A Reference Architecture for Applications with Conversational Components

Saurabh Srivastava  
Department of Computer Science & Engineering  
Indian Institute of Technology  
Kanpur, India  
ssri@iitk.ac.in

T.V. Prabhakar  
Department of Computer Science & Engineering  
Indian Institute of Technology  
Kanpur, India  
tvp@iitk.ac.in

Abstract—Providing a multi-modal user interface adds value to any application. Allowing users to speak or chat with the system is one such area where software practitioners are putting a lot of effort. This involves building components which can understand the nuances of human conversation. Such components, often called “chatbots”, can be built either from scratch, or using a commercial platform. The process of architecting such applications may differ significantly from the “conventional” applications that the software practitioners usually build. In this work, we present a Reference Architecture for building such applications. We apply the Reference Architecture to a sample use-case and provide two Concrete Architectures for the same. The two architectures are designed keeping in mind, two commercial platforms, IBM Watson Assistant and Google DialogFlow, assuming that they were used to build the conversational components.

Index Terms—Reference architecture, Software architecture

I. INTRODUCTION

Ability to interact with a user in a natural language can enhance an application’s reach significantly. However, understanding natural languages is a complicated task. It may involve detecting sarcasm, figuring out colloquial terms, understanding hidden meaning or context etc. Nevertheless, recent breakthroughs in Natural Language Understanding (NLU) has led to more chatbots being built and deployed [1]. A chatbot is a software component, that can take inputs in the form of natural language phrases, and can return responses in the same way. An application providing a conversational interface usually deploys such components to mediate between the user and the rest of the system. In the current work, we attempt to provide a Reference Architecture for this class of applications. A Reference Architecture provides a blueprint to build concrete architectures for a specific domain or class of applications. For example, AUTOSAR [2] and Continua [3] offer reference architectures for building software components in the automotive and personal healthcare domains, respectively. Reference Architectures also provide a shared vision and terminology for different stakeholders [4].

This class of applications is different because of the Artificial Intelligence (AI) techniques which are put to use while building the conversational components. Many researchers opine that building a system with the presence of AI is significantly different from a system built without the same [5] [6] [7] [8].

Some issues associated with these systems, especially the applications with conversational components are:

- **High dependence on data**: Gleaning the intent of the user (e.g. “ordering a pizza”, “seeking weather update”, “casual greeting” etc.) from a given phrase usually requires a model built with Machine Learning tools. Additionally, converting a speech fragment to corresponding text also needs language-specific models (often customised for specific accents). Models are heavily dependent on the data used to train them. Considering the natural variations in the real world for these input domains (e.g. the different ways the same statement can be paraphrased, or the differences in pronunciation of the same word by different people), it is practically infeasible to build models which can cover all possible scenarios. This leaves clout of uncertainty when the model is deployed for use. This uncertainty affects any behaviour of the application, directed by an assumption that the failure rate of the models is satisfactorily low.

- **Expectation of worldly knowledge**: In conventional software systems, the expectations from software components are well-defined. For instance, a component that books a cab for a user expects a set of inputs (such as time, day, locations etc.), follows a set procedure (e.g. checking fair catalogues, calculating minimum distance routes etc.) and outputs a booking id. The inputs may be supplied by another software component, or by a human through a constrained interface (e.g. a web form with validations). However, when the inputs are taken through an unconstrained interface, e.g. typed or spoken text, there may be hidden expectations from the system to contain “worldly knowledge” or “common sense”. For example, when asked about the time of booking, the user may reply with a phrase like “2 in the morning”. There are hidden expectations from the system to understand that the term “morning” doesn’t have the literal meaning in this context, but points to early hours of the day. A system with a conversational interface may be overburdened to impart this common sense somehow in its flow.

- **Hard to debug or change**: In a conventional software system, the process of finding and correcting bugs is
well-understood. It involves shortlisting failing test cases, performing bug localisation, manipulating code, and repeating the tests. However, in general, a model cannot be “modified” in the same way as a code fragment. Also, “localising” the reasons for a problem might not be easy (may even be impossible in many cases). If a conversational model regularly fails over a type of phrases, the most practical solution would involve adding more training examples of that type, re-training or updating the model, and “hoping” that the newer model will be better. There is, however, a possibility that the new model may falter on a different class of phrases, initiating another round of similar efforts.

In this work, we attempt to provide a Reference Architecture for applications that employ conversational components. In the process, we delineate important aspects of such systems, and provide an overview of the possible design decisions that can be taken, while designing the concrete architectures for such systems. The rest of this paper is organised as follows. In Section II we present a Reference Architecture for the such applications. Next, we apply this architecture to a sample use-case and come up with concrete architectures using two commercially available platforms in Section III. We present a concise reference to some of the related work in the past in Section IV, and conclude the paper in Section V.

II. REFERENCE ARCHITECTURE

We now propose a Reference Architecture for applications with conversational components. We focus only on those components, which are either a part of the Conversational Subsystem or, interact with the same. Figure 1 shows the overview of the Reference Architecture. A brief description of the elements presented in the architecture is as follows.

1) Voice Utilities: If the application provides a speech interface, an important functionality required in the conversational component is to converting a speech sample - spoken by the user - to the corresponding text, and convert the produced text response to an audible sound clip. This is important since the models that work over the natural language inputs require them to be in textual form. Even the commercial solutions that provide an endpoint to take speech inputs, convert them to text implicitly, before any further processing.

2) Intents and Parameters: Intents are the different classes of inputs that the application expects to receive. Parameters are attributes or details in a query that are needed to be parsed out of the inputs to produce a response or perform an action. Table I shows some examples of intents and parameters. An Intent Classifier component classifies a natural language phrase into one of the “pre-defined” intents. Usually, there is a default intent, which is used as a fallback, in case the classifier cannot categorise the input into any meaningful class. Parameters can take many forms. They can either take values from a closed set or can be assigned a value from an open domain. The values can either be present “literally” or may be derived out of the fragments of the input phrase. A Parameter Extractor component is tasked with extracting these values. It is possible that an input phrase does not have any parameters in it as well (e.g. a phrase like “Hey there!”). Most of the commercial solutions choose to keep these techniques “blackboxed”. The developers can provide definitions for intents and parameters, along with some training examples (sample input phrases), and the models are built over them in the background.

3) Response Generator: Similar to a conversation between two human beings, a user usually anticipates a response from the application, for even mundane inputs (like “Hi!”). Also, during a conversation, the user may need the feedback of what input is required next. In some cases, the application must also inform the user that the previous input was unintelligible. A Response Genera-

<table>
<thead>
<tr>
<th>User Input</th>
<th>Intent</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>I want to book a cab</td>
<td>booking</td>
<td>None</td>
</tr>
<tr>
<td>Book a cab for 9 AM today</td>
<td>booking</td>
<td>date=currentDate(), time=0900</td>
</tr>
<tr>
<td>What are your fares</td>
<td>enquiry</td>
<td>detail=fare</td>
</tr>
<tr>
<td>Do you operate at night</td>
<td>enquiry</td>
<td>detail=timing</td>
</tr>
<tr>
<td>Your service is horrible</td>
<td>feedback</td>
<td>sentiment=negative</td>
</tr>
</tbody>
</table>
5) **Context** for the Flow Manager, based on which these decisions are made, are the previous steps, and take the flow to a previous state (e.g. “Shall I confirm your booking?”). The Flow Manager can either initiate seeking a “Yes” or “No” input from the user. Based on the application is one step away from booking a cab, and is was “Shall I confirm your booking?”, it means that the step. For instance, if the last response from the application of the conversation and decide what should be the next of the Flow Manager is to keep track of the current state of the application and the conversational components, is to provide a natural language interface to the user, to interact with the application. The core functionalities provided by the application (e.g. booking a cab), are relatively decoupled from the Conversational subsystem. To provide a bridge between the core functionalities of the application and the conversational components, a set of **Actions and Fulfilments** are defined. Fulfilments are the connection points where the Conversational subsystem connects to the rest of the application. The Flow Manager picks a particular Fulfilment to call, based on the **Event** that has occurred. An event represents a specific condition during the conversation with the user. For example, if the **Flow Manager**: During a natural conversation between two human beings, who is going to speak next, and what would that be, is usually not hard. However, it is not trivial for an application to do the same. Imagine a typical flow, where the user initiates the conversation with “Hi!”. A good option would be to return the greeting with another greeting. So, the application responds with “Hello”. But what should the application do next? Shall it wait for the user to provide another input? Shall it extend the greeting message with a response like “What can I do for you?” This indecision is even severe in the middle of a conversation. These issues point to another important component of the subsystem - the **Flow Manager**. The job of the Flow Manager is to keep track of the current state of the conversation and decide what should be the next step. For instance, if the last response from the application was “Shall I confirm your booking?”, it means that the application is one step away from booking a cab, and is seeking a “Yes” or “No” input from the user. Based on the user’s response, the Flow Manager can either initiate triggering of a fulfilment that culminates in the booking of a cab, or, it can delete the booking data collected in the previous steps, and take the flow to a previous state (say asking for booking details again). The important resource for the Flow Manager, based on which these decisions are made, are the **Context** objects. These objects persist across one or more input-response cycles and capture the state of the application at any given instant.

4) **Actions and Fulfilments**: The job of conversational components, is to provide a natural language interface to the user, to interact with the application. The core functionalities provided by the application (e.g. booking a cab), are relatively decoupled from the Conversational subsystem. To provide a bridge between the core functionalities of the application and the conversational components, a set of **Actions and Fulfilments** are defined. Fulfilments are the connection points where the Conversational subsystem connects to the rest of the application. The Flow Manager picks a particular Fulfilment to call, based on the **Event** that has occurred. An event represents a specific condition during the conversation with the user. For example, if the **Intent Classifier** has detected that the user intends to book a cab, and the Parameter Extractor has extracted all the required details from the user query, the Flow Manager may invoke a fulfilment, that deals with the event “user made a booking request”. A fulfilment, can either produce a response (e.g. “We do not have any cabs right now, please try again in 30 minutes”), or it can trigger an action in the real world (e.g. initiating a pipeline, that involves booking a cab, sending an email, generating a One Time Password etc.). If required, the Flow Manager then invokes Response Generator to generate a response for the event.

Other than the components discussed above, another aspect worth mentioning here is that many commercial platforms offer seamless integration with popular messaging platforms such as Facebook Messenger or Telegram. This means that the application can be deployed as a “bot” over these mediums, and the users can interact with them on the messaging platform. In case the conversational components are to be used in custom solutions, such as a website, or as part of an Android App, the Conversational Subsystem can be connected via **Endpoints**, which can be reached via RESTful APIs [9].

### III. Sample Concrete Architectures

In this section, we apply the reference architecture towards creating two concrete architectures for the same use-case. The differences in the concrete architectures allow us to show some variations in the reference architecture. These variations show differences between two or more concrete architectures, derived out of the same reference architecture. We pick two commercially popular platforms - IBM Watson Assistant [10] and Google Dialogflow [11] and show how the concrete architectures differ from each other when the same application is architected using them. Figure 2 shows these architectures. Here, **Entities** are to **Parameters**, what **Classes** are to **Objects**.

Before we describe the architectures, we revisit our cab booking application example. Being a common use-case, we only summarise the important points, skipping the details:

- The application has two front-end facades, via a Website and through an Android App.
- There are a set of background business processes, performing their expected tasks. For instance, there is a process that, when invoked with necessary inputs, books a cab. Similarly, there may be another process that can record a user’s feedback, and so on.
- Users can book the cabs, receive fare information, provide feedback etc. by browsing to relevant screens or pages. The application also provides a conversational interface to perform these tasks. On the website, the users can type the messages in a chat widget. In the app, users can provide these messages by either typing them in a textbox or say it out loud.

We now briefly describe the concrete architectures for the above use-case, designed using either Dialogflow or Watson Assistant for building the conversational capabilities.
Architecting with Dialogflow: A possible concrete architecture for the application is shown in Figure 2(a). Some points worth mentioning here are:

- Dialogflow merges three components we mentioned in the Reference Architecture - Intent Classifier, Parameter Extractor and Response Generator inside a “blackbox”, that we call the Dialogflow Engine. Although, the engine works on inputs supplied by the developers - intent descriptions, entity details, event definitions, etc., the actual working of the engine is not available for inspection. There is no explicit counterpart for the Flow Manager in Dialogflow. However, developers can control the flow partially by defining additional intents (called follow-up intents) and setting additional context variables.
- Dialogflow provides a unified interface (an API endpoint /detectIntent) for both text as well as speech communication. Under the hood, it converts speech to text and vice-versa before feeding it to the engine. The speech-to-text module is a blackbox too. It is mentioned though that the text-to-speech facility is provided via Wavenet [12].
- Dialogflow provides the option to configure a single fulfillment webhook. This endpoint is responsible for differentiating between the possible events, and invoke any actions (in the form of background processes) if required.

Architecting with Watson Assistant: Figure 2(b) shows a possible architecture for the same use-case with Watson Assistant. A Brief explanation for the architecture is as follows.

- Watson Assistant too, merges the Intent Classifier, Parameter Extractor and Response Builder inside a “blackbox”, that we call the Assistant Engine. Watson Assistant provides a Dialog Tree where rules can be added to “trigger” a node. The Dialog Tree can either executes nodes in order (top child to bottom child, then bottom sibling), or it can jump to specific nodes on encountering specific conditions or events, allowing finely tuned conversations.
- Watson Assistant can work on textual inputs only (over the API endpoint /message). There are two independent services called Watson Speech-to-Text and Watson Text-to-speech, which can be considered as counterparts of Voice Utilities. They can be invoked explicitly for the conversion from speech to text and vice versa, alongside calling the Watson Assistant.
- Watson Assistant doesn’t allow calling an external webhook from the Dialog Tree. Developers, however, can define different IBM cloud functions, to act as fulfilments, which can be invoked from nodes in the tree. Inside the cloud functions, cURL requests can be made to any external endpoints imitating actions.

Comparing the two architectures: The two concrete architectures differ significantly. We discuss a few ramifications of these differences briefly:

1) Dialogflow allows configuration of a single webhook. Watson Assistant may invoke a different cloud function for each event, which can then call a RESTful action. The former approach may be preferable if the actions are not supposed to be exposed. Otherwise, the latter approach may be better for maintenance. It will, however, require that actions have URLs, that can be invoked externally. Also, the involvement of a cloud function adds a redirection, which can increase the response times.

2) Dialogflow provides a single-input, single-output endpoint for both speech as well as text inputs. Watson
Assistant, on the other hand, requires two additional calls, to auxiliary services, in case the input is in speech form. This attains additional overhead for the application, as well as add more out-of-system calls, which are susceptible to network failures.

3) Watson Assistant’s Dialog Tree makes the flow control easily customizable. It also provides features to handle digressions, where a user can deviate from the main conversation temporarily, and return to it after a few exchanges. A classic example is a user replying with an enquiry, when the application is expecting a detail (e.g. the user replying with “Can I get a sedan” when the application was expecting an answer to the question “What time shall we book the cab”). Dialogflow provides limited abilities to control conversation flows, and complex scenarios like digressions cannot be handled.

Dialogflow provides an easier learning curve for beginners, providing common features and the internalising voice utilities. Watson Assistant is useful for building a wider range of applications but requires significant effort up-front.

IV. RELATED WORK

Researchers have attempted to provide Reference Architectures for different classes of applications in the past ([2] [3] [13] [14] etc.). Comparisons between Reference Architectures and Product Line Architectures have also been drawn [15]. The idea of providing Reference Architectures for chatbots or general guidelines about building them have been attempted too in some blogs ([16] [17] [18] etc.). There are some blogs that try to compare popular chatbot building platforms as well ([19] [20] etc.). These works cover a feature-based comparison of these platforms, giving little attention to architectural issues. To the best of our knowledge, an attempt to provide Reference Architectures for applications involving a conversational component has not been addressed in the literature. This is important because conversational components like chatbots usually do not work in isolation, and are part of a broader application. In this work, we provide a decomposition of the chatbot components and give an overview of how they affect the architecture of an application when deployed.

V. CONCLUSION AND FUTURE WORK

The advancements in NLU and AI techniques have opened the doors for adding conversational interfaces to applications. These applications, however, differ significantly from conventional software applications, due to the uncertainty associated with using any Machine Learning model. In the current work, we presented a reference architecture for applications which employ conversational components. We then applied the Reference Architecture to the use-case of a cab booking application, and offered possible Concrete Architectures for the application, when the conversational capabilities are provided with the help of either Google Dialogflow or IBM Watson Assistant. We analysed the differences between the architectures and discussed their utilities in different scenarios.

In future, it would be interesting to inspect other commercial platforms and compare how similar they are to Dialogflow and Watson Assistant. We showed how both these platforms “blackbox” their engines. However, it may be interesting to analyse the behaviour of these platforms, when supplied with the same data and asked the same queries, in an attempt to benchmark them for particular classes of applications. Since the field is evolving rapidly, there may be refinements to the presented Reference Architecture in future, to reflect any other features provided by common chatbot building platforms.

REFERENCES