Chatbot Deployment with IBM Cloud Watson Assistant

Phase-5

Abstract:

This guide outlines the process of deploying a chatbot using IBM Cloud Watson Assistant. It covers account setup, chatbot creation, integration with platforms, and optimization for real-world applications. Practical tips, use cases, and troubleshooting advice are provided for a streamlined deployment experience.

1.IBM Watson Assistant:

Watson Assistant is at the head of the tech industry of conversations through artificial intelligence. Platform is simple and intuitive, with extremely well-created documentation. In terms of costs, it offers a **free plan** (Lite plan), which allows 10,000 messages per month via chatbot, and the ability to integrate the bot with web services.

2.Setup:

- To access the Watson Assitant platform, but also other services, you need to create an account on **IBM Cloud**: https://cloud.ibm.com/registration. After you have created your account and logged in, select *"Watson"* from the side menu, and in the dashboard displayed you will

find in the *"Getting Started"* section, the option *"Build a chatbot"*.

- From here you will be directed to the initial configuration page of the chatbot (pricing plan, region, service name).
- When you're ready, click *"Create"* and that's it! The bot is created and ready to be configured as you wish, through the dedicated platform.

3.Design a hierarchical flow:

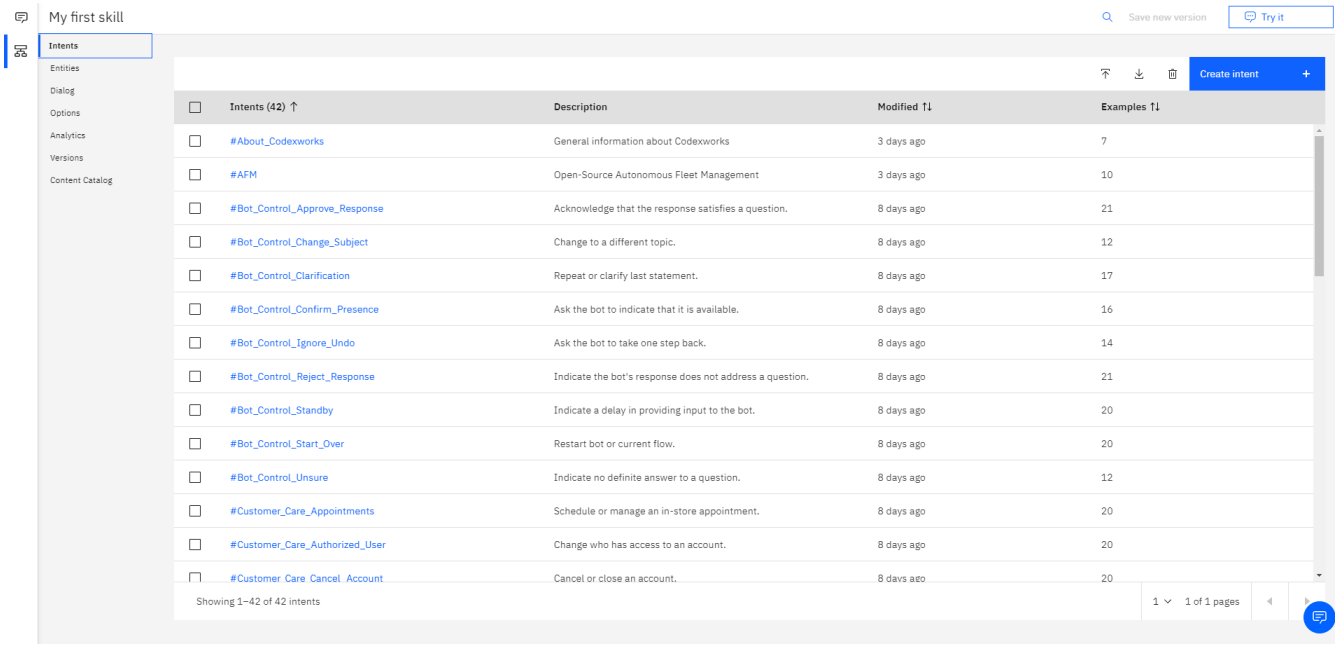
- The process of learning for chatbot involves the development of several **dialogue skills** (up to 5 in the Lite version).
- A dialogue skill consists of **Intents**, **Entities** and **Dialogs**, these 3 components being the basis for learning of virtual assistant, through the IBM Watson Assitant platform.

4.Intents:

- An intent is a **purpose or intention expressed by the user**, such as asking a question - *"How do I schedule a call?"*, or expressing an intention - *"I would like to pay the bill"*.
- By recognizing the desire of the user, **the virtual assistant can choose the appropriate dialogue flow**, and can

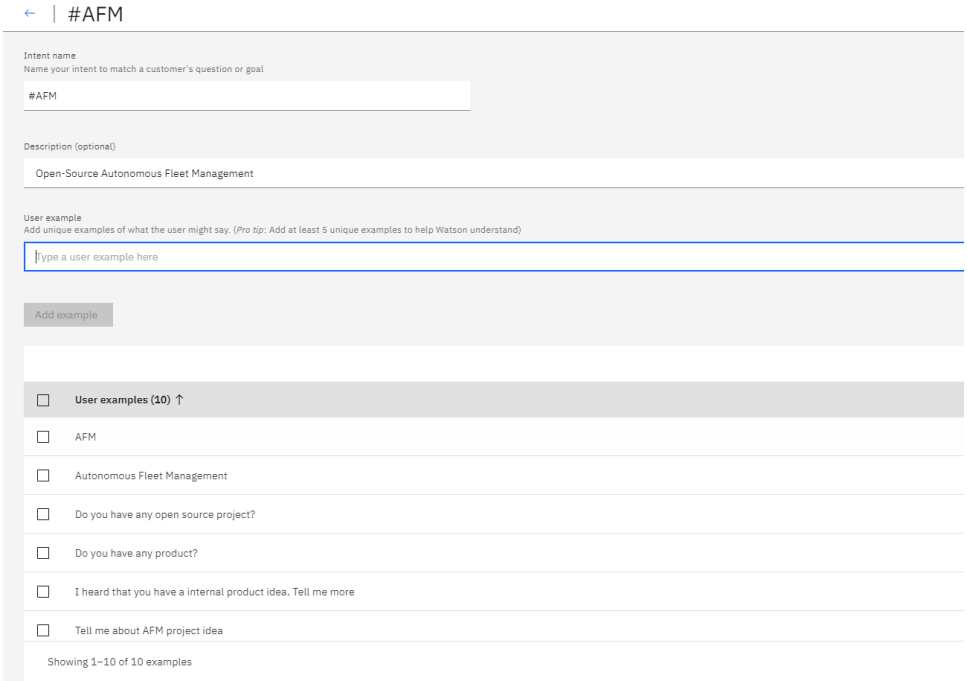
formulate an appropriate response. Most of the time, the user's intention is cataloged by the bot in several categories, giving a probability to each option.

- In the image below you can see several intents defined for the first dialog skill of the bot:



An advantage of using this platform is that it puts available a **predefined range of intents**, which can be used for free. (check *Content Catalog* section). Thus, the **Bot Control** category, and all associated intents, were added and learned by our assistant.

How exactly does an intent look like on the platform? You can see in the image below how an intent was defined for the flow of data responsible for information about **AFM** (Autonomous Fleet Management).



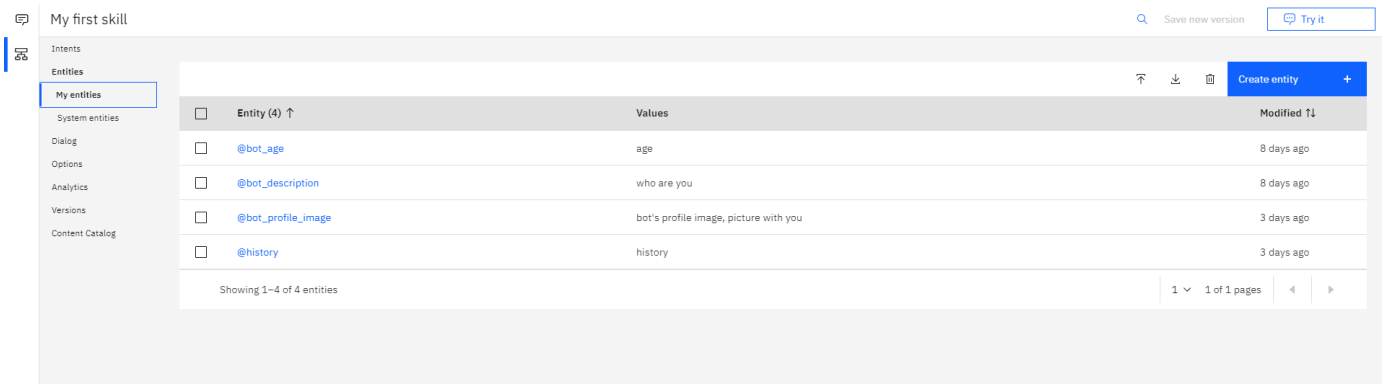
Intents (using IBM Watson Assistant API):

```
import requests

 # Define Intent

        intent_data = {

                "intents": [

                {

                        "intent": "greetings",

                        "examples": [

                        {"text": "Hello"},

                        {"text": "Hi"},

                         # Add more examples ]

                },

                 # Add more intents

        ]
```

```
}
respon=requests.post(
"https://api.ussouth.assistant.watson.cloud.ibm.com/instances/{instance_id}/v2/assistants/{assistant_id}/i ntents",
 headers={"Authorization": "Bearer {api_key}", "Content-Type": "application/json"}, json=intent_data )
```

5.Entities:

- An entity is **a class of data/information that is relevant to a user's purpose in expressing intent**. By recognizing the entities, the virtual assistant can choose specific actions, in acomplish an intention.
- For example, a user may want to know more about **CodexWorks** (*"Tell me more about Codexworks"*). At the same time, he may want information about the history of CodexWorks. (*"Tell me more about Codexworks's history"*). There is a difference between the two answers that must be provided. **History is an entity**, changing the context of the intention provided. For the user, the history becomes relevant, not the general information about CodexWorks.
- In the image below you can see some entities defined within the platform:



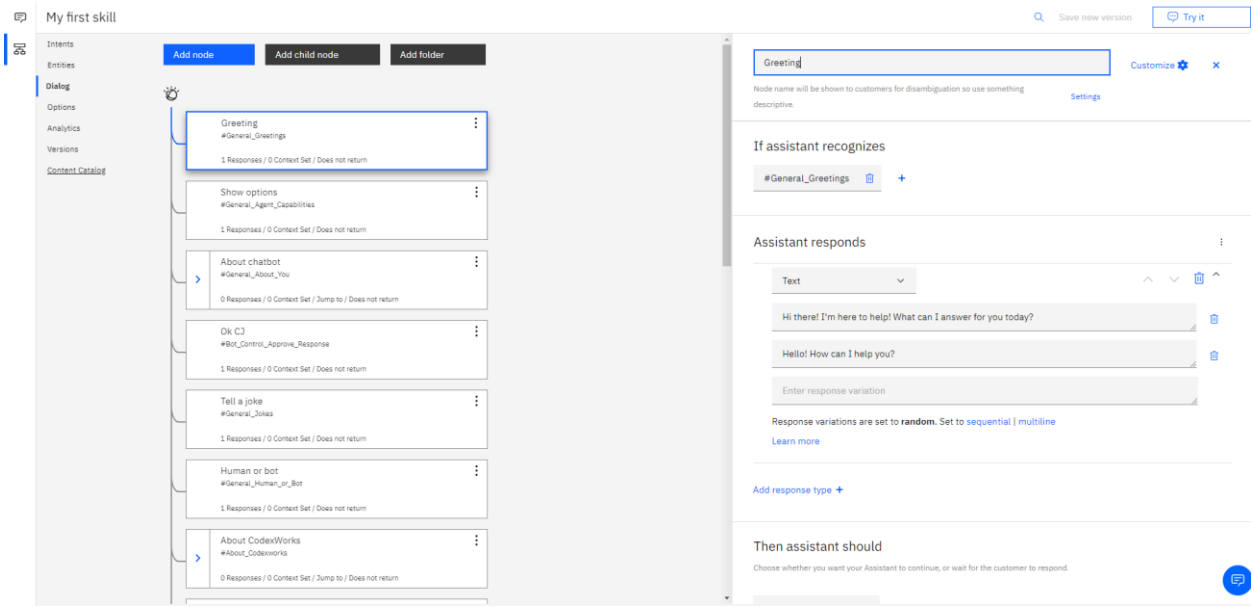
Entities (using IBM Watson Assistant API):

```
# Define Entity Type
 entity_data = {
        "entities": [
                {
                        "entity": "location",
                        "values": [
                 {
                         "value": "New York",
                        "synonyms": ["NY", "NYC"]
                },
                 # Add more values
        ]
},
 # Add more entity types
 ] } response = requests.post(
"https://api.ussouth.assistant.watson.cloud.ibm.com/instances/{instance_id}/v2/assistants/{assistant_id}/ entities",
```

headers={"Authorization": "Bearer {api_key}", "Content-Type": "application/json"}, json=entity_data)

6.Dialogs:

- A dialog is **the component that uses discovered Intents and Entities based on inputs provided by user, in order to find and provide an appropriate response**. The dialogs have a linear tree structure. Thus, a branch is created for each intent (or group of intents), together with the associated entities. Each dialog offers the possibility to prepare several answers, being provided only one answer (chosen sequentially, or randomly).
- The image below shows a part of the **tree structure** of the dialogues for our bot, in which a well-established order of them can be observed. On the right you can see how the dialog for the **Greetings** use case is defined, and the possible answers that the bot can offer, if it recognizes in the user's input a form of greeting.



Dialog Nodes (using IBM Watson Assistant API):

```
# Define Dialog Node
 dialog_data = {
 "dialog_nodes": [
{ "dialog_node": "greetings",
 "conditions": "#greetings",
"output": {
"generic": [
 { "response_type": "text",
 "values": [ { "text": "Hello! How can I assist you today?" } ] } ] } },
 # Add more dialog nodes
 ]
 }
response = requests.post(
"https://api.ussouth.assistant.watson.cloud.ibm.com/instances/{instance_id}/v2/assistants/{assistant_id}/ dialog_nodes",
 headers={"Authorization": "Bearer {api_key}", "Content-Type": "application/json"}, json=dialog_data )
```

Interact with your chatbot:

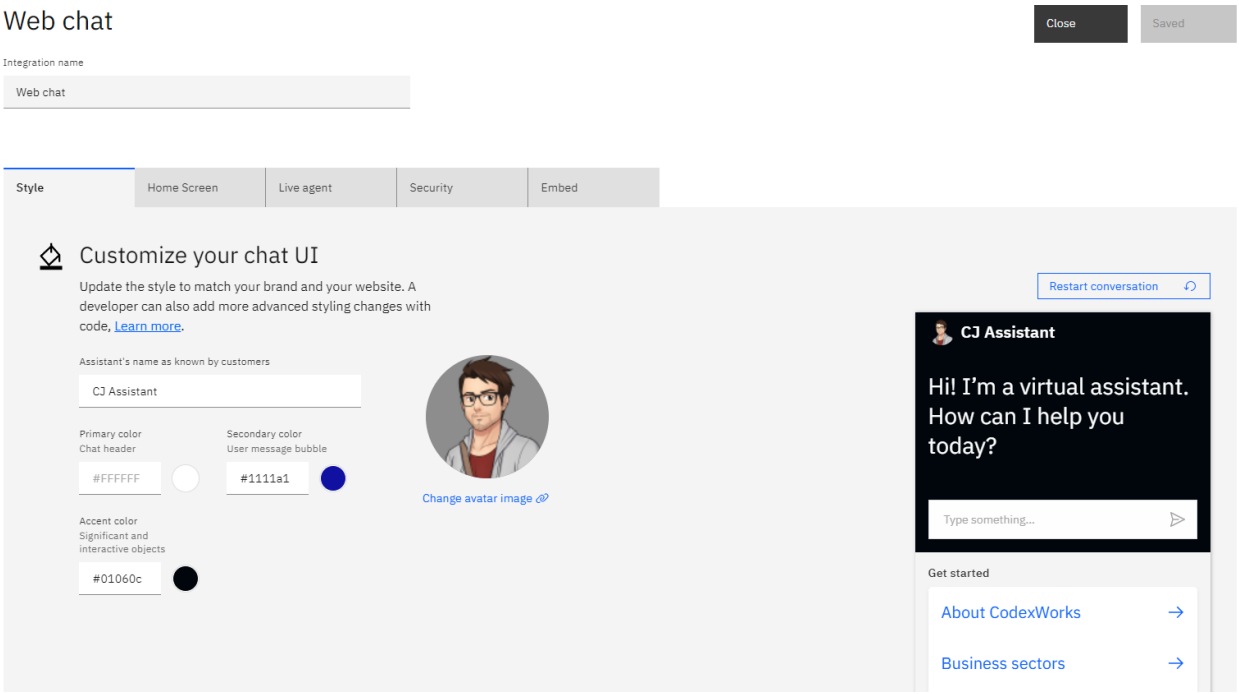
Test:

- After defining a few dialogues, the bot can already be successfully tested! His training is done **in real time**, immediately after adding a new element, of any type.

- In the upper right corner you can see the *Try it* button that opens the chat for the bot you created, with which you can interact and see in which category maps the input (question or intention) provided by the user.

Deploy:

- An extremely valuable advantage of these platforms is that **the deployment process is very simple**. To create a deployment channel, go to the main panel of the platform and select *Integrations*. From there you will be able to create a new integration, with whatever service you want. The most common is Web chat.



Step 1: Integrate with Facebook Messenger:

To integrate your chatbot with Facebook Messenger, follow these steps:

1. Set Up a Facebook App and Page

- Create a Facebook App on the [Facebook for Developers portal] (https://developers.facebook.com/apps/).
- Create a Facebook Page if you don't already have one.

2. Obtain App Credentials

- Get your App ID and App Secret from the Facebook App settings.

3. Set Up Webhooks

- Configure a webhook for your Facebook App to receive messages. This webhook will forward messages to your chatbot.

Code:

```
# This is typically done through the Facebook Developer Portal

# You'll need to provide a URL for your webhook endpoint (e.g., using Flask, Django, etc.)

# Verify the request using the provided verification token

# Example Flask endpoint

@app.route('/webhook', methods=['GET'])

def verify_webhook():

    verify_token = 'your_verification_token'

    if request.args.get('hub.verify_token') == verify_token:

        return request.args.get('hub.challenge')

    return 'Invalid verification token'
```

4. Implement Webhook Endpoint (using a web server or cloud function)

- Create an endpoint to handle incoming messages from Facebook. This will be the URL you provided in the webhook setup.

5. Process Incoming Messages

- Parse the incoming messages from Facebook and send them to your Watson Assistant using its API.

Code:

```
# Parse incoming messages and extract sender ID and message text

# Send the message text to Watson Assistant for processing

# Example Flask endpoint

@app.route('/webhook', methods=['POST'])

def receive_message():

    data = request.get_json()

    for entry in data['entry']:

        for messaging_event in entry['messaging']:

            sender_id = messaging_event['sender']['id']

            message_text = messaging_event['message']['text']

            # Send message_text to Watson Assistant for processing
```

6. Send Responses to Facebook Messenger

- Once you receive a response from Watson Assistant, send it back to Facebook Messenger using the Messenger API.

Code:

```
# Once you receive a response from Watson Assistant, send it back to Facebook Messenger

def send_message(sender_id, message_text):
```

```
data = {
    'recipient': {'id': sender_id},
    'message': {'text': message_text}
}
response = requests.post(
    'https://graph.facebook.com/v13.0/me/messages',
    params={'access_token': 'your_page_access_token'},
    json=data
)
```

Step 2: Integrate with Slack:

To integrate your chatbot with Slack, follow these steps:

1. Set Up a Slack App
   - Create a Slack App on the [Slack App Directory](https://api.slack.com/apps).
2. Obtain App Credentials
   - Get your Slack App credentials, including the Client ID, Client Secret, and Verification Token.
3. Set Up OAuth & Permissions
   - Configure OAuth & Permissions in your Slack App settings to allow your app to interact with Slack workspaces.
4. Implement OAuth Flow (if necessary)
   - If you're using OAuth, implement the OAuth flow to authenticate your app with Slack.

5. Set Up Event Subscriptions

- Configure event subscriptions to receive messages from Slack.

Code:

```
# You'll need to set up an event subscription for message events in your Slack App settings

# Define an endpoint to receive Slack events

# Example Flask endpoint

@app.route('/slack/events', methods=['POST'])

def slack_events():

    data = request.get_json()

    if 'event' in data and 'type' in data['event'] and data['event']['type'] == 'message':

        user_id = data['event']['user']

        message_text = data['event']['text']

        # Send message_text to Watson Assistant for processing
```

6. Process Incoming Messages

- Parse incoming messages from Slack and send them to your Watson Assistant using its API.

Code:

```
# Parse incoming messages and extract user ID and message text

# Send the message text to Watson Assistant for processing

# Example Flask endpoint (continued)

@app.route('/slack/events', methods=['POST'])

def slack_events():
```

```
    data = request.get_json()
    if 'event' in data and 'type' in data['event'] and data['event']['type'] == 'message':
        user_id = data['event']['user']
        message_text = data['event']['text']
        # Send message_text to Watson Assistant for processing
```

7. Send Responses to Slack

- Once you receive a response from Watson Assistant, send it back to Slack using Slack's API.

Code:

```
# Once you receive a response from Watson Assistant, send it back to Slack
def send_message_to_slack(user_id, message_text):
    payload = {
        'token': 'your_bot_user_access_token',
        'channel': user_id,
        'text': message_text
    }
    response = requests.post('https://slack.com/api/chat.postMessage', data=payload)
```

Step 3: Refine Responses

Refining responses involves:

- Context Management: Ensure the chatbot maintains context for multi-turn conversations.
- Natural Language Processing (NLP): Continuously train and improve your Watson Assistant to understand user queries better.
- Fallback Mechanism: Enhance the fallback responses to handle ambiguous or unclear user input.