thumb to guide the process of picking one Intent Set over another. The decision could be affected by factors which are external to a chatbot. Take the example of *phoneBot*, if the products of different Manufacturers require making calls to

different API endpoints, it may be better to categorise the queries based on Manufacturers, as compared to Colour or Price.

Besides, there are two more properties, specific to Intent Sets for IR chatbots:

- *Only nominal attributes can be used to create Intent Sets.*

This is rather straightforward. Only an attribute that can take a fixed number of possible values can be used to create finite disjoint partitions of the data. If required, non-nominal attributes can be grouped into categories, by defining ranges. For example, for *phoneBot*, the Price attribute may be used for creating Intents, by defining ranges similar to those shown in Table 5.2. However, platforms usually provide better support modelling such attributes as Entities.

- *The queries that the chatbot can handle must be atomic with respect to the values for the attribute used to create the Intent Set.*

A common way to process an IR query over tabular data is to create a Relational Algebra (RA) expression [208]. The σ conditions for the query are derived from the Intent with which the query was associated (e.g. `Manufacturer = 'Samsung'`) as well as the values of other Slots (e.g. `Color = 'blue' \wedge Price < 200`). Since the chatbot associates any query with one and only defined Intent, the σ condition can have only one expression related to the attribute which was used for creating Intents. For instance, if *Intent Set_a* is used for *phoneBot*, the RA expression cannot contain two values for the Manufacturer attribute. Formally, if the RA expression that fetches the data looks like:

$$\Pi(A_1, A_2, A_3 \dots) \sigma(A_i = x \wedge A_j = y \wedge A_k = z \dots)$$

and, the Intent Set was created by using possible values for attribute A_i , then one and only one condition involving A_i could be part of the RA expression, meaning that the query can select rows with a single value of A_i only.

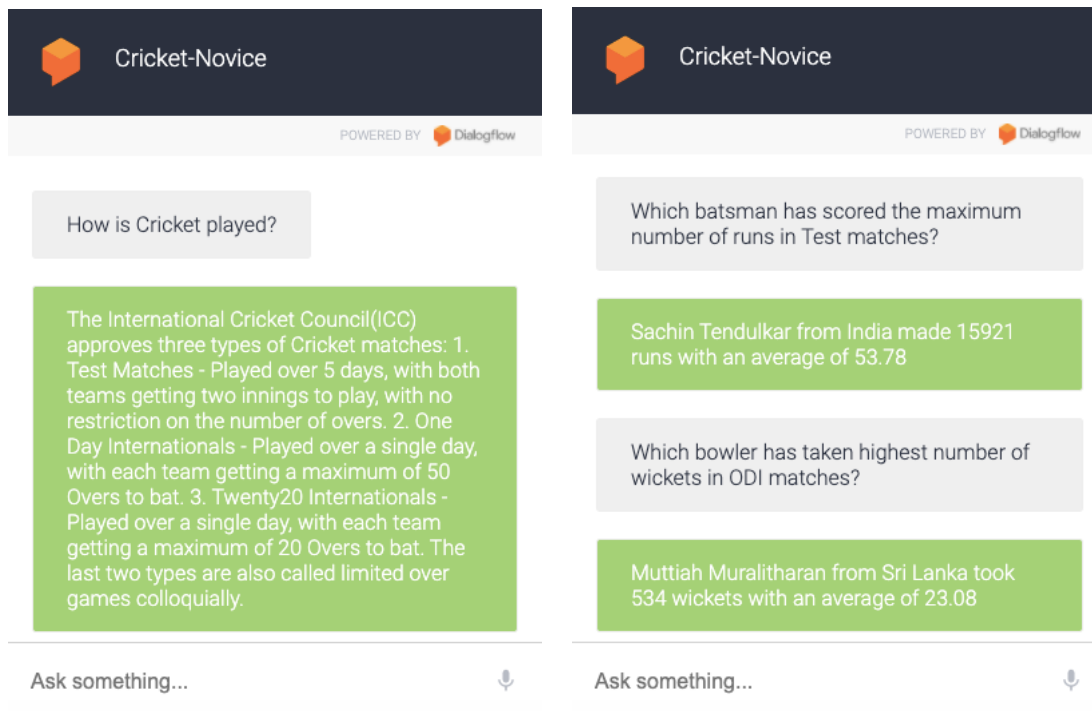
5.2 Case Study

The concept of Intent Set can be explained fairly easily. An Intent Set is a medium of definition for a chatbot over a platform. For the same chatbot, there may be more ways than one, to provide these definitions, and hence, there is a design choice involved in the process. However, to provide a more convincing proof of how this choice can affect the built chatbot, we performed a set of experiments. We picked a set of sample use cases to build and defined the chatbot using two different Intent Sets, on three different platforms. The sample chatbot we built is an IR chatbot about the game of Cricket that provides information from two data sources. First, there is a collection of explanatory text, which answers questions related to the rules and format of the game. Second, a table containing data about extraordinary individual performances in the game.

IR chatbots are good candidates to perform such experiments because of three reasons. First, the background processing for IR chatbots is usually not too complicated. It requires generating an RA expression or an equivalent statement in a querying language like the Structured Query Language (SQL) [209]. It allows using the same processing pipeline for the Intents with minor customisation. This means that the same Fulfilment endpoint can be used for all the Intents. Second, it is easier to come up with multiple Intent Sets for IR chatbots, by merely picking any nominal attribute from the dataset, and creating Intents for each possible value. The other attributes can then be added as Slots to these Intents. Since we needed at least two Intent Sets for the same example chatbot to perform the experiments, an IR chatbot seemed the right choice. Third, the performance of IR chatbots can be quantitatively analysed with relative ease. It can be done by inspecting the response of the chatbot over a carefully crafted set of queries. Since the queries seek specific data from the background sources, the response can be objectively evaluated as “correct” or “incorrect”. We can also define certain responses as “partially correct”.

The chatbot we built is named *Cricket Novice*. As mentioned before, it is supposed to answer queries related to the game of Cricket. Figure 5.1 shows a few query-response pairs for *Cricket Novice*. The two use cases for *Cricket Novice* are:

- Answer basic queries about the game of Cricket. This includes questions like “How is Cricket played?”, “How many players are there in a team?”,

(a) Answering *descriptive* queries(b) Answering *statistical* queriesFigure 5.1: Sample Conversations with *Cricket Novice*

“What are the formats in which Cricket is played?” and so on. The answers to these queries are *static* pieces of text, explaining the queried concept or regulation. From here on, we call such queries as *descriptive* queries, since they need a descriptive response. Figure 5.1(a) shows a sample conversation with *Cricket Novice* where the user asks a *descriptive* query.

- Answer queries related to individual performances in International Cricket matches. This includes questions like “Which batsman has scored the maximum number of runs in Test matches?” and “Which bowler has taken highest number of wickets in ODI matches?”. The answers to these queries are *dynamically* generated based on the information that is of interest to the user. From here on, we call these queries *statistical* queries, since the response is based on some statistics. Figure 5.1(b) shows a sample conversation with *Cricket Novice* where the user asks two *statistical* queries.

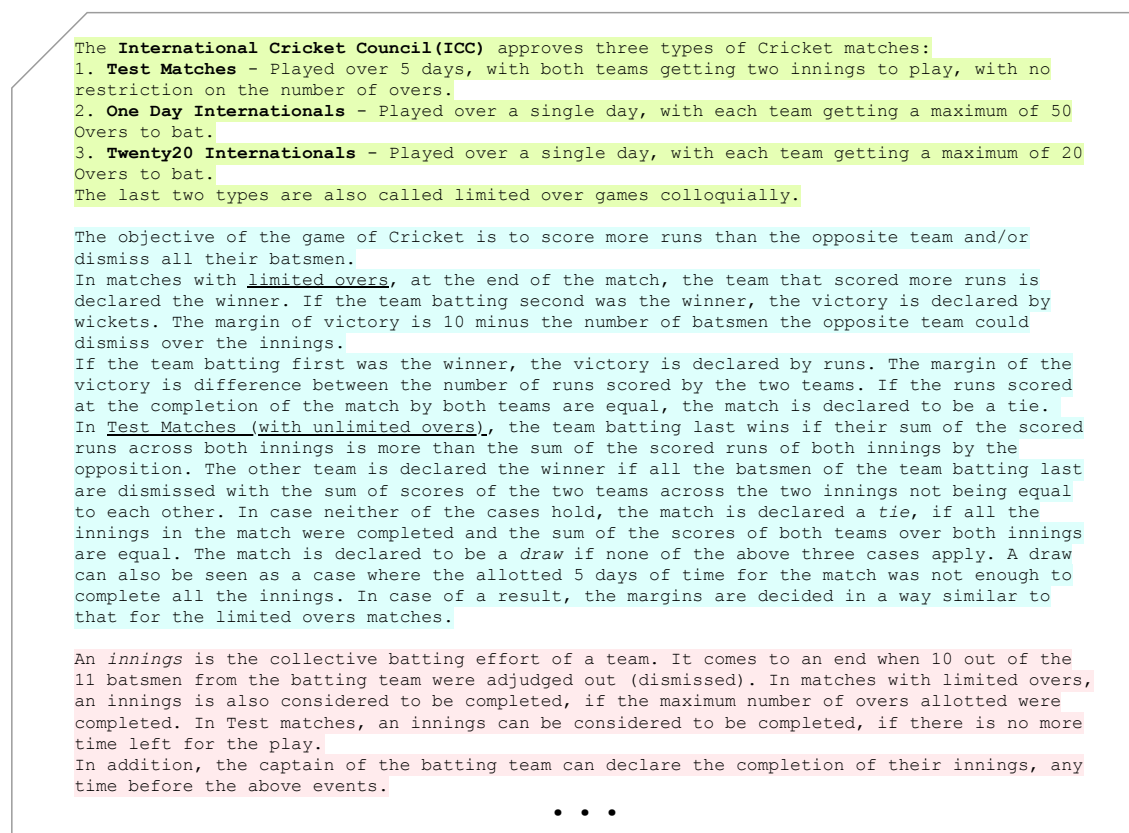


Figure 5.2: Snapshot of a document containing some basic information about the game of Cricket. Each paragraph can be read independently as a different document.

5.2.1 Experimental Setup

To study the impact of using a particular Intent Set on a chatbot's responses, we created a set of Intents to cater to selected *descriptive* queries, and another set of Intents, to answer *statistical* questions. We built multiple versions of the chatbot, and subjected them to a common, standardised set of user queries. We then compared the accuracy of these chatbots to understand the changes in their behaviour.

We now explain the experimental setup for the case study:

- **Descriptive data:** We created a document explaining the basics of the game of Cricket. Each paragraph of the document described some aspect of the game. A snapshot of the document is shown in Figure 5.2. Different paragraphs are highlighted differently, meaning that they can be read indepen-

MT	SS	ST1	ST2	S	P	T	D
test	innings	highest	batting	runs	Brian Lara	West Indies	scored 400* runs
odi	career	highest	batting	average	Ryan ten Doeschate	Netherlands	had an average of 67
odi	innings	highest	batting	strike rate	James Franklin	New Zealand	had a strike rate of 387.5
odi	career	highest	bowling	wickets	Muttiah Muralitharan	Sri Lanka	took 534 wickets
odi	career	lowest	bowling	strike rate	Rashid Khan	Afghanistan	bowled 2623 deliveries
test	career	highest	batting	average	Sir Donald Bradman	Pakistan	had an average of 99.94
t20	innings	highest	batting	strike rate	Dwayne Smith	West Indies	had a strike rate of 414.28
t20	career	highest	batting	centuries	Rohit Sharma	India	scored 4 centuries
t20	career	lowest	bowling	average	Rashid Khan	Afghanistan	has an average of 12.4
t20	career	lowest	bowling	economy rate	Daniel Vettori	New Zealand	had economy rate of 5.7
test	innings	best	bowling	figures	Jim Laker	England	had figures of 51.2-23-10-53
test	career	highest	batting	runs	Sachin Tendulkar	India	scored 15921 runs
test	career	highest	batting	centuries	Sachin Tendulkar	India	scored 51 centuries
test	career	highest	batting	double centuries	Sir Donald Bradman	Australia	scored 12 double centuries

...

MT - Match Type **SS** - Stat Span **ST1** - Stat Type₁ **ST2** - Stat Type₂
S - Stat **P** - Player **T** - Team **D** - Details

Table 5.4: Some rows from the table *c*. The Shaded columns are “output” attributes, used for preparing responses.

dently. There were a total of 25 paragraphs in the document, meaning 25 different *descriptive* queries could be answered using the document. Each paragraph acts like a response for one *descriptive* Intent.

- **Statistical data:**

We created a table containing some hand-picked data items. The data was taken from an online Cricket database called Statsguru [210]. We term this table *c*. Table 5.4 shows the schema and some sample rows in the table.

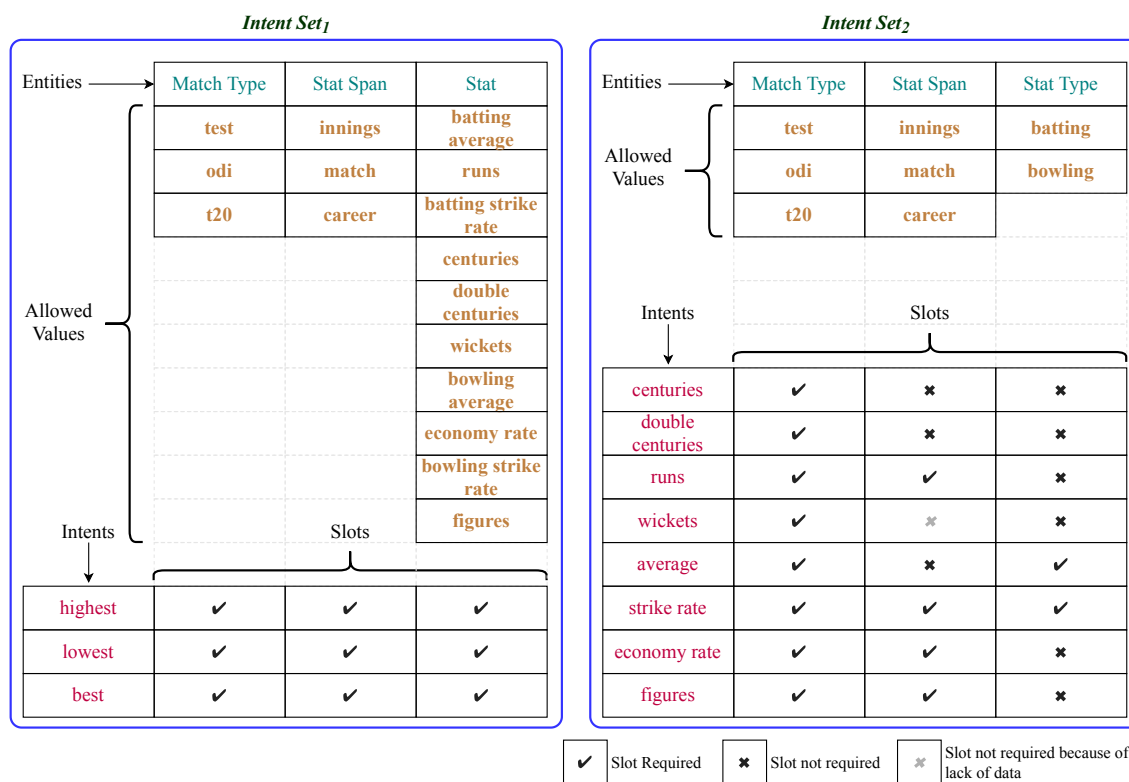


Figure 5.3: The *statistical* Intents for Cricket Novice. The *descriptive* and **default** Intents were common in both Intent Sets, and hence, not shown.

The table has a total of 8 attributes. Out of these, 5 attributes are “input” attributes, i.e. the values for these attributes are provided as input, by parsing a user query. The rest are “output” attributes, i.e. they are used to prepare a response for the user query. For our experiments, we assume that each *statistical* query attempts to fetch data from at most one row in table *c*. There were a total of 35 rows in *c*.

- **Intent Sets:**

We used two Intent Sets for defining *Cricket Novice*. We call them *Intent Set₁* and *Intent Set₂*. An overview of both the Intent Sets is provided in Figure 5.3. The Intents that we created to cater to the *descriptive* queries were common in both Intent Sets. We created 25 such Intents, each catering to a particular basic query about Cricket. The idea was to increase the number of Intents to a significant number so that the chatbot has a realistic problem when performing

Intent Classification (assuming that a typical commercial chatbot may have dozens of defined Intents). The two Intent Sets differed in the Intents that were meant to cater to the *statistical* queries. The Slots for the Intents also changed accordingly. Since we only had 35 rows in c , it meant that certain queries, even though valid, were not answerable due to lack of data. The Slots were created keeping these queries in mind. The **Stat Type** Entity in $Intent Set_2$, along with the triggered Intent, uniquely identifies the statistic that the user wishes to know. The **Stat** Entity in $Intent Set_1$ is actually a composite attribute, with respect to c , and provides the same information as the combination in $Intent Set_2$.

- **Training data:**

The training data for the chatbot comprised of Examples of how a particular query could be formulated by the user. For *descriptive* Intents, we used 3 to 5 alternative formulations for each Intent. The definition of a *descriptive* Intent involves providing these formulations, and a static response. For example:

```
"intent-name":"what-is-innings",
"questions":
[
  "What is an innings?",
  "What constitutes an innings?",
  "How does an innings end?",
  "For how long can a team bat?",
  "Is there a limit for how many overs a team can bat?"
],
"answer": "An innings is the collective
          ...
before the above events."
```

For the *statistical* Intents, we created three formulations each, for each row in the table c . These formulations belonged to different Intents in both Intent Sets, with values for different Slots embedded in them. Table 5.5 and 5.6 provide a glimpse of the training data for the *statistical* Intents.

Intent	Match Type	Stat Span	Stat	Question	Alternate 1	Alternate 2
best	test	match	bowling figures	Who has the best bowling figures in Test matches	Which bowler has the best figures in a Test match	Which player has the best bowling figures in Tests
best	odi	innings	bowling figures	Who has the best bowling figures in ODI matches	Which bowler has the best figures in a One Day match	Which player has the best bowling figures in One Dayers
highest	odi	innings	batting strike rate	Who has the highest batting strike rate in an innings of an ODI match	Which batsman holds the record for the highest strike rate in a One Day	Who has the best batting strike rate in an innings of an ODI
highest	t20	career	batting strike rate	Who has the highest batting strike rate in T20 matches	Which batsman has the best strike rate in Twenty-Twenty	Who has the best career batting strike rate in 20-20
lowest	t20	career	bowling average	Who has the lowest bowling average in T20 matches	Which bowler has the best average in Twenty-Twenty	Which player has the least bowling average in 20-20
lowest	test	career	bowling strike rate	Who has the lowest bowling strike rate in Test matches	Which bowler has the best strike rate in Tests	Which player has the least bowling strike rate in Test cricket

...

Table 5.5: A part of training data for the *statistical* Intents in *Intent Set₁*

Intent	Match Type	Stat Span	Stat Type	Question	Alternate 1	Alternate 2
average	test	career	batting	Who has the highest batting average in Test matches	Which player holds the record for the best batting average in Test cricket	Which batsman has the best average in Tests
centuries	odi	career	batting	Who has scored the highest number of centuries in ODI matches	Which player holds the record maximum centuries in One Day cricket	Which batsman has most centuries in ODIs
economy rate	t20	career	bowling	Who has the lowest economy rate in T20 matches	Who has the best career economy rate in Twenty-Twenty	Which bowler holds the record for the least career economy rate in 20-20
figures	t20	innings	bowling	Who has the best bowling figures in T20 matches	Which bowler has the best figures in a Twenty-Twenty match	Which player has the best bowling figures in 20-20
strike rate	odi	innings	batting	Who has the highest batting strike rate in an innings of an ODI match	Which batsman holds the record for the highest strike rate in a One Day	Who has the best batting strike rate in an innings of an ODI
wickets	test	career	bowling	Who has taken the highest number of wickets in Test matches	Who is the highest wicket taker in Test matches	Which player has taken maximum wickets in Test cricket

...

Table 5.6: A part of training data for the *statistical* Intents in *Intent Set₂*

In addition, we also used some *Negative Examples* to teach the chatbot certain queries, which it should not attempt to answer. Negative Examples are a way to train the chatbot towards invoking the processing pipeline associated with the default Intent. We used 10 Negative Examples for our experiments:

- 1 What is the highest score by any team in One Dayers?
- 2 What is the lowest total on which a team has been dismissed?
- 3 Which side has the best win ratio?
- 4 Which country has won their inaugural test match?
- 5 When did India play its first One Day match?
- 6 How many runs has Brian Lara scored in test cricket?
- 7 How many wickets does Glenn McGrath have to his name?
- 8 What is the best bowling figure in One Day cricket for Kapil Dev?
- 9 When did Sachin Tendulkar make his international debut?
- 10 How many times wickets did Shane Warne take while playing in Australia?

- **Platforms:**

We built *Cricket Novice* on three platforms - Dialogflow [27], Watson Assistant [21] and Lex [28].

5.2.2 Experiments

There were two experiments that we performed over *Cricket Novice*:

- We performed an *Accuracy Experiment* to compare the accuracy of the different versions of *Cricket Novice*. These versions were built using the two Intent Sets, across three platforms, meaning a total of six different versions. For each version, we evaluated the chatbots in four different phases:
 1. In the *first phase*, we only used one Example each with respect to all the rows in table *c* for training the chatbots. In terms of Tables 5.5 and 5.6, it meant that only the **Questions** column was used for training.

2. In the *second phase*, we added one more Example each, with respect to all the rows in table *c* towards the training of the chatbots. The total number of training Examples, thus became 70. In terms of Tables 5.5 and 5.6, it meant that the **Questions** and **Alternate 1** columns were used for training.
 3. In the *third phase*, we added one more Example each, with respect to all the rows in table *c* towards the training of the chatbots. The total number of training Examples, thus became 105. In terms of Tables 5.5 and 5.6, it meant that all three columns, **Questions**, **Alternate 1** and **Alternate 2** were used for training.
 4. In the *fourth phase*, we added the *Negative Examples* to the training set. Negative Examples are added to aid a chatbot’s understanding of “irrelevant” queries, increasing the chances of the matching of the **default Intent**.
- We also permed an *Order Experiment* on two out of the three platforms - Watson Assistant and Dialogflow. On these two platforms, the chatbot’s models are trained implicitly, as soon as there is a minute change in the training data (across all Intents and Entities). We performed another experiment to check if the training process is affected by the order in which changes are made to the training data. To do so, we did the following:
 - The *second phase* mentioned above supplied 70 training Examples to the chatbot. One way to do so is to add 35 more Examples to *phase one*, as mentioned above.
 - Another way to reach the same state is to remove 35 Examples from *phase three*, essentially leaving the chatbot with the same 70 training Examples.

In other words, we evaluated the response of the chatbot after *phase two* of training in two different ways. One, by adding Examples from the previous phase, and second, by removing Examples from the next phase. Ideally, with the same training data, the chatbots should behave the same. The *Order Experiment* was meant to investigate this.

During the experiments, we asked *Cricket Novice* 20 hand-picked queries, after each phase. The set of queries, as shown, remained the same across all experiments:

- 1 Who made the most runs in test cricket?
- 2 Who has taken maximum wickets in twenty twenty?
- 3 Which bowler holds the record for best figures in an innings of a Test match?
- 4 Which batsman has the best average?
- 5 Which bowler has the best average?
- 6 Can you name the player who has the best career average in ODIs?
- 7 Can you tell who scored maximum number of centuries in test cricket?
- 8 Which player has scored most double centuries?
- 9 Which batsman has the best strike rate?
- 10 Which bowler has the best strike rate?
- 11 I want to know which bowler has the best bowling figures.
- 12 What's the highest score by any batsman in ODIs?
- 13 How many centuries has Sachin scored?
- 14 How many ways can a batsman get out?
- 15 In test matches, which batsman has the highest strike rate?
- 16 Which bowler has the best economy rate in test matches?
- 17 Name the player with the best economy rate in an ODI.
- 18 What's the name of the guy who took most wickets in tests?
- 19 What's the highest team total in One Dayers?
- 20 What role does a captain play in cricket?

The “correct” answers to these queries were different for different phases. This was decided on the basis of training data. For example, if there was no possible training to answer a particular query, the “correct” behaviour was to match the default Intent. Also, if the required data was not provided as part of the query, the chatbot was supposed to *prompt* for the same.

We examined the diagnostic information provided by the Test consoles of the platforms, to note down the matched Intents and filled Slots for each query. During the *Accuracy Experiment*, we compared the responses with the expected responses. For the *descriptive* queries, the chatbot's response could either be termed as **failure** or **success**, depending upon the matched Intent for the query. For the *statistical* Intents, we also defined a state of **partial success**, where the chatbot could match the query to the correct Intent, but messed up something related to the Slot filling process. For each phase, we calculated a **score** based on the response of the chatbot to the 20 queries. During the *Order Experiment*, we defined the concepts of **match**, **mismatch** and **partial match**. We used another piece of diagnostic information provided by the platforms - Intent Match confidence. This score, a value on a scale of 0 to 1, shows the probability of the query being associated with the matched Intent. If the behaviour of the two versions matched exactly, i.e. the matched Intent, the confidence and the Slot filling process, were all the same, we termed it as a match. If the matched Intent differed, we called it a mismatch. If the matched Intent was the same, but there was a difference in some other aspect - confidence or Slot filling - we called it a partial match.

For the *Accuracy Experiment*, the scores were measured in **Utility Scores**. The Utility Score, U , for a phase was calculated as

$$U = \sum_{i=1}^{20} ra_i, \text{ where,}$$

$$ra_i = \begin{cases} 0, & \text{if the } i^{\text{th}} \text{ response is categorised as a } \mathbf{failure} \\ 1, & \text{if the } i^{\text{th}} \text{ response is categorised as a } \mathbf{success} \\ 0.5, & \text{if the } i^{\text{th}} \text{ response is categorised as a } \mathbf{partial success} \end{cases}$$

For the *Order Experiment*, the scores were measured in **Dissimilarity Scores**. The Dissimilarity Score, D , between the two versions of phase two was calculated as

$$D = \frac{1}{20} \sum_{i=1}^{20} rc_i, \text{ where,}$$

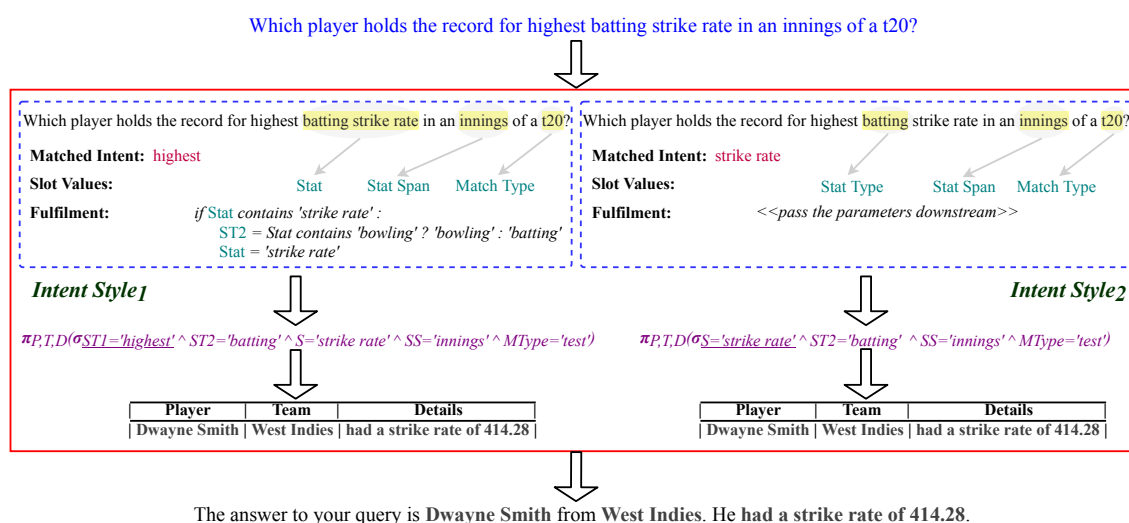


Figure 5.4: A rough sketch of processing of a user query when the two Intent Sets shown in Figure 5.3 are put in use

$$rc_i = \begin{cases} 0, & \text{if the } i^{th} \text{ response pair is categorised as a mismatch} \\ 1, & \text{if the } i^{th} \text{ response pair is categorised as a match} \\ 0.5, & \text{if the } i^{th} \text{ response pair is categorised as a partial match} \end{cases}$$

The overall idea behind conducting these experiments was to find out how picking different Intent Sets affect the behaviour of the same chatbot. Figure 5.4 provides a hint towards the background processing that *Cricket Novice* performs in order to prepare a response.

5.3 Observations

As mentioned, we conducted two major set of experiments with *Cricket Novice*. The *Accuracy Experiment* and the *Order Experiment*. We now present some observations from these experiments.

5.3.1 The Accuracy Experiment

We conducted an *Accuracy Experiment* on the different versions of *Cricket Novice*, across the platforms, as mentioned in Section 5.2.2. Figure 5.5 shows the comparison

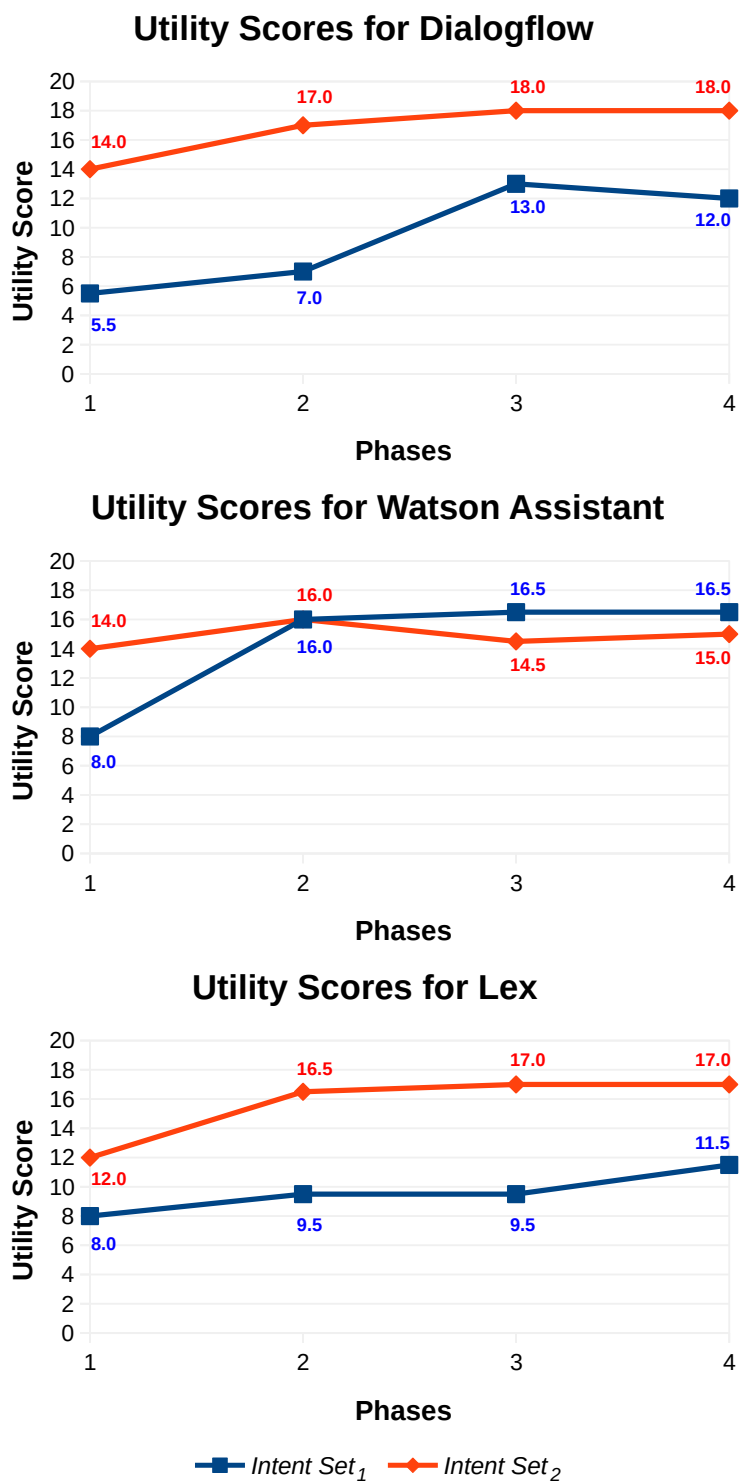


Figure 5.5: Utility Scores for *Cricket Novice* on different platforms

of the Utility Scores after the four training phases. The maximum Utility Score that any version could achieve was 20. The following observations can be made by looking at the data:

1. The first observation, which is expected, is that in general, the accuracy of the chatbot increased with increase in training data, with the only exception being the case of *Intent Set₂* for Watson Assistant, where the chatbot did better in the *second phase* as compared to later phases.
2. Even with the same training data, and the same set of probing queries during the experiment, the accuracy of the chatbots varied widely for *Intent Set₁*, and moderately for *Intent Set₂*.
3. On Dialogflow and Lex, *Intent Set₂* performed better in all phases. On Watson Assistant, except the *first phase*, *Intent Set₁* performed better than *Intent Set₂*.

We must point out that these experiments do not mean that either of the Intent Set is better than the other, or that some platform is a better choice over others. The only concrete observation here is that different Intent Sets can lead to significantly different chatbots. The other comment is that different platforms, due to their different “blackboxed” mechanisms, may do a better job at building a chatbot with different Intent Sets. It means that the utility of an Intent Set may be relatively higher or lower, depending upon the platform that is used for building the chatbots. Also, considering that *Intent Set₁* had 3 Intents, and *Intent Set₂* had 8 Intents, it is inconclusive if defining the chatbot use cases with more Intents is better or worse than doing so with fewer Intents. It is because we neither of the two Intent Sets performed better than the other across phases and platforms.

5.3.2 The *Order Experiment*

The purpose of the *Order Experiment* was to investigate the impact of providing the same training Examples for a chatbot, albeit, in a different order. It might not seem something that a chatbot should be affected with from the outset. However, some chatbot-building platforms, such as Watson Assistant and Dialogflow, retrain their NLP models with even a minor change in the Examples. Since the retraining

	<i>Intent Set₁</i>	<i>Intent Set₂</i>
Dialogflow	0.175	0.2
Watson Assistant	0.275	0

Table 5.7: Dissimilarity Scores on Dialogflow and Watson Assistant

is completed within a few seconds, it hints towards the use of some online learning procedure [54]. As mentioned in Section 5.2.2, we experimented with the chatbots for the two Intent Sets on Watson Assistant and Dialogflow, to understand if either one of them was susceptible to the different ordering of Examples.

The Dissimilarity Scores for the four scenarios is shown in Table 5.7. Dissimilarity score ranges between 0 and 1. The lower the score, the better it is. There is only one safe observation that can be made from the data - chatbots built on both platforms are susceptible to a change in Example ordering. There is nothing conclusive about either Intent Sets being more susceptible to this change over the other.

The detailed results, including the precise response that the chatbots gave for every query during the experiments, the prompts that were shown and the Intent Match Confidence are available at a git repository [211] under the MIT license [207].

5.3.3 Discussion

Although the process of creating NLP models for Intent Classification and Parameter Extraction is blackboxed, the results of the two experiments provide the following coarse observation about these processes:

1. There seems to be notable differences in the ML models produced on one platform, when compared to another. This can be observed from the Utility scores of the different platforms for exactly same training data (i.e. for the same Intent Set, and the same phase). This opens up an interesting route for further research - analysing the ML processes in the background, and predicting their suitability for a particular chatbot use case.
2. The result of the Ordering Experiment hints towards some ML process, which involves starting with a generalised model, and fine tuning it in small increments to cater to specific examples. Since it may involve minor changes in the model being made with minor changes in the training data (instead of a

complete retraining after each change, which may be too inefficient), the possibility of a method similar to BERT [212] seems highly likely, considering its popularity with NLP researchers [213].

These observations point to the lack of deterministic behaviour that these chatbots may exhibit for a given use case. This greatly enhances the responsibility of constant evaluation of the chatbot by testing its response against common user utterances. It also makes currently chatbots less reliable for critical use cases, and hence require additional efforts to make them more reliable for practical usage.

5.4 Related Work and Future Reading

The problem of finding Intents within a document containing information has been studied as a variant of *Topic Modelling* [214]. The idea of defining Intents by developers, falls under the category of *Supervised* Intent creation. There are attempts to do so from descriptive texts in *Unsupervised* and *Semi-supervised* settings as well (such as [214], [215], [216] and [217]). For structured data such as information stored in tables or RDF tuples, efforts with limited success have been attempted (e.g [218], [219], [220] and [221]). They take up the problem in a more abstracted form - attempting to provide conversational interfaces over structured data. Some platforms (such as [222] and [223]) also provide services to extract possible Intents automatically, if they are fed large amount of relevant business data.

For further reading, one can look at some articles which discuss the task of Intent creation in more detail, and talk about particular issues that a developer may face in such cases (e.g. the discussions at [224], [225] and [226]). There are some articles which specifically focus on techniques which can be used for finding Intents in an *Unsupervised* setting (e.g. [227] and [228]). One of the most common techniques used for the purpose is Latent Dirichlet Allocation (LDA) [229]. [230] provides a survey of LDA, with respect to Topic Modelling. For creating Intents associated with IR chatbots, the knowledge of Relational Algebra may be helpful. [231] and [232] provide an overview of the related concepts. The history of Natural Language interfaces for structured data, such as that stored in an RDBMS can be seen at [233].

5.5 Summary

This chapter concluded our contributions towards Architectural Issues associated with Chatbots. We discussed the concept of Intent Sets in this chapter. An Intent Set is a collection of Intents and Entities which can together cover all possible queries that a chatbot may receive. We showed that this set might not be unique, in which case, the developer will have to put a considerable amount of thought before picking one option over the other.

Intent Sets have two main properties. First, in an “ideal” scenario, i.e. when the chatbot never makes a mistake while Intent Classification or Parameter Extraction, it does not matter which Intent Set is chosen to define the chatbot since they are all equivalent. However, since the modern-day chatbots are still not perfect, different Intent Sets provide different behaviours. Second, Intent Sets divide the set of queries that a chatbot may receive, into disjoint sets. It means that every query must be put in one, and only one of the sets. Each of these sets represents a chatbot Intent.

To show the impact of picking one Intent Set over another, we built a chatbot called Cricket Novice, which could answer certain basic queries related to the game of Cricket. We used three platforms - Dialogflow, Watson Assistant and Lex - to build the chatbot, with two different Intent Sets. Our experiments showed that the behaviour of the chatbots built with the two Intent Sets differed significantly from each other, even when the training data provided to both was the same. This confirmed our hypothesis that picking an Intent Set is a design choice for a chatbot, which can affect its behaviour significantly.

We also inspected a particular version of the chatbot built over two platforms - Dialogflow and Watson Assistant - to see if the order in which training Examples were provided, made any difference to the built chatbot. The results showed that both platforms’ model building procedure is sensitive to Example ordering.

Chapter 6

Conclusions and Future Work

This chapter provides a brief Conclusion in Section 6.1 for this thesis, and suggest some future directions for extending the current work in Section 6.2.

6.1 Conclusions

Although Natural Language Processing (NLP) is a field that has seen contributions for many decades, there are still gaping holes needing to be patched. Despite all these constraints, chatbots are becoming a popular choice among businesses to interact with the user. The imperfections in NLP techniques mean that there should be safety-nets in place. In other words, whenever a chatbot is built, the developers must have mechanisms to deal with failures. Handing over the control to a human, or, saying sorry to the user are two common alternatives usually available. This also means that during the design of the chatbot, the Software Architect may have to choose between attempting a lot and often failing versus trying something limited, and doing it better most of the time. It is a choice that needs to be made separately for each project, depending upon the seriousness of the use cases and the cost of erring.

The NLP techniques, whatever is their State-of-the-art, are arduous to implement and maintain. This is why building a chatbot from scratch is a daunting task. The developers may end up spending more time on perfecting their NLP libraries, than building the chatbot on top of them. Employing a chatbot-building platform,

thus, seems a wise idea. They provide most of the basic NLP tasks - such as classifying user queries into appropriate classes or parsing named entities from phrases - as “blackboxed” services. However, for some use cases, where sharing data with another platform is not allowed, the developers are left with no option other than building the chatbot from scratch.

This thesis work is dedicated towards enriching the body of knowledge related to chatbots. We start by looking at a chatbot the way we see any other software component or subsystem. We discussed its major constituents and how they interact with each other. We also envisioned the chatbot as part of a larger, Containing system, and discussed its relationship with its environment. Due to the overbearing load of managing NLP tasks, we argued that using a platform for building a chatbot is the right direction for most use cases. We, therefore, discussed aspects related to categorisation and evaluation of these platforms. We also discussed the definition of use cases on these platforms and the design choices that emerge due to their use. Throughout the thesis work, we focused on three popular chatbot-building platforms - Google Dialogflow, IBM Watson Assistant and Amazon Lex, for performing any case studies to support our intuitions.

A brief discussion of the contributions of this thesis is provided below:

- We presented the major *constituents* of a typical chatbot as well as its Containing System. We also presented a Reference Architecture for an application with a conversational interface.

Chapter 2 was dedicated to dissecting a chatbot and showing its significant elements. Out of all the elements, the most important parts are the Intent Classifier, which maps a query with a pre-defined intention and the Parameter Extractor, which parses any named entities in user utterances. In the Containing system, the most prominent parts for a chatbot are the Fulfilment hooks, which are used to process a user query in the background, and the front-end interfaces, which put constraints on what a chatbot can and cannot do. We also presented case studies over the three platforms mentioned before, to show the Concrete Architectures of an application with conversational interface, assuming a platform is used to build the conversational components. These Concrete Architectures showed the effects of a platform on the architecture of

the Containing system.

- We discussed the chatbot-building platforms in detail. This included a categorisation based on the commercial offerings, a compiled list of their desirable features, and application of the Hospitality framework over them.

In Chapter 3, we presented the details of chatbot-building platforms. We first showed how the platforms can be divided into three categories - NLP-as-a-Service (NLPaaS), Conversation-as-a-Service (CaaS) and ChatWidget-as-a-Service (CWaaS). We then presented a list of desired features on the CaaS platforms that can ease the job of a developer. The Hospitality Framework was first applied to cloud platforms to compare them from a Quality Attributes perspective. We showed how the framework is generic enough to be re-purposed for evaluating chatbot-building platforms. The case studies presented in the chapter showed the application of the framework over a simple, fruit-selling store chatbot, as well as a report on the current support of the desirable features on the three platforms mentioned before.

- We discussed a pattern used by most chatbot-building platforms to allow the definition of a chatbot. The pattern called the *Contextual Reactive pattern* adds a set of design decisions to the overall process of implementing the chatbot use cases.

Chapter 4 presented a pattern. Patterns are usually described in a specific format. They define the problem that they address, enlist the constraints that must be met, discuss the reasons to pick a particular solution and then describe the solution in detail. For the Contextual Reactive pattern, the major driving forces include allowing the development of the chatbot in phases over a period of time and keeping the defined chatbot decoupled from the business logic it uses for processing the queries. With the help of a case study - a chatbot for a fictitious, Chanakya Airlines - we showed how the pattern achieves both of its major goals.

- We presented the concept of an *Intent Set*, a critical design choice for building a chatbot over a platform. We discussed how designing Intent Sets affects the built chatbot significantly.

We discuss a particular design choice in Chapter 5, which emerge because of the use of Contextual Reactive pattern on chatbot-building platforms. Intent Sets are the means for defining a chatbot on a platform. However, in many cases, there may be more ways than one, to define a chatbot for the same set of use cases. In these cases, the developer must spend more time on evaluating candidate Intent Sets before defining the chatbot. Selecting one Intent Set instead of another can have a significant impact on the built chatbot. We present a case study which confirms this hypothesis. The case study involved using two Intent Sets for building a chatbot, called the Cricket Novice, for answering the same set of queries related to the game of Cricket. Performed over the three platforms mentioned above, the case study confirmed significant variations in the chatbot's behaviour with a change in Intent Set.

Overall, we can conclude the following from the work presented in this thesis:

1. Whether being built from scratch, or using a platform, understanding the different constituents of a chatbot, and how it affects the architecture of its Containing system is crucial. In particular, it must be kept in mind that the current state-of-the-art in the field of chatbots, still leave a lot of imperfections, which must be handled appropriately by its Containing system.
2. For most chatbot projects, using a platform at some level of development seems appropriate. If the chatbot's use cases are well-defined, and are fairly common (such as troubleshooting or customer support), CWaaS platforms may provide a ready-to-use solution, deployable on common mediums such as Messenger or Slack. For most projects, CaaS platforms provide the right level of development flexibility (i.e. allowing the developer to implement a wide range of custom use cases), while providing significant abstractions to hide the underlying details (i.e. how the core NLP tasks are being performed in the background). For certain scenarios, using a NLPaaS platform may also be useful, especially when the chatbot has to cater to complex queries involving multiple Intents.
3. Different chatbot-building platforms may be better for different projects, however, there are no rules of thumb to decide upon the best alternative. Our anal-

ysis showed that the same Intent Set can provide widely varying results, when used for defining the chatbot on a different platform. Also, while some of the core features required for chatbot development are provided by all, there are features which are not omnipresent. Therefore, it requires a deeper analysis with all stakeholders in loop, to find the right platform for a given project.

6.2 Future Work

The overall idea of this thesis is to initiate formal work in the field of architectural knowledge related to chatbots and AI-intensive systems in general. We now present a few dimensions which may lead to more value addition in that perspective.

1. **Extending and updating the Desired Features list with Architectural Annotations:**

The presented list of desired platform features was compiled as a result of the application of the Hospitality Framework to chatbot-building platforms. It may be argued that the list, thus, may not cover all the features that may be desirable for implementing certain chatbot use cases. Also, since the chatbot-building platforms are still evolving, there may be many more features which may not have been envisioned yet. The list, thus, is an ongoing effort which needs to be updated periodically.

Another aspect that may render the list as a more useful architectural tool, is enriching the features with architectural annotations. The Hospitality framework requires mapping a Quality Attribute to a set of Tactics, which in turn are mapped to a set of desired features from the platform. It would thus be nice to re-look the list from a pure Software Architectural perspective, linking each feature to the support for one or more Quality Attributes or Tactics. It will help a Software Architect analyse a candidate platform by simply inspecting the availability of a shorter list of appropriate features.

2. **Building a Pattern Language for AI use cases definitions:**

The Contextual Reactive pattern is a neat way to provide training data to a platform for building chatbots. The idea is to define a stimulus, and teach the

chatbot a response for the same. Chatbots are not the only examples of AI components. There may be variations of the pattern which are employed on platforms for building components that perform other AI activities such as Object Detection, Speech Transcription and Spam Filtering. A Pattern Language attempts to capture supplementary and complementary solutions in a small problem domain. In simple terms, a Pattern Language that contains the Contextual Reactive Pattern will probably contain - (i) other definition patterns that are used on some chatbot-building platform or during the development of the chatbot from scratch; (ii). any patterns that can be applied before or after the application of Contextual Reactive Pattern to either mould or modify the training data. Exploring a possibility to come up with the collection of these variations could be an interesting challenge.

3. Suggesting Intent Sets for a chatbot:

We have shown how picking one Intent Set over another can affect the accuracy of the built chatbot significantly. We also discussed a few properties of Intent Sets. However, more research is required to understand the relationship between an Intent Set and the chatbot's behaviour. Picking an Intent Set is an important design issue in the chatbot development phase, as it affects the overall quality of the built chatbot. The fact that Intent Sets are platform-sensitive, makes this job even harder. The ideal path for this analysis may involve knowing the details of the "blackboxed" solutions provided by CaaS platforms, and evaluating the impact of different variations of the same training data on the underlying model. Assuming that this may not be possible for these platforms to allow so due to business constraints, the analysis becomes challenging. It would involve formulating a number of Intent Sets for the same problem, build different versions of the same chatbot, and analyse their behaviour minutely. An overall problem that is worth exploring is coming up with a methodology or framework to comment upon different Intent Sets for a chatbot, hence providing design hints for the development.

However, since most of the platforms do not reveal the details of their model-building process, it is not possible to provide a general analysis framework to comment on the efficacy of an Intent Set for a given chatbot. However,

suppose these platforms initiate individual internal projects. In that case, it may be possible to suggest the developers one or more “good” Intent Sets, provided that they are willing to share more business data with them.

Appendix A

Guide to Privacy Policy Resources of Selected Platforms

Platform	Useful Links for understating Privacy Policy
Watson Assistant	<i>Summary:</i> http://www.sharelatex.com <i>Detailed:</i> https://www.ibm.com/watson/assets/duo/pdf/Watson-Privacy-and-Security-POV_final_062819_tps.pdf
Snatchbot	<i>Specific to GDPR:</i> https://snatchbot.me/gdpr <i>Detailed:</i> https://snatchbot.me/privacy
Lex	<i>Summary:</i> https://docs.aws.amazon.com/lex/latest/dg/data-protection.html <i>Some more details:</i> https://aws.amazon.com/lex/faqs/#Data_and_Security

continued ...

... continued

Platform	Useful Links for understating Privacy Policy
Dialogflow	<p><i>Terms of use:</i> https://cloud.google.com/dialogflow/docs/data-logging-terms</p> <p><i>Google's General Privacy Policy:</i> https://cloud.google.com/dialogflow/docs/data-logging-terms</p>
Chatbot.com	<p><i>Specific to GDPR:</i> https://www.chatbot.com/legal/gdpr-faq/</p> <p><i>Detailed:</i> https://www.chatbot.com/legal/privacy-policy/</p>
Azure Bot Service and LUIS	<p><i>Specific to GDPR:</i> https://blog.botframework.com/2018/04/23/general-data-protection-regulation-gdpr/</p> <p><i>Data Storage Details in LUIS:</i> https://docs.microsoft.com/en-us/azure/cognitive-services/luis/luis-concept-data-storage</p> <p><i>Microsoft Azure's General Privacy Policy:</i> https://azure.microsoft.com/en-in/overview/trusted-cloud/privacy/</p>
kore.ai	<p><i>Acceptable Use Policy:</i> https://kore.ai/acceptable-use-policy/</p> <p><i>Privacy Policy:</i> https://kore.ai/privacy-policy/</p>

continued ...

... continued

Platform	Useful Links for understating Privacy Policy
Wit.ai	<p><i>Frequently Asked Question:</i> (see answers to “Does Wit own my data?”, “Do you comply with the GDPR?” and “Is Wit.ai Privacy Shield certified?”)</p> <p>https://wit.ai/faq</p> <p><i>Terms of Service:</i> (see “Data Processing Addendum”)</p> <p>https://wit.ai/terms#16</p> <p><i>Privacy Policy:</i></p> <p>https://wit.ai/privacy</p>
Yellow Messenger	<p><i>Privacy Policy:</i></p> <p>https://yellowmessenger.com/privacy-policy</p>
IntelliTricks Drift	<p><i>Data Processing Agreement:</i></p> <p>https://www.intelliticks.com/data-processing-agreement/</p> <p><i>Privacy Policy:</i></p> <p>https://www.intelliticks.com/privacy-policy/</p> <p><i>Specific to GDPR:</i></p> <p>https://www.drift.com/gdpr/</p> <p><i>Privacy Policy:</i></p> <p>https://www.drift.com/privacy-policy/</p>

Appendix B

Explanations for Selected Values in Table 3.12

(A) Lex only expect placeholders instead of actual values, e.g. a training example in Lex looks like “Book a ticket from <i>\$source</i> to <i>\$destination</i> ”, as compared to say a tagged example in Dialogflow or Watson Assistant, e.g. “Book a ticket from Delhi: <i>source</i> to Mumbai: <i>destination</i> ”.
(B) In Watson Assistant, although an explicit value cannot be specified, an if condition in the Dialog Tree can handle it. No direct or indirect mechanism found in Lex for doing the same.
(C) In Watson Assistant, it cannot be done explicitly but can be done by tagging an example <i>irrelevant</i> in the dashboard or passing it indirectly by restoring a skill from a JSON file. No direct or indirect mechanism found in Lex for doing the same.
(D) Applicable to Contextual Entities - Watson Assistant tries to match values with the context, and there is no way to stop it from adding a new value to the set of existing, pre-defined values.

continued ...

... continued

<p>(E) In Lex, updating a Parameter is not possible directly. The Parameter must first be added to an Intent, only then can it be edited. The chatbot developer may find the interface for adding/modifying synonyms confusing. For example, if an entity value had only one synonym, deletion of the synonym is not allowed in the editing popup. The workaround is to delete the value itself, and add it again (without any synonym).</p>
<p>(F) Lex does not provide a way to provide a static response straightaway to the user. A fulfilment step must be completed (either by invoking a Lambda function or returning the parameters back to the client). Only after the completion of the fulfilment, can a static response be shown.</p>
<p>(G) Dialogflow allows rich responses for a set of target platforms (e.g. Facebook Messenger or Slack). A simple image response cannot be sent. It must be configured differently for different platforms.</p>
<p>(H) In Dialogflow, static placeholders can be added in the response, e.g. “Your \$product has been shipped”, but no mechanism for implementing conditions. In Watson Assistant, simple conditions can be handled using the conditional (?) operator, while the Dialog Tree can be used for complex use cases. In Lex, a separate Lambda function can be configured for “initialization and validation”, but no mechanism for implementing conditions.</p>
<p>(I) In Lex, if the chosen option is “supplied values”, synonyms are ignored. It might be preferable to get the reference value whenever supplied value matches the reference value (or provided synonyms) exactly, and get the supplied value as fallback. In Watson Assistant, the reference value is provided by default (with expression <code>@entity</code>), whereas the supplied value can be retrieved by adding the literal property (with expression <code>@entity.literal</code>). Dialogflow has a similar case. The property for supplied value is <code>original</code> (instead of <code>literal</code>).</p>

continued ...

... continued

<p>(J) Dialogflow has a timeout of 5 seconds, whereas Watson Assistant has a timeout of 8 seconds. Lex can only redirect flow to an external caller, if this is how the conversation was initiated. The only other option is to invoke a Lambda function.</p>
<p>(K) Dialogflow allows writing inline fulfillment code, which is deployed as a Google Cloud Function. Only Node.js can be used for this purpose. Watson Assistant can connect to any IBM Cloud Function, and Lex can connect to any AWS Lambda function, which in turn can be implemented in any language (some languages are supported natively, while others can be supported via docker images and runtime APIs).</p>
<p>(L) Dialogflow: Telephony - Dialogflow Phone, Avaya, SignalWire, Voximplant, AudioCodecs, Genesys Cloud; Text based - Web Demo Widger, Dialogflow Messenger, Facebook Messenger, Slack, Viber, Twitter, Twilio IP, Twilio SMS, Skype, Telegram, Kik, Line, Cisco Sparc, Amazon Alexa Watson Assistant: Webchat Widget, Salesforce, Zendesk, Intercom (Premium Accounts) and Facebook Messenger, Slack, Voice Agent (Telephony) Lex: Facebook Messenger, Kik, Slack, Twilio SMS</p>
<p>(M) Dialogflow added support for it in the June 13, 2019 release and updated it in the February 20, 2020 release. Watson Assistant provides support for creation and modification of skills through only v1 API [234]. The v2 API only provides methods to interact with the assistant. A “workspace” in v1 API is equivalent to a “skill” in v2 API [235]. Lex provides a well-documented user guide to access its AWS CLI for Lex related tasks [236].</p>
<p>(N) In Lex, while context variables can be used across Intents, they cannot be set in the UI. They must be set in the responses sent by Lambda Functions.</p>
<p>(O) Dialogflow offers many “follow-up Intents” such as Yes, No, Previous, Next, Cancel etc. Lex provides Intents through the Alexa Skill set (except Yes and No Intents). Watson Assistant doesn’t provide such features, but the same can be done via context variables in the Dialog Tree.</p>

continued ...

... continued

(P) Watson Assistant provides a dedicated feature to handle digressions. In Dialogflow, some cases of digressions may be handled through a complex setting and unsetting of input and output context variables [237]. In Lex, no specific feature is provided to handle digressions, nor do they advertise if setting different context variables will change the behaviour of the Intent matching algorithm.

(Q) The Dialog Tree in Watson Assistant may be able to handle multiple-intent queries, but getting it right is fairly difficult [160]. Dialogflow and Lex does not have any visible mechanism to handle multiple-intent queries.

Appendix C

Comparison of the Elements of *Contextual Reactive Pattern*

	Dialogflow	Watson Assistant	Lex
Intents	<ol style="list-style-type: none">1. Intent names can be anything without space2. Two Intents are added automatically - Welcome and Default Fallback Intent	<ol style="list-style-type: none">1. Intent names must start with a #2. Every Intent must be created or added by the developer	<ol style="list-style-type: none">1. Intent names can be anything without space2. Every Intent must be created or added by the developer

continued ...

... continued

	Dialogflow	Watson Assistant	Lex
Entities	<ol style="list-style-type: none"> Entity names can be anything without space Details of the presence of the Entity in any example cannot be seen on the definition page No implicit version control 	<ol style="list-style-type: none"> Entity names must start with a @ Details of the presence of the Entity in any example across Intents are shown in an <i>Annotations</i> tab on the definition page No implicit version control 	<ol style="list-style-type: none"> Entity names can be anything without space Details of the presence of the Entity in any example cannot be seen on the definition page Entities are implicitly version controlled - every time a change is made to an Entity, a new version is created Entities are called "Slot Types" in Lex
Slots	<ol style="list-style-type: none"> Slots are called <i>Parameters</i> Slots are defined on the Intent definition page 	<ol style="list-style-type: none"> Slots are defined in the Dialog tree 	<ol style="list-style-type: none"> Slots are defined on the Intent definition page

continued ...

... continued

	Dialogflow	Watson Assistant	Lex
Examples	<p>1. Examples are called <i>Training Phrases</i></p> <p>2. Values associated with any Entity in the examples, are explicitly tagged, e.g. Search for a flight <u>tomorrow</u></p> <p style="text-align: center;">↑</p> <p><i>journey-date</i></p>	<p>1. Examples are called <i>User Examples</i></p> <p>2. Values associated with any Entity in the examples, are explicitly tagged, e.g. Search for a flight <u>tomorrow</u></p> <p style="text-align: center;">↑</p> <p><i>journey-date</i></p>	<p>1. Examples are called <i>User Utterances</i></p> <p>2. Lex does not expect actual values of Entities in the examples, but a placeholder for them shall be provided explicitly, e.g. Search for a flight {<i>journey-date</i>}</p>
Fulfilments	<p>1. Simple responses can be configured on the Intent definition page</p> <p>2. For invoking external business logic, a common fulfilment webhook could be defined for the chatbot</p>	<p>1. Responses can be configured in the Dialog tree for many complex scenarios</p> <p>2. Fulfilment logic can be configured at external API endpoints or IBM Cloud functions [81]</p>	<p>1. Simple responses can be configured on the Intent definition page</p> <p>2. Fulfilment logic can be configured as AWS Lambda functions [82]</p>
Other Notable Aspects	<p>1. Dialogflow provides a set of pre-configured Intents, such as those for having a casual conversation</p> <p>2. External business logic must be exposed through a single API endpoint</p>	<p>1. Watson Assistant provides a set of pre-configured Intent <u>group</u>, known as <i>Skills</i></p> <p>2. Different Intents can invoke different API endpoints</p>	<p>1. Lex provides a set of pre-built intents, mostly useful for playing media (e.g. <code>PauseIntent</code>, <code>ShuffleOnIntent</code>, <code>LoopOffIntent</code> etc.)</p> <p>2. Different Intents can invoke different AWS Lambda functions</p>

References

- [1] D. Taranov, “Facebook comments growth tools 2.0 : Manychat.” <https://support.manychat.com/support/solutions/articles/36000232601-facebook-comments-trigger-2-0>, July 2020. (Accessed on 08/07/2020).
- [2] D. L. AG, “The lufthansa chatbot.” <https://www.lufthansa.com/us/en/chatbot>. (Accessed on 08/07/2020).
- [3] Business Insider, “80% of businesses want chatbots by 2020,” 2016.
- [4] C. Rajnerowicz, “The Best Innovative Chatbot Examples by Industry.” <https://www.tidio.com/blog/chatbot-examples>. (Accessed on 01/07/2020).
- [5] J. Jose, “Create your ELIZA Chatbot in 20 minutes with Regular Expressions (Day 6).” <http://botartisanz.com/blog/create-your-eliza-chatbot-in-20-minutes-with-regular-expressions-day-6>. (Accessed on 01/07/2020).
- [6] S. Cornaby, “Let’s Program A Chatbot 6: Don’t Fear The Regex.” <https://scottcornaby.com/2013/10/26/lets-program-a-chatbot-6-dont-fear-the-regex/>. (Accessed on 01/07/2020).
- [7] N. Joshi, “Choosing Between Rule-Based Bots And AI Bots.” <https://www.forbes.com/sites/cognitiveworld/2020/02/23/choosing-between-rule-based-bots-and-ai-bots/#3e83037f353d>. (Accessed on 01/07/2020).

- [8] U. Shahid, “A Beginner’s Guide to Chatbots.” <https://blog.datasciencedojo.com/introduction-to-chatbots/>. (Accessed on 01/07/2020).
- [9] J. Alburger, “Rule-Based Chatbots vs. AI Chatbots: Key Differences.” <https://www.hubtype.com/blog/rule-based-chatbots-vs-ai-chatbots/>. (Accessed on 01/07/2020).
- [10] J. Weizenbaum, “ELIZA—a computer program for the study of natural language communication between man and machine,” *Communications of the ACM*, vol. 9, no. 1, pp. 36–45, 1966.
- [11] K. M. Colby, “Ten criticisms of parry,” *ACM SIGART Bulletin*, no. 48, pp. 5–9, 1974.
- [12] A. Turing, “Mind,” *Mind*, vol. 59, no. 236, pp. 433–460, 1950.
- [13] R. Carpenter, “Jabberwacky-live chat bot.” <http://www.jabberwacky.com/>. (Accessed on 01/07/2020).
- [14] R. Carpenter, “Cleverbot.com—a clever bot.” <https://www.cleverbot.com/>. (Accessed on 01/07/2020).
- [15] W. contributors, “Dr. Sbaitso - Wikipedia.” <https://en.wikipedia.org/wiki/Dr..Sbaitso>. (Accessed on 01/07/2020).
- [16] M. d. G. B. Marietto, R. V. de Aguiar, G. d. O. Barbosa, W. T. Botelho, E. Pimentel, R. d. S. França, and V. L. da Silva, “Artificial intelligence markup language: a brief tutorial,” *arXiv preprint arXiv:1307.3091*, 2013.
- [17] S. Worswick, “Mitsuku.” <https://www.pandorabots.com/mitsuku/>. (Accessed on 01/07/2020).
- [18] A. Solutions, “Elbot the Chatbot.” <https://www.elbot.com/>. (Accessed on 01/07/2020).
- [19] W. contributors, “SmarterChild - Wikipedia.” <https://en.wikipedia.org/wiki/SmarterChild>. (Accessed on 01/07/2020).

- [20] R. High, “The era of cognitive systems: An inside look at IBM Watson and how it works,” *IBM Corporation, Redbooks*, pp. 1–16, 2012.
- [21] IBM Corporation, “Watson Assistant.” <https://www.ibm.com/cloud/watson-assistant/>. (Accessed on 01/07/2020).
- [22] AI Multiple, “Top 60 Chatbot Companies of 2020: In-depth Guide.” <https://research.aimultiple.com/chatbot-companies/>, July 2020. (Accessed on 10/07/2020).
- [23] Apple Inc., “Siri - Apple.” <https://www.apple.com/siri/>. (Accessed on 01/07/2020).
- [24] Google LLC, “Google Assistant — Your own personal Google.” <https://assistant.google.com/>. (Accessed on 01/07/2020).
- [25] Microsoft Corporation, “Cortana - Your personal productivity assistant.” <https://www.microsoft.com/en-us/cortana>. (Accessed on 01/07/2020).
- [26] T. B. Elliot, *Alexa User Guide For Beginners: Tips and Tricks for Fully Optimizing Your Amazon Devices (Amazon Echo plus, Amazon Echo Dot, Fire TV, Fire Tablet, Etc.)*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018.
- [27] Google LLC, “Dialogflow.” <https://dialogflow.com/>. (Accessed on 01/07/2020).
- [28] Amazon.com Inc., “Amazon Lex – Build Conversation Bots.” <https://aws.amazon.com/lex/>. (Accessed on 01/07/2020).
- [29] P. B. Brandtzaeg and A. Følstad, “Chatbots: changing user needs and motivations,” *interactions*, vol. 25, no. 5, pp. 38–43, 2018.
- [30] A. P. Chaves and M. A. Gerosa, “How should my chatbot interact? A survey on human-chatbot interaction design,” *arXiv preprint arXiv:1904.02743*, 2019.
- [31] L. Waldera, “Development of a Preliminary Measurement Tool of User Satisfaction for Information-Retrieval Chatbots,” B.S. thesis, University of Twente, 2019.

- [32] X. Wang, S. S. Sohn, and M. Kapadia, “Towards a Conversational Interface for Authoring Intelligent Virtual Characters,” in *Proceedings of the 19th ACM International Conference on Intelligent Virtual Agents*, pp. 127–129, ACM, 2019.
- [33] E. Fast, B. Chen, J. Mendelsohn, J. Bassen, and M. S. Bernstein, “Iris: A conversational agent for complex tasks,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, p. 473, ACM, 2018.
- [34] R. Håvik, J. D. Wake, E. Flobak, A. Lundervold, and F. Guribye, “A Conversational Interface for Self-screening for ADHD in Adults,” in *International Conference on Internet Science*, pp. 133–144, Springer, 2018.
- [35] J. Allen, G. Ferguson, and A. Stent, “An architecture for more realistic conversational systems,” in *Proceedings of the 6th international conference on Intelligent user interfaces*, pp. 1–8, 2001.
- [36] T. Reenskaug, “Mvc xerox parc 1978-79,” *Trygve/MVC*, 1979.
- [37] G. Pilato, A. Augello, and S. Gaglio, “A modular architecture for adaptive ChatBots,” in *2011 IEEE Fifth International Conference on Semantic Computing*, pp. 177–180, IEEE, 2011.
- [38] J. Cahn, “CHATBOT: Architecture, Design, & Development,” 2017. Senior Thesis, University of Pennsylvania.
- [39] X. Franch, L. López, C. Cares, and D. Colomer, “The i* framework for goal-oriented modeling,” in *Domain-specific conceptual modeling*, pp. 485–506, Springer, 2016.
- [40] Z. Babar, A. Lapouchnian, and E. Yu, “Chatbot Design-Reasoning about design options using i* and process architecture,” in *CEUR Workshop Proceedings*, pp. 73–78, 2017.
- [41] T. E. B. Manager, “A breakdown of chatbot architecture and how it works - enterprise bot manager.” <https://www.enterprisebotmanager.com/chatbot-architecture-and-how-they-work/>. (Accessed on 07/31/2020).

- [42] N. Saxena, “Chatbot architecture tutorial.” <https://towardsdatascience.com/chatbot-tutorial-choosing-the-right-chatbot-architecture-5539c8489def>, January 2020. (Accessed on 07/31/2020).
- [43] M. Labs, “How do chatbots work? a guide to the chatbot architecture.” <https://marutitech.com/chatbots-work-guide-chatbot-architecture/>. (Accessed on 07/31/2020).
- [44] A. Smith, “Understanding architecture models of chatbot and response generation mechanisms - dzone ai.” <https://dzone.com/articles/understanding-architecture-models-of-chatbot-and-r>, March 2020. (Accessed on 07/31/2020).
- [45] AltexSoft, “A technical guide to building an ai chatbot.” <https://www.altexsoft.com/blog/datascience/a-technological-guide-to-building-an-ai-chatbot/>, February 2019. (Accessed on 07/31/2020).
- [46] B. R. Singh, “Chat bots — designing intents and entities for your nlp models.” <https://medium.com/@brijrajsingh/chat-bots-designing-intents-and-entities-for-your-nlp-models-35c385b7730d>, January 2017. (Accessed on 07/31/2020).
- [47] S. Karri, “What goes into making a successful nlp design for chatbots.” <https://blog.kore.ai/what-goes-into-making-a-successful-nlp-design-for-chatbots>. (Accessed on 07/31/2020).
- [48] elprocus.com, “Chatbot : Architecture, applications and design process steps.” <https://www.elprocus.com/chatbot-design-process-and-its-architecture/>. (Accessed on 07/31/2020).
- [49] J. Gao, M. Galley, and L. Li, *Neural Approaches to Conversational AI: Question Answering, Task-oriented Dialogues and Social Chatbots*. Now Foundations and Trends, 2019.

- [50] J. Guo, Y. Fan, L. Pang, L. Yang, Q. Ai, H. Zamani, C. Wu, W. B. Croft, and X. Cheng, “A deep look into neural ranking models for information retrieval,” *Information Processing & Management*, p. 102067, 2019.
- [51] P. Suta, X. Lan, B. Wu, P. Mongkolnam, and J. H. Chan, “An Overview of Machine Learning in Chatbots,” *International Journal of Mechanical Engineering and Robotics Research*, vol. 9, no. 4, 2020.
- [52] H. Chen, X. Liu, D. Yin, and J. Tang, “A survey on dialogue systems: Recent advances and new frontiers,” *Acm Sigkdd Explorations Newsletter*, vol. 19, no. 2, pp. 25–35, 2017.
- [53] D. Newman, “How chatbots and deep learning will change the future of organizations.” <https://www.forbes.com/sites/danielnewman/2016/06/28/how-chatbots-and-deep-learning-will-change-the-future-of-organizations/#4a205fb94734>, June 2016. (Accessed on 08/01/2020).
- [54] S. C. Hoi, J. Wang, and P. Zhao, “Libol: A library for online learning algorithms,” *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 495–499, 2014.
- [55] A. B. Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. García, S. Gil-López, D. Molina, R. Benjamins, *et al.*, “Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI,” *Information Fusion*, vol. 58, pp. 82–115, 2020.
- [56] A. Holzinger, P. Kieseberg, E. Weippl, and A. M. Tjoa, “Current advances, trends and challenges of machine learning and knowledge extraction: from machine learning to explainable AI,” in *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*, pp. 1–8, Springer, 2018.
- [57] L. Deng and X. Huang, “Challenges in adopting speech recognition,” *Communications of the ACM*, vol. 47, no. 1, pp. 69–75, 2004.
- [58] V. Radha and C. Vimala, “A review on speech recognition challenges and approaches,” *doaj. org*, vol. 2, no. 1, pp. 1–7, 2012.

- [59] P. Taylor, *Text-to-Speech Synthesis*. Cambridge University Press, 2009.
- [60] A. Rosenberg, “Speech, prosody, and machines: Nine challenges for prosody research,” in *Proc. Speech Prosody*, pp. 784–793, 2018.
- [61] R. S. Lavin, “Issues in Chinese prosody: conceptual foundations of a linguistically-motivated text-to-speech system for Mandarin,” in *Proceedings of the 16th Pacific Asia Conference on Language, Information and Computation*, pp. 259–270, 2001.
- [62] Y. Kim, “Convolutional neural networks for sentence classification,” *arXiv preprint arXiv:1408.5882*, 2014.
- [63] K. Yao, B. Peng, Y. Zhang, D. Yu, G. Zweig, and Y. Shi, “Spoken language understanding using long short-term memory neural networks,” in *2014 IEEE Spoken Language Technology Workshop (SLT)*, pp. 189–194, IEEE, 2014.
- [64] Z. Zhao and Y. Wu, “Attention-Based Convolutional Neural Networks for Sentence Classification.,” in *INTERSPEECH*, pp. 705–709, 2016.
- [65] C. Marshall, “What is named entity recognition (ner) and how can i use it?.” <https://medium.com/mysuperaai/what-is-named-entity-recognition-ner-and-how-can-i-use-it-2b68cf6f545d>, December 2019. (Accessed on 08/05/2020).
- [66] G. Mesnil, Y. Dauphin, K. Yao, Y. Bengio, L. Deng, D. Hakkani-Tur, X. He, L. Heck, G. Tur, D. Yu, *et al.*, “Using recurrent neural networks for slot filling in spoken language understanding,” *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 23, no. 3, pp. 530–539, 2014.
- [67] B. Liu and I. Lane, “Recurrent Neural Network Structured Output Prediction for Spoken Language Understanding,” in *Proc. NIPS Workshop on Machine Learning for Spoken Language Understanding and Interactions*, 2015.
- [68] N. T. Vu, “Sequential convolutional neural networks for slot filling in spoken language understanding,” *arXiv preprint arXiv:1606.07783*, 2016.

- [69] P. Xu and R. Sarikaya, “Convolutional neural network based triangular crf for joint intent detection and slot filling,” in *2013 ieee workshop on automatic speech recognition and understanding*, pp. 78–83, IEEE, 2013.
- [70] B. Liu and I. Lane, “Attention-based recurrent neural network models for joint intent detection and slot filling,” *arXiv preprint arXiv:1609.01454*, 2016.
- [71] Q. Chen, Z. Zhuo, and W. Wang, “Bert for joint intent classification and slot filling,” *arXiv preprint arXiv:1902.10909*, 2019.
- [72] I. Facebook, “Messenger.” <https://www.messenger.com/>. (Accessed on 08/07/2020).
- [73] S. Technologies, “Where work happens — slack.” <https://slack.com/intl/en-in/>. (Accessed on 08/07/2020).
- [74] T. F. LLC, “Telegram messenger.” <https://telegram.org/>. (Accessed on 08/07/2020).
- [75] freshworks.com, “Chat widget — freshchat by freshworks.” <https://www.freshworks.com/live-chat-software/chat-widget/>. (Accessed on 08/08/2020).
- [76] R. Fielding, *Representational state transfer (REST). Chapter 5 in Architectural Styles and the Design of Networkbased Software Architectures*. PhD thesis, University of California, Irvine, CA, 2000.
- [77] G. Muller, “Right Sizing Reference Architectures; How to provide specific guidance with limited information,” in *INCOSE International Symposium*, vol. 18, pp. 2047–2054, Wiley Online Library, 2008.
- [78] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, second ed., 2003.
- [79] I. Corporation, “Watson speech to text - overview — ibm cloud.” <https://www.ibm.com/in-en/cloud/watson-speech-to-text>. (Accessed on 08/05/2020).

- [80] I. Corporation, “Watson text to speech - overview — ibm cloud.” <https://www.ibm.com/in-en/cloud/watson-text-to-speech>. (Accessed on 08/05/2020).
- [81] I. Corporation, “Ibm cloud functions.” <https://cloud.ibm.com/functions/>. (Accessed on 08/08/2020).
- [82] A. Inc., “Aws lambda – serverless compute - amazon web services.” <https://aws.amazon.com/lambda/>. (Accessed on 08/08/2020).
- [83] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, “AUTOSAR—A Worldwide Standard is on the Road,” in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, vol. 62, p. 5, 2009.
- [84] F. Wartena, J. Muskens, L. Schmitt, and M. Petkovic, “Continua: The reference architecture of a personal telehealth ecosystem,” in *The 12th IEEE International Conference on e-Health Networking, Applications and Services*, pp. 1–6, IEEE, 2010.
- [85] J. Vesterager, M. Tolle, and P. Bernus, “VERA: virtual enterprise reference architecture,” in *VTT SYMPOSIUM*, vol. 224, pp. 39–52, VTT; 1999, 2003.
- [86] R. B. Bohn, J. Messina, F. Liu, J. Tong, and J. Mao, “NIST cloud computing reference architecture,” in *2011 IEEE World Congress on Services*, pp. 594–596, IEEE, 2011.
- [87] S. Martínez-Fernández, C. P. Ayala, X. Franch, and H. M. Marques, “Benefits and drawbacks of reference architectures,” in *European Conference on Software Architecture*, pp. 307–310, Springer, 2013.
- [88] E. Y. Nakagawa, P. Oliveira Antonino, and M. Becker, “Reference architecture and product line architecture: A subtle but critical difference,” in *Software Architecture* (I. Crnkovic, V. Gruhn, and M. Book, eds.), (Berlin, Heidelberg), pp. 207–211, Springer Berlin Heidelberg, 2011.

- [89] R. Kompella, “Conversational ai chat-bot — architecture overview.” <https://towardsdatascience.com/architecture-overview-of-a-conversational-ai-chat-bot-4ef3dfefd52e>. (Accessed on 08/15/2020).
- [90] J. Hansen, “Modern chatbot reference architecture.” <https://jannehansen.com/chatbot-architecture/>. (Accessed on 08/15/2020).
- [91] S. Shinde, “Intelligent chatbot framework & reference architecture.” <https://medium.com/@suraj.shinde/intelligent-chatbot-framework-reference-architecture-3e7222907bb4>. (Accessed on 08/15/2020).
- [92] minigbusinessdata.com, “Dialogflow vs Lex vs Luis vs Watson vs Chatfuel.” <https://miningbusinessdata.com/dialogflow-vs-lex-vs-luis-vs-watson-vs-chatfuel/>, July 2020. (Accessed on 08/08/2020).
- [93] A. Brooks, “10 Best Chatbot Builders in 2019.” <https://www.ventureharbour.com/best-chatbot-builders/>, June 2020. (Accessed on 08/08/2020).
- [94] G. LLC, “Speech-to-text: Automatic speech recognition — google cloud.” <https://cloud.google.com/speech-to-text>. (Accessed on 08/05/2020).
- [95] Sonix, “Automatically convert audio to text—it’s fast, simple, & affordable — sonix.” <https://sonix.ai/>. (Accessed on 08/05/2020).
- [96] A. Inc., “Amazon transcribe – speech to text - aws.” <https://aws.amazon.com/transcribe/>. (Accessed on 08/05/2020).
- [97] G. Transcribe, “Go transcribe: Fast, simple & affordable ai based transcription.” <https://go-transcribe.com/>. (Accessed on 08/05/2020).
- [98] C. M. University, “Cmusphinx open source speech recognition.” <https://cmusphinx.github.io/>. (Accessed on 08/05/2020).
- [99] D. Povey *et al.*, “Kaldi: About the kaldi project.” <https://kaldi-asr.org/doc/about.html>. (Accessed on 08/05/2020).

- [100] M. Foundation, “Home · mozilla/deepspeech wiki · github.” <https://github.com/mozilla/DeepSpeech/wiki>. (Accessed on 08/05/2020).
- [101] G. LLC, “Text-to-speech: Lifelike speech synthesis — google cloud.” <https://cloud.google.com/text-to-speech>. (Accessed on 08/05/2020).
- [102] M. Corporation, “Text to speech — microsoft azure.” <https://azure.microsoft.com/en-us/services/cognitive-services/text-to-speech/>. (Accessed on 08/05/2020).
- [103] A. Inc., “Amazon polly.” <https://aws.amazon.com/polly/>. (Accessed on 08/05/2020).
- [104] M. Foundation, “Github - mozilla/tts: Deep learning for text to speech (discussion forum: <https://discourse.mozilla.org/c/tts>).” <https://github.com/mozilla/TTS>. (Accessed on 08/05/2020).
- [105] C. M. University, “Cmu flite: Speech synthesizer.” <http://www.festvox.org/flite/>. (Accessed on 08/05/2020).
- [106] R. Dunn, “Github - espeak-ng/espeak-ng: espeak ng is an open source speech synthesizer that supports more than hundred languages and accents..” <https://github.com/espeak-ng/espeak-ng>. (Accessed on 08/05/2020).
- [107] A. Jain, “A brief introduction to intent classification — by akshat jain — towards data science.” <https://towardsdatascience.com/a-brief-introduction-to-intent-classification-96fda6b1f557>, November 2018. (Accessed on 08/05/2020).
- [108] T. Wochinger, “Rasa nlu in depth: Intent classification.” <https://blog.rasa.com/rasa-nlu-in-depth-part-1-intent-classification/>, February 2019. (Accessed on 08/05/2020).
- [109] W. contributors, “Template processor - wikipedia.” https://en.wikipedia.org/wiki/Template_processor. (Accessed on 08/05/2020).

- [110] awesomeopensource.com, “The top 118 template engine open source projects.” <https://awesomeopensource.com/projects/template-engine>. (Accessed on 08/05/2020).
- [111] D. Stenberg, “curl.” <https://curl.haxx.se/>. (Accessed on 08/05/2020).
- [112] hellotars.com, “Increase conversion rates with conversational landing pages for google ads - tars.” <https://hellotars.com/>. (Accessed on 08/06/2020).
- [113] M. Yang, “Messenger bot marketing made easy with manychat.” <https://manychat.com/>. (Accessed on 08/06/2020).
- [114] M. Lambert, “Chatbot decision trees. seriously, how hard can they be?.” <https://chatbotslife.com/chatbot-decision-trees-a42ed8b8cf32>, April 2018. (Accessed on 08/05/2020).
- [115] yellow.systems, “How to build a chatbot from scratch (cost and features) - yellow.” <https://yellow.systems/blog/how-to-build-a-chatbot-from-scratch>. (Accessed on 08/06/2020).
- [116] P. Pandey, “Building a simple chatbot from scratch in python (using nltk) — by parul pandey — analytics vidhya — medium.” <https://medium.com/analytics-vidhya/building-a-simple-chatbot-in-python-using-nltk-7c8c8215ac6e>, September 2018. (Accessed on 08/06/2020).
- [117] N. S. Chauhan, “Build your first chatbot using python nltk — by nagesh singh chauhan — towards data science.” <https://towardsdatascience.com/build-your-first-chatbot-using-python-nltk-5d07b027e727>, October 2018. (Accessed on 08/06/2020).
- [118] D. Singh, “Build a chatbot with python — pluralsight.” <https://www.pluralsight.com/guides/build-a-chatbot-with-python>, April 2020. (Accessed on 08/06/2020).
- [119] S. Bird, E. Loper, E. Klein, *et al.*, “Natural language toolkit — nltk 3.5 documentation.” <https://www.nltk.org/>. (Accessed on 08/06/2020).

- [120] N. Kohn, “Build an ai / machine learning chatbot in python with rasa — part 1.” <https://medium.com/hackernoon/build-simple-chatbot-with-rasa-part-1-f4c6d5bb1aea>, March 2018. (Accessed on 08/06/2020).
- [121] N. Kohn, “Build an ai / machine learning chatbot in python with rasa — part 2.” <https://medium.com/hackernoon/build-simple-chatbot-with-rasa-part-2-16726357b72c>, March 2018. (Accessed on 08/06/2020).
- [122] R. Jain, “1. build a conversational chatbot with rasa stack and python— rasa nlu.” <https://medium.com/@itsromiljain/build-a-conversational-chatbot-with-rasa-stack-and-python-rasa-nlu-b79dfbe59491>, March 2019. (Accessed on 08/06/2020).
- [123] R. Jain, “2. build a conversational chatbot with rasa stack and python — rasa core.” <https://medium.com/@itsromiljain/build-a-conversational-chatbot-with-rasa-stack-and-python-rasa-core-41b9c38c26b>, April 2019. (Accessed on 08/06/2020).
- [124] P. Voigt and A. v. d. Bussche, *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer Publishing Company, Incorporated, 1st ed., 2017.
- [125] S. Roca, J. Sancho, J. García, and Álvaro Alesanco, “Microservice chatbot architecture for chronic patient support,” *Journal of Biomedical Informatics*, vol. 102, p. 103305, 2020.
- [126] B. Croft and J. Lafferty, *Language modeling for information retrieval*, vol. 13. Springer Science & Business Media, 2003.
- [127] F. Song and W. B. Croft, “A general language model for information retrieval,” in *Proceedings of the eighth international conference on Information and knowledge management*, pp. 316–321, 1999.
- [128] T. Ray, “Openai’s gigantic gpt-3 hints at the limits of language models for ai — zdnet.” <https://www.zdnet.com/article/openais-gigantic-gpt-3-hints-at-the-limits-of-language-models-for-ai/>, June 2020. (Accessed on 08/10/2020).

- [129] O. Davydova, “25 chatbot platforms: A comparative table — by data monsters — chatbots journal.” <https://chatbotsjournal.com/25-chatbot-platforms-a-comparative-table-aeefc932eaff>, May 2017. (Accessed on 08/10/2020).
- [130] Ometrics, “2020 chatbot platform comparison reviews.” <https://www.ometrics.com/blog/chatbot-platform-comparison-reviews/>, 2020. (Accessed on 08/09/2020).
- [131] S. Advice, “Chatbot software prices.” <https://www.softwareadvice.com/live-chat/chatbot-comparison/p/all/>. (Accessed on 08/09/2020).
- [132] capterra.com, “Best conversational ai platform software 2020 — reviews of the most popular tools & systems.” <https://www.capterra.com/conversational-ai-platform-software/>, 2020. (Accessed on 08/10/2020).
- [133] AIMultiple, “Top 175 chatbot platforms of 2020: In-depth guide.” <https://aimultiple.com/chatbot-platform>, 2020. (Accessed on 08/10/2020).
- [134] P. RESEARCH, “Top 19 chatbot platforms in 2020 - reviews, features, pricing, comparisons.” <https://www.predictiveanalyticstoday.com/top-chatbot-platform/>, 2020. (Accessed on 08/10/2020).
- [135] M. Corporation, “Luis (language understanding) - cognitive services - microsoft.” <https://www.luis.ai/>. (Accessed on 08/06/2020).
- [136] A. Inc., “Amazon comprehend - natural language processing (nlp) and machine learning (ml).” <https://aws.amazon.com/comprehend/>. (Accessed on 08/06/2020).
- [137] G. LLC, “Cloud natural language — google cloud.” <https://cloud.google.com/natural-language>. (Accessed on 08/06/2020).
- [138] I. Corporation, “Watson natural language understanding - overview - india — ibm.” <https://www.ibm.com/in-en/cloud/watson-natural-language-understanding>. (Accessed on 08/06/2020).

- [139] S. Madan, “A Literature Analysis on Privacy Preservation Techniques,” in *Advances in Computing and Intelligent Systems*, pp. 223–230, Springer, 2020.
- [140] A. Ilavarasi, B. Sathiyabhama, and S. Poorani, “A survey on privacy preserving data mining techniques,” *International Journal of Computer Science and Business Informatics*, vol. 7, no. 1, 2013.
- [141] A. Weidauer, “Github -rasahq/rasa: Open source machine learning framework to automate text- and voice-based conversations.” <https://github.com/RasaHQ/rasa>. (Accessed on 08/06/2020).
- [142] M. Honnibal, “spacy - industrial-strength natural language processing in python.” <https://spacy.io/>. (Accessed on 08/06/2020).
- [143] R. T. Ltd., “gensim: Topic modelling for humans.” <https://radimrehurek.com/gensim/>. (Accessed on 08/06/2020).
- [144] G. LLC, “Tensorflow.” <https://www.tensorflow.org/>. (Accessed on 08/06/2020).
- [145] botscrew.com, “Botscrew bot framework - botscrew.” <https://botscrew.com/blog/botscrew-bot-framework/>. (Accessed on 08/06/2020).
- [146] A. S. Foundation, “Apache opennlp.” <https://opennlp.apache.org/>. (Accessed on 08/06/2020).
- [147] A. S. Foundation, “Apache uima.” <https://uima.apache.org/>. (Accessed on 08/06/2020).
- [148] S. University, “The stanford natural language processing group.” <https://nlp.stanford.edu/>. (Accessed on 08/06/2020).
- [149] H. B. Ezra and A. B. Ezra, “Snatchbot: Free chatbot solutions, intelligent bots service and artificial intelligence.” <https://snatchbot.me/>. (Accessed on 08/06/2020).
- [150] D. Dumik, “Chatfuel.” <https://chatfuel.com/>. (Accessed on 08/06/2020).

-
- [151] Chatfuel, “Build smarter chatbots with ai.” <https://blog.chatfuel.com/how-to-build-smarter-bots-with-ai/>, August 2019. (Accessed on 08/18/2020).
- [152] F. ROSENZWEIG, “What to do if users don’t click the bot’s buttons - manychat blog.” <https://manychat.com/blog/what-to-do-if-users-dont-click-the-bots-buttons/>, July 2019. (Accessed on 08/18/2020).
- [153] Edgar, “Ai setup — chatfuel documentation.” <https://docs.chatfuel.com/en/articles/828883-ai-setup>, August 2020. (Accessed on 08/18/2020).
- [154] W. contributors, “Goto - wikipedia.” <https://en.wikipedia.org/wiki/Goto>. (Accessed on 08/18/2020).
- [155] G. LLC, “Follow-up intents — dialogflow documentation — google cloud.” <https://cloud.google.com/dialogflow/docs/contexts-follow-up-intents>. (Accessed on 08/18/2020).
- [156] G. LLC, “Input and output contexts — dialogflow documentation — google cloud.” <https://cloud.google.com/dialogflow/docs/contexts-input-output>. (Accessed on 08/18/2020).
- [157] W. Chegham, “Mastering follow-up intents with dialogflow — medium.” <https://medium.com/google-developer-experts/mastering-follow-up-intents-with-dialogflow-851b75b83f5a>, November 2017. (Accessed on 08/18/2020).
- [158] G. LLC, “Inline editor — dialogflow documentation — google cloud.” <https://cloud.google.com/dialogflow/docs/fulfillment-inline-editor>. (Accessed on 08/18/2020).
- [159] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “RFC2616: Hypertext Transfer Protocol–HTTP/1.1,” 1999.
- [160] R. Brink, “Handling Multi-Intent Questions in Watson Assistant.” <https://medium.com/ibm-watson/handling-multi-intent-questions-in-watson-assistant-ccd0c6ea21e1>. Accessed: 2019-07-20.

-
- [161] A. Agrawal and T. Prabhakar, “Hospitality of cloud platforms,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 389–396, ACM, 2013.
- [162] L. Bass, F. Bachmann, and M. Klein, “Deriving architectural tactics—a step toward methodical architectural design,” *Software Engineering Institute, Carnegie Mellon University CMU/SEI-2003-TR-004*, 2003.
- [163] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, third ed., 2012.
- [164] M. Stal, F. Buschmann, and R. Meunier, “Pattern-oriented Software Architecture—A System of Patterns,” 1996.
- [165] F. Bachmann, L. Bass, and R. Nord, “Modifiability tactics,” *Software Engineering Institute, Carnegie Mellon University CMU/SEI-2007-TR-002*, 2007.
- [166] J. Scott and R. Kazman, “Realizing and refining architectural tactics: Availability,” tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2009.
- [167] P. Bourque, R. E. Fairley, *et al.*, *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.
- [168] C. T. Solutions, “An Analysis of the Best NLP Tool to Build a Conversational Bot.” <https://www.bitcoininsider.org/article/19026/analysis-best-nlp-tool-build-conversational-bot>, February 2018. (Accessed on 08/08/2020).
- [169] ubisend.com, “Blog - your chatbot and mobile messaging resource - ubisend.” <https://blog.ubisend.com/>. (Accessed on 08/13/2020).
- [170] chatbots.org, “Chatbots.org - virtual assistants, virtual agents, chat bots, conversational agents, chatterbots.” <https://www.chatbots.org/>. (Accessed on 08/13/2020).

- [171] J. Atwood and J. Spolsky, “Stack overflow - where developers learn, share, & build careers.” <https://stackoverflow.com/>, September 2008. (Accessed on 08/13/2020).
- [172] V. Belton and T. Stewart, *Multiple criteria decision analysis: an integrated approach*. Springer Science & Business Media, 2002.
- [173] C.-L. Hwang and K. Yoon, *Multiple attribute decision making: methods and applications a state-of-the-art survey*, vol. 186. Springer Science & Business Media, 2012.
- [174] R. Kazman, L. Bass, G. Abowd, and M. Webb, “SAAM: A method for analyzing the properties of software architectures,” in *Proceedings of 16th International Conference on Software Engineering*, pp. 81–90, IEEE, 1994.
- [175] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, “The architecture tradeoff analysis method,” in *Proceedings. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No. 98EX193)*, pp. 68–78, IEEE, 1998.
- [176] S. CBAM, “Cost Benefit Analysis Method,” tech. rep., Software Engineering Institute, Carnegie Mellon University, 2015.
- [177] R. Kazman, “Tool support for architecture analysis and design,” in *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints’ 96) on SIGSOFT’96 workshops*, pp. 94–97, 1996.
- [178] L. Bass, “ArchE-An Architecture Design Assistant,” tech. rep., Software Engineering Institute, Carnegie Mellon University, 2007.
- [179] S. Thiel, A. Hein, and H. Engelhardt, “Tool Support for Scenario-Based Architecture Evaluation,” in *STRAW*, pp. 41–45, 2003.
- [180] M. T. Ionita, D. K. Hammer, and H. Obbink, “Scenario-based software architecture evaluation methods: An overview,” in *Workshop on methods and techniques for software architecture review and assessment at the international conference on software engineering*, pp. 19–24, 2002.

-
- [181] M. A. Babar and I. Gorton, “Comparison of scenario-based software architecture evaluation methods,” in *11th Asia-Pacific Software Engineering Conference*, pp. 600–607, IEEE, 2004.
- [182] M. A. Babar, L. Zhu, and R. Jeffery, “A framework for classifying and comparing software architecture evaluation methods,” in *2004 Australian Software Engineering Conference. Proceedings.*, pp. 309–318, IEEE, 2004.
- [183] D. Braun, A. Hernandez-Mendez, F. Matthes, and M. Langen, “Evaluating natural language understanding services for conversational question answering systems,” in *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, (Saarbrücken, Germany), pp. 174–185, Association for Computational Linguistics, Aug. 2017.
- [184] X. Liu, A. Eshghi, P. Swietojanski, and V. Rieser, “Benchmarking natural language understanding services for building conversational agents,” *arXiv e-prints*, p. arXiv:1903.05566, Mar 2019.
- [185] P. Resnik and J. Lin, *Evaluation of NLP Systems*, ch. 11, pp. 271–295. John Wiley & Sons, Ltd, 2010.
- [186] R. BERGER, M. EBNER, and M. EBNER, “Conception of a Conversational Interface to Provide a Guided Search of Study Related Data,” in *International Journal of Emerging Technologies in Learning (iJET)*, pp. 37–47, April 2019.
- [187] M. Canonico and L. D. Russis, “A comparison and critique of natural language understanding tools,” in *In Cloud Computing 2018*. 110–115, 06 2018.
- [188] D. Peras, “Chatbot evaluation metrics,” *Economic and Social Development: Book of Proceedings*, pp. 89–97, 2018.
- [189] D. Braun and F. Matthes, “Towards a Framework for Classifying Chatbots,” in *Proceedings of the 21th International Conference on Enterprise Information Systems - Volume 1: ICEIS*, 2019.
- [190] G. Daniel, J. Cabot, L. Deruelle, and M. Derras, “Multi-platform chatbot modeling and deployment with the jarvis framework,” in *International Con-*

- ference on Advanced Information Systems Engineering*, pp. 177–193, Springer, 2019.
- [191] botpress.com, “Botpress — chatbot platform comparison.” <https://botpress.com/blog/chatbot-platform-comparison>. (Accessed on 08/15/2020).
- [192] discover.bot, “A bot development and information portal - discover.bot.” <https://discover.bot/>. (Accessed on 08/19/2020).
- [193] A. Multiple, “Chatbot — aimultiple.” <https://research.aimultiple.com/category/e2e-chatbot/>, September 2020. (Accessed on 08/19/2020).
- [194] T. Wellhausen and A. Fiesser, “How to write a pattern? A rough guide for first-time pattern authors,” in *Proceedings of the 16th European Conference on Pattern Languages of Programs*, pp. 1–9, 2011.
- [195] visual.paradigm.com, “User story vs use case for agile software development.” <https://www.visual-paradigm.com/guide/agile-software-development/user-story-vs-use-case/>. (Accessed on 08/18/2020).
- [196] D. Faggella, “7 chatbot use cases that actually work — emerj.” <https://emerj.com/ai-sector-overviews/7-chatbot-use-cases-that-actually-work/>, December 2019. (Accessed on 08/18/2020).
- [197] O. web technology, “Building a chatbot leveraging artificial intelligence technologies — open web technology - go digital.” <https://openwt.com/en/cases/building-chatbot-leveraging-artificial-intelligence-technologies>. (Accessed on 08/18/2020).
- [198] thebotforge.io, “Forging a successful chatbot project — the bot forge.” <https://www.thebotforge.io/forging-a-successful-chatbot-project/>. (Accessed on 08/18/2020).
- [199] A. Multiple, “Why chatbots fail in 2020 (and why natural languages are hard).” <https://research.aimultiple.com/why-chatbots-fail/>, July 2020. (Accessed on 08/19/2020).

- [200] discover.bot, “Pitfalls of natural language processing chatbots - discover.bot.” <https://discover.bot/bot-talk/natural-language-processing-common-challenge/>, October 2018. (Accessed on 08/19/2020).
- [201] pubnub.com, “What is a serverless function? — pubnub.” <https://www.pubnub.com/blog/what-is-a-serverless-function/>, May 2019. (Accessed on 08/19/2020).
- [202] SMARTBEAR, “What is an api endpoint? — smartbear software resources.” <https://smartbear.com/learn/performance-monitoring/api-endpoints/>, 2018. (Accessed on 08/19/2020).
- [203] M. Cohn, “User stories and user story examples by mike cohn.” <https://www.mountangoatsoftware.com/agile/user-stories>. (Accessed on 08/20/2020).
- [204] I. ISO, “8601: 2004,” *Data elements and interchange formats—Information interchange—Representation of dates and times*, vol. 3, 2004.
- [205] S. Srivastava, “ChanakyaAirlinesConfiguration Repository — Bitbucket.” <https://bitbucket.org/ssri5/chanakyaairlinesconfiguration/>, May 2020. (Accessed on 08/21/2020).
- [206] S. Srivastava, “AirlineOperationsBackend Repository — Bitbucket.” <https://bitbucket.org/ssri5/airlineoperationsbackend/>, May 2020. (Accessed on 08/21/2020).
- [207] Open Source Initiative and others, “The MIT license.” <https://opensource.org/licenses/MIT>, 2006. (Accessed on 08/21/2020).
- [208] E. F. Codd, “A relational model of data for large shared data banks,” in *Software pioneers*, pp. 263–294, Springer, 2002.
- [209] D. D. Chamberlin and R. F. Boyce, “SEQUEL: A structured English query language,” in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pp. 249–264, 1974.

- [210] ESPNcricinfo.com, “Statsguru — searchable cricket statistics database.” <http://stats.espncricinfo.com/ci/engine/stats/index.html>, 1993. (Accessed on 12/09/2019).
- [211] S. Srivastava, “Cricketnoviceexperiments repository — bitbucket.” <https://bitbucket.org/ssri5/cricketnoviceexperiments/>, August 2020. (Accessed on 08/27/2020).
- [212] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [213] I. Tenney, D. Das, and E. Pavlick, “Bert rediscovers the classical nlp pipeline,” *arXiv preprint arXiv:1905.05950*, 2019.
- [214] S. I. Nikolenko, S. Koltcov, and O. Koltsova, “Topic modelling for qualitative studies,” *Journal of Information Science*, vol. 43, no. 1, pp. 88–102, 2017.
- [215] J. C. K. Cheung and X. Li, “Sequence clustering and labeling for unsupervised query intent discovery,” in *Proceedings of the fifth ACM international conference on Web search and data mining*, pp. 383–392, 2012.
- [216] Y. Li, B.-J. P. Hsu, and C. Zhai, “Unsupervised identification of synonymous query intent templates for attribute intents,” in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pp. 2029–2038, 2013.
- [217] G. Forman, H. Nachlieli, and R. Keshet, “Clustering by intent: a semi-supervised method to discover relevant clusters incrementally,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 20–36, Springer, 2015.
- [218] F. Reinaldha and T. E. Widagdo, “Natural language interfaces to database (NLIDB): Question handling and unit conversion,” in *2014 International Conference on Data and Software Engineering (ICODSE)*, pp. 1–6, IEEE, 2014.

- [219] N. Sukthankar, S. Maharnawar, P. Deshmukh, Y. Haribhakta, and V. Kamble, “nQuery-A Natural Language Statement to SQL Query Generator,” in *Proceedings of ACL 2017, Student Research Workshop*, pp. 17–23, 2017.
- [220] K. Höffner, J. Lehmann, and R. Usbeck, “CubeQA—question answering on RDF data cubes,” in *International Semantic Web Conference*, pp. 325–340, Springer, 2016.
- [221] M. Atzori, G. M. Mazzeo, and C. Zaniolo, “QA3: A natural language approach to question answering over RDF data cubes,” *Semantic Web*, vol. 10, no. 3, pp. 587–604, 2019.
- [222] kore.ai, “Auto dialog generation — kore.ai.” <https://kore.ai/platform/features/auto-dialog-generation/>, 2020. (Accessed on 08/28/2020).
- [223] lang.ai, “Unsupervised intent discovery - technology — lang.ai.” <https://lang.ai/tech>. (Accessed on 08/28/2020).
- [224] D. Lee, “How we analyzed 200k messages to design a chatbot — chatbots magazine.” <https://chatbotsmagazine.com/how-we-analyzed-200k-messages-to-design-a-chatbot-264a13724752>, January 2019. (Accessed on 08/28/2020).
- [225] C. Greyling, “Handle compound user intents in your chatbot — medium.” <https://medium.com/@CobusGreyling/handle-compound-user-intents-in-your-chatbot-cac598c5fea5>, May 2020. (Accessed on 08/28/2020).
- [226] C. Greyling, “How to resolve intent conflicts within your chatbot — medium.” <https://medium.com/@CobusGreyling/how-to-resolve-intent-conflicts-within-your-chatbot-74f567d30d48>, July 2020. (Accessed on 08/28/2020).
- [227] A. C. Gonzalez, “Sorry i didn’t get that! — how to understand what your users want.” <https://building.lang.ai/sorry-i-didnt-get-that-how-to-understand-what-your-users-want-a90c7ca18a8f>, October 2018. (Accessed on 08/28/2020).

- [228] M. Du and S. Yao, “Discovering and classifying in-app message intent at airbnb — medium.” <https://medium.com/airbnb-engineering/discovering-and-classifying-in-app-message-intent-at-airbnb-6a55f5400a0c>, January 2019. (Accessed on 08/28/2020).
- [229] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.
- [230] H. Jelodar, Y. Wang, C. Yuan, X. Feng, X. Jiang, Y. Li, and L. Zhao, “Latent Dirichlet Allocation (LDA) and Topic modeling: models, applications, a survey,” *Multimedia Tools and Applications*, vol. 78, no. 11, pp. 15169–15211, 2019.
- [231] G. Sandberg, “A primer on relational data base concepts,” *IBM systems journal*, vol. 20, no. 1, pp. 23–40, 1981.
- [232] M. Dadashzadeh and D. W. Stemple, “Converting SQL queries into relational algebra,” *Information & management*, vol. 19, no. 5, pp. 307–323, 1990.
- [233] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, “Natural language interfaces to databases-an introduction,” *arXiv preprint cmp-lg/9503016*, 1995.
- [234] I. Corporation, “Watson Assistant v2 - IBM Cloud API Docs.” <https://cloud.ibm.com/apidocs/assistant/assistant-v2>. (Accessed on 15/08/15/2020, See the first “Tip”).
- [235] I. Corporation, “Watson Assistant API overview.” <https://cloud.ibm.com/docs/assistant?topic=assistant-api-overview>. (Accessed on 20/07/2019, See the second “Note”).
- [236] A. Inc., “lex-models - AWS CLI 1.18.101 Command Reference.” <https://docs.aws.amazon.com/cli/latest/reference/lex-models/index.html>. (Accessed on 20/07/2019).
- [237] G. LLC, “Input and output contexts — Dialogflow Documentation — Google Cloud.” <https://cloud.google.com/dialogflow/docs/contexts-input-output>. (Accessed on 20/07/2019).

Publications

1. *Saurabh Srivastava* and *T.V. Prabhakar*. A Reference Architecture for Applications with Conversational Components. In IEEE 10th International Conference on Software Engineering and Service Science (**ICSESS**), Beijing, China. 2019.
2. *Saurabh Srivastava* and *T.V. Prabhakar*. Desirable Features of a Chatbot-building Platform. In 2nd IEEE International Conference on Humanized Computing and Communication (**HCCAI**), Irvine, USA. 2020. (accepted)
3. *Saurabh Srivastava* and *T.V. Prabhakar*. Hospitality of Chatbot building Platforms. In 2nd ACM SIGSOFT International Workshop on Software Qualities and Their Dependencies (**SQUADE**), Tallinn, Estonia. 2019.
4. *Saurabh Srivastava*, *Sumit Kalra*, and *T.V. Prabhakar*. Contextual Reactive Pattern on Chatbot-building Platforms. In 25th European Conference on Pattern Languages of Programs (**EuroPLoP**), Kloster Irsee, Bavaria, Germany. ACM ICPS, 2020. (accepted)
5. *Saurabh Srivastava* and *T.V. Prabhakar*. Intent Sets - Architectural Choices for Building Practical Chatbot. In 12th International Conference on Computer and Automation Engineering (**ICCAE**), Sydney, Australia. ACM ICPS, 2020.