**DOMAIN NAME: CLOUD APPLICATION DEVELOPMENT**

**PROJECT NAME: CHATBOT WITH WATSON**

**Phase 5 Submission Document**

**CONTENTS:**

1. ABSTRACT

2. PROBLEM STATEMENT

• BACKGROUND

• PROBLEM

• KEY ISSUES

3. PROJECT OBJECTIVES

4. SYSTEM OVERVIEW

• SYSTEM ARCHTECTURE

• SYSTEM WORKFLOW

5. HARDWARE AND SOFTWARE COMPONENTS

• DEVELOPMENT MACHINE

• SERVER FOR HOSTING

• IBM CLOUD ACCOUNT

• IBM WATSON ASSISTANT

6. SYSTEM CONFIGURATION AND DEPLOYMENT

• CONFIGURATION OF IBM WATSON ASSISTANT

• DEVELOPMENT ENVIRONMENT SETUP

• WEB SERVER CONFIGURATION

7. SYSTEM OPERATION AND MAINTAINENCE

• MONITORING AND MANAGING IoT DEVICES

• UPDATING THE TRAFFIC MANAGEMENT SOFTWARE

8. USE CASES AND APPLICATIONS

9. EVALUATION AND PERFORMANCE ANALYSIS

10. CONCLUSION

**ABSTRACT:-**

The main aim of this paper is to show how to use IBM Watson tools to create a domain-specific chatbot and figure out the importance and effectiveness of some tools by IBM Watson to evaluate their accuracy. This paper also tries to explain the advantages and disadvantages of various tools also showing some methods that can be used to improve the chatbot system or particular modules of the chatbot.

**INTRODUCTION:-**

Chatbots are the most important but still evolving feature for a website. This technology is also getting integrated with new generation hardware devices to improve autonomous tasks and reduce human effort. The integration of a chatbot with current other applications is possible because of recent research on ”human parity level speech detection” and smarter sentiment analysis.

Traditionally, actual humans were operating chatbots after sending the first default message to the end-user. Nevertheless, now, many approaches can make these chatbot implementations easy for developers. There are two types of chatbots, the first type is domain-specific, which is also called machinedriven dialog systems.

In this type of system, the end-user needs to follow the instructions given by machine, and this type of system can handle only some domain-specific scenarios, and the second type of chatbot is a general-purpose chatbot that can handle various domains at the same time. In the current era, there are various approaches available for developers to implement a chatbot, for example, machine learning approaches that include semantic parsing, tone analyzer, etc. or Rulebased approaches that we will discuss in detail in next sections. To create a chatbot, the developer needs various tools to handle the user’s input and process it into the required output.

There are many open-source and paid systems available in the market, some of the open-source systems are Microsoft Bot Framework, Rasa, Botpress, ANA Chat, and some of the paid systems are IBM Watson, Amazon Lex and many more. This paper explains the methods to create a basic chatbot using one of the various approaches to creating dialog systems. It includes the explanation of various architectural tools required for creating a chatbot system.

**PROBLEM STATEMENT:**

**Background:**

The chatbot can contribute to a richer customer experience by offering personalized responses and product recommendations, thereby increasing customer satisfaction and loyalty. With continuous learning and improvement, the chatbot can adapt to evolving user needs and industry trends.

**PROBLEM:**

Inefficiencies in Customer Support Operations for require a scalable and intelligent solution to streamline interactions, reduce response times, and enhance customer satisfaction.In the rapidly evolving landscape of customer support plays a pivotal role in ensuring customer satisfaction and loyalty. However, the current customer support operations are facing several challenges.

**KEY ISSUES:**

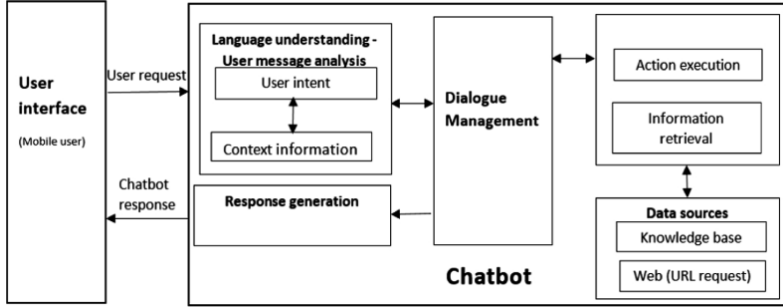
1. **Automate Routine Tasks:** The chatbot will handle routine inquiries, such as FAQs and basic troubleshooting, freeing up human agents to focus on more complex and high-value tasks.
2. **Improve Response Times:** By providing instant responses, the chatbot will significantly reduce customer wait times, leading to higher satisfaction and better service quality.
3. **Enhance Consistency:** The chatbot will ensure consistent and accurate responses across all interactions, adhering to company policies and providing a uniform customer experience.
4. **Scale Efficiently:** The chatbot will be scalable, able to handle a growing customer base without the need for a linear increase in human support agents.
5. **Adapt and Learn:** Utilizing Watson's NLP capabilities, the chatbot will continually learn and adapt to evolving customer needs and preferences, improving its performance over time.

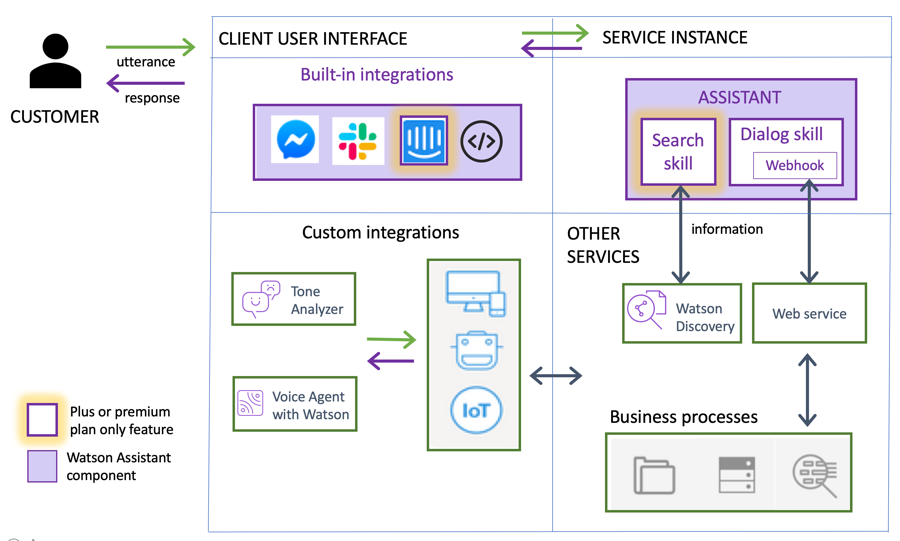
**PROJECT OBJECTIVES:**

1. **User Engagement:** Build a chatbot that can effectively engage with users in natural language, understand their queries, and provide relevant responses.
2. **Problem Solving:** Enable the chatbot to address common user queries, issues, or tasks efficiently, offering solutions or guidance in real-time.
3. **Seamless Integration:** Integrate the chatbot with [Your Platform/Channels] to ensure users can access it through multiple touchpoints such as websites, mobile apps, and social media.
4. **Personalization:** Implement user profiling and personalization features to tailor responses based on user history, preferences, and interactions.
5. **Knowledge Base:** Utilize Watson's NLP capabilities to continually expand the chatbot's knowledge base, ensuring it can answer a broad range of questions accurately.
6. **Analytics and Reporting:** Incorporate data analytics to gather insights into user interactions, track performance, and identify areas for improvement.
7. **Security and Compliance:** Ensure that user data is handled securely and that the chatbot complies with relevant privacy and data protection regulations.
8. **Scalability:** Design the chatbot architecture to handle increasing user loads as adoption grows, without compromising performance.
9. **User Training:** Implement features for ongoing user training and feedback to improve the chatbot's conversational abilities.
10. **User Support:** Provide seamless handoff to human agents when the chatbot cannot fulfill user requests, maintaining a smooth user experience.
11. **Monitoring and Maintenance:** Establish a robust monitoring system to detect and address any issues or anomalies promptly, and maintain the chatbot's relevance over time.

**SYSTEM** **OVERVIEW**:

**• SYSTEM ARCHITECTURE:**

****



**• SYSTEM WORKFLOW:**

An architecture flow for a chatbot using IBM Cloud Assistant would illustrate the flow of information and interactions between different components. Below is a simplified textual representation of the architecture flow for such a chatbot

1. **User Interaction**:
   * The flow begins when a user interacts with the chatbot through a messaging platform, such as a web interface or a messaging app.
2. **Messaging Platform**:
   * The user's message is received by the messaging platform, which may include Facebook Messenger, Slack, or a custom web chat interface.
3. **Webhook**:
   * The messaging platform can be configured to send incoming messages to a webhook.
   * The webhook is a server-side component responsible for processing and forwarding user messages.
4. **IBM Watson Assistant**:
   * The webhook or messaging platform forwards the user's message to IBM Watson Assistant.
5. **IBM Watson Assistant Processing**:
   * IBM Watson Assistant processes the user's message using natural language understanding.
   * It identifies intents, entities, and context to understand the user's request.
6. **Dialog Tree**:
   * IBM Watson Assistant uses the dialog tree to generate an appropriate response.
   * Dialog nodes may be used to customize responses based on the context.
7. **Response Generation**:
   * IBM Watson Assistant generates a response, which is typically text-based.
   * The response is sent back to the webhook or messaging platform.
8. **Webhook** :
   * If a webhook was used earlier, it can further process the response from IBM Watson Assistant.
   * It may perform additional actions or make external API calls.
9. **Messaging Platform**:
   * The final response is delivered to the user through the messaging platform.
   * The user sees the chatbot's response in the messaging interface.
10. **User Interaction :**
    * The user can continue the interaction, sending messages back to the chatbot.
    * The flow loops back to the "User Interaction" step, repeating the process for each user input.

This flow demonstrates the high-level interactions within the chatbot architecture. The specific implementation details, such as the use of additional services, authentication and security measures, and database interactions, can vary based on the requirements of your chatbot project.

**HARDWARE AND SOFTWARE COMPONENTS:**

Building a chatbot with IBM Watson Assistant primarily involves software components, as the chatbot's intelligence and interaction are handled by the cloud-based service. However, you might need specific hardware and software for development and integration. Here are the typical hardware and software requirements:

**Hardware Requirements:**

1. **Development Machine**:
   * You'll need a computer for developing, testing, and deploying the chatbot.
   * The specific hardware requirements depend on your development environment, but a modern laptop or desktop computer is typically sufficient.
2. **Server for Hosting**:
   * If you choose to self-host your chatbot or require dedicated servers, you'll need server hardware.
   * The server's hardware specifications will depend on the expected load, but it should have sufficient CPU, RAM, and storage.

**Software Requirements:**

1. **IBM Cloud Account**:
   * You'll need an IBM Cloud account to access the Watson Assistant service.
2. **IBM Watson Assistant**:
   * The core software component is IBM Watson Assistant, a cloud-based chatbot development platform. You can access this service from your IBM Cloud account.
3. **Development Environment**:
   * You'll need a development environment to write and test your chatbot's code. Common choices include:
     + **IDE (Integrated Development Environment)**: Such as Visual Studio Code, PyCharm, or similar tools, depending on your chosen programming language.
     + **Programming Language**: Depending on your platform and preference, you may use Python, JavaScript, Node.js, Java, or other languages for webhooks and integration.
4. **Web Server**:
   * If you're self-hosting your chatbot, you'll need a web server software such as Apache, Nginx, or a serverless platform like AWS Lambda or IBM Cloud Functions.
5. **Database System**:
   * If your chatbot needs to store or retrieve data, you may need a database system. IBM Cloud provides various database services like Db2, Cloudant, and others.
6. **Version Control System**:
   * It's essential to use version control software like Git to track changes in your chatbot's code and collaborate with a team.
7. **SDKs and Libraries**:
   * Depending on the programming language you choose, you might need SDKs and libraries for integrating with IBM Watson Assistant or other services.
8. **Monitoring and Analytics Tools**:
   * Implement monitoring and analytics tools to track your chatbot's performance. This may include services like IBM Log Analysis and third-party analytics solutions.
9. **Continuous Integration/Continuous Deployment (CI/CD) Tools**:
   * CI/CD tools like Jenkins, Travis CI, or GitLab CI/CD can automate your deployment and testing processes.
10. **Testing Frameworks**:
    * Depending on your programming language, you might use testing frameworks to ensure your chatbot functions correctly.
11. **Documentation and Collaboration Tools**:
    * Tools for documenting your chatbot's functionality and for collaborating with your team. This may include Google Docs, Confluence, or other documentation platforms.
12. **Messaging Platform**:
    * If you intend to integrate your chatbot with messaging platforms like Facebook Messenger, Slack, or custom web chat interfaces, you'll need accounts on these platforms.
13. **User Feedback and Support Channels**:
    * Software for collecting and managing user feedback and support channels. Tools like Zendesk or Help Scout can be useful for this.

These hardware and software requirements can vary based on your specific project's complexity, scale, and the features you intend to include in your chatbot. IBM Watson Assistant simplifies the development process, making it more accessible to developers without the need for complex infrastructure.

**SYSTEM CONFIGURATION AND DEPLOYMENT:**

System configuration and deployment for a chatbot using IBM Watson Assistant involve setting up the required environment and resources to make your chatbot accessible to users. Below are the key steps for system configuration and deployment:

**1. Configuration of IBM Watson Assistant:**

* **Create an Assistant**: Log in to your IBM Cloud account, create a Watson Assistant service, and create an assistant. Configure the assistant's name, language, and settings.
* **Add Skills**: Create or import skills into your assistant. Skills include intents, entities, and dialog nodes. Train your assistant with sample user queries to improve its understanding.

**2. Development Environment Setup:**

* **Programming Language**: Choose a programming language (e.g., Python, Node.js, Java) to build the integration between your chatbot and Watson Assistant.
* **IBM Watson SDK**: Use the appropriate SDK or library to interact with IBM Watson Assistant. IBM provides SDKs for various languages.

**3. Integration with Messaging Platforms:**

* Connect your chatbot with messaging platforms such as Facebook Messenger, Slack, or a custom web chat interface.
* Configure webhooks or APIs provided by the messaging platform to communicate with your chatbot.

**4. Web Server Configuration (if self-hosted):**

* If you are self-hosting your chatbot, set up a web server (e.g., Apache, Nginx) to serve your application.
* Ensure the server is properly configured, including security measures such as HTTPS.

**5. Environment Variables and Secrets Management:**

* Store API keys, credentials, and other sensitive information in environment variables or use a secrets management service.
* Access these variables securely in your code to protect sensitive data.

**6. Database Configuration (if required):**

* If your chatbot needs to interact with a database, configure the database connection.
* Ensure the database is accessible and that you have the necessary access rights.

**7. Continuous Integration/Continuous Deployment (CI/CD):**

* Set up CI/CD pipelines to automate the deployment process.
* Use a CI/CD tool like Jenkins, Travis CI, or GitLab CI/CD to ensure that code changes are tested and deployed seamlessly.

**8. Monitoring and Logging:**

* Implement monitoring and logging tools to track the performance and usage of your chatbot.
* Use tools like IBM Log Analysis or third-party monitoring solutions to monitor system health and user interactions.

**9. Error Handling and Recovery:**

* Develop error handling mechanisms to gracefully manage issues and provide users with a smooth experience.
* Implement recovery strategies for service interruptions or chatbot failures.

**10. Scalability and Auto-Scaling (if necessary):**

* Monitor the system's performance, and set up auto-scaling policies to handle increased traffic.

**11. User Authentication and Security (if required):**

* Implement authentication and security measures if your chatbot handles sensitive data or user validation.

**12. User Feedback and Support Channels:**

* Set up channels for collecting and managing user feedback, and establish support mechanisms.

**13. Documentation and Training:**

* Document your chatbot's capabilities, integration steps, and best practices for your team and users.

**14. User Acceptance Testing:**

* Conduct user acceptance testing to ensure the chatbot meets user requirements and performs as expected.

**15. Deployment:**

* Deploy your chatbot to a live environment accessible to users.
* Ensure the chatbot is properly configured for the messaging platforms it will be used on.

These steps may vary based on your specific project requirements and the platforms you're integrating with. Make sure to test and monitor your chatbot continuously to ensure it functions as intended and delivers a great user experience.

**LIBRARIES:**

* **WATSON ASSISTANT SETUP**

/\*\*

\* Copyright 2017 IBM Corp. All Rights Reserved.

\*

\* Licensed under the Apache License, Version 2.0 (the 'License'); you may not

\* use this file except in compliance with the License. You may obtain a copy of

\* the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an 'AS IS' BASIS, WITHOUT

\* WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the

\* License for the specific language governing permissions and limitations under

\* the License.

\*/

'use strict';

require('dotenv').config({

silent: true,

});

/\*\*

\* Setup for Watson Assistant.

\*

\* @param {Object} assistantClient - Assistant client

\* @constructor

\*/

function WatsonAssistantSetup(assistantClient) {

this.assistantClient = assistantClient;

}

/\*\*

\* Automatically create the Assistant skill if not provided.

\* Returns the skillID or error.

\*

\* If a SKILL\_ID is specified in the runtime environment,

\* just use it -- no validation because we might have auth issues.

\* If no SKILL\_ID is specified then try to find it using a lookup by name.

\* Name will be taken from params.default\_name unless overridden

\* using the SKILL\_NAME environment variable.

\*

\* If a skill is not found by ID or name, then try to

\* create one from the JSON in the repository. Use the

\* name as mentioned above so future lookup will find what

\* was created. This requires a Manager role for the API Key

\* and no longer works w/ the CloudFoundry binding key.

\*

\* @param {Object} params - Parameter dictionary as follows.

\* @param {String} params.default\_name - Name of app, used as default skill name when needed

\* to create/find (can be overriden by process.env.SKILL\_NAME).

\* @param {Object} params.skill\_json - The skill JSON to import.

\* @param {function{Error,String}} callback - A callback to capture Error or skill ID string.

\*/

WatsonAssistantSetup.prototype.setupAssistantWorkspace = function (params, callback) {

let skillID = process.env.SKILL\_ID;

if (skillID) {

console.log('Using configured SKILL\_ID: ' + skillID);

callback(null, skillID);

return;

}

this.assistantClient.listWorkspaces(null, (err, data) => {

if (err) {

console.warn('Error during Watson Assistant listWorkspaces(): ', err);

callback(new Error('Unable to list skills for Watson Assistant. ' + err));

} else {

const workspaces = data.result['workspaces'];

// Find by name, because we probably created it earlier (below) and want to use it on restarts.

const skillName = process.env.SKILL\_NAME || params.default\_name;

console.log('Looking for Watson Assistant skill by name: ', skillName);

for (let i = 0, size = workspaces.length; i < size; i++) {

if (workspaces[i]['name'] === skillName) {

console.log('Found Watson Assistant skill: ', skillName);

skillID = workspaces[i]['workspace\_id'];

callback(null, skillID);

return;

}

}

console.log('Creating Watson Assistant skill ', skillName);

const ws = params.skill\_json;

ws['name'] = skillName;

this.assistantClient.createWorkspace(ws, function (err, ws) {

if (err) {

callback(new Error('Failed to create Watson Assistant skill: ' + err));

} else {

skillID = ws.result['workspace\_id'];

console.log('Successfully created Watson Assistant skill');

console.log(' Name: ', ws.result['name']);

console.log(' ID:', skillID);

callback(null, skillID);

}

});

}

});

};

module.exports = WatsonAssistantSetup;

* **WATSON RECOVERY SETUP**

/\*\*

\* Copyright 2017 IBM Corp. All Rights Reserved.

\*

\* Licensed under the Apache License, Version 2.0 (the 'License'); you may not

\* use this file except in compliance with the License. You may obtain a copy of

\* the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an 'AS IS' BASIS, WITHOUT

\* WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the

\* License for the specific language governing permissions and limitations under

\* the License.

\*/

'use strict';

require('dotenv').config({

silent: true,

});

const fs = require('fs'); // file system for loading JSON

/\*\*

\* Setup for Watson Discovery.

\*

\* @param {Object} params - Params needed to

\* @param {Object} callback - Discovery client

\* @constructor

\*/

function WatsonDiscoverySetup(discoveryClient) {

this.discoveryClient = discoveryClient;

}

/\*\*

\* Get the default Discovery configuration.

\* @param {Object} params - Discovery params so far. Enough to get configurations.

\* @return {Promise} Promise with resolve({enhanced discovery params}) or reject(err).

\*/

WatsonDiscoverySetup.prototype.getDiscoveryConfig = function (params) {

return new Promise((resolve, reject) => {

this.discoveryClient.listConfigurations(params, (err, data) => {

if (err) {

if (err.code === 501) {

// this feature is disabled in newer versions of Disco, which is running on CPD

return resolve(params);

} else {

console.error(err);

return reject(new Error('Failed to get Discovery configurations.'));

}

} else {

const configs = data.result.configurations;

for (let i = 0, size = configs.length; i < size; i++) {

const config = configs[i];

if (config.name === 'Default Configuration') {

params.configuration\_id = config.configuration\_id;

return resolve(params);

}

}

return reject(new Error('Failed to get default Discovery configuration.'));

}

});

});

};

/\*\*

\* Find the Discovery environment.

\* If a DISCOVERY\_ENVIRONMENT\_ID is set then validate it or error out.

\* Otherwise find it by name (DISCOVERY\_ENVIRONMENT\_NAME). The by name

\* search is used to find an environment that we created before a restart.

\* If we don't find an environment by ID or name, we'll use an existing one

\* if it is not read\_only. This allows us to work in free trial environments.

\* @return {Promise} Promise with resolve({environment}) or reject(err).

\*/

WatsonDiscoverySetup.prototype.findDiscoveryEnvironment = function (params) {

return new Promise((resolve, reject) => {

this.discoveryClient.listEnvironments(params, (err, data) => {

if (err) {

console.error(err);

return reject(new Error('Failed to get Discovery environments.'));

} else {

const environments = data.result.environments;

// If a DISCOVERY\_ENVIRONMENT\_ID is set, validate it and use it (or fail).

const validateID = process.env.DISCOVERY\_ENVIRONMENT\_ID;

// Otherwise, look (by name) for one that we already created.

const DISCOVERY\_ENVIRONMENT\_NAME = process.env.DISCOVERY\_ENVIRONMENT\_NAME || params.default\_name;

// Otherwise we'll reuse an existing environment, if we find a usable one.

let reuseEnv;

console.log('environments[0]: ' + JSON.stringify(environments[0], null, 2));

for (let i = 0, size = environments.length; i < size; i++) {

const environment = environments[i];

console.log('environment: ' + JSON.stringify(environment, null, 2));

if (validateID) {

if (validateID === environment.environment\_id) {

console.log('Found Discovery environment using DISCOVERY\_ENVIRONMENT\_ID.');

console.log(environment);

params.environmentId = environment.environment\_id;

return resolve(params);

}

} else {

if (environment.name === DISCOVERY\_ENVIRONMENT\_NAME) {

console.log('Found Discovery environment by name.');

console.log(environment);

params.environmentId = environment.environment\_id;

return resolve(params);

} else if (!environment.read\_only) {

reuseEnv = environment;

}

}

}

if (validateID) {

return reject(new Error('Configured DISCOVERY\_ENVIRONMENT\_ID=' + validateID + ' not found.'));

} else if (reuseEnv) {

console.log('Found existing Discovery environment to use: ', reuseEnv);

params.environmentId = reuseEnv.environment\_id;

return resolve(params);

}

// Not found by ID or name or reuse stategy.

// Set the expected name, so when one is created we will find it.

params.name = DISCOVERY\_ENVIRONMENT\_NAME;

return resolve(params);

}

});

});

};

/\*\*

\* Find the Discovery collection.

\* If a DISCOVERY\_COLLECTION\_ID is set then validate it or error out.

\* Otherwise find it by name (DISCOVERY\_COLLECTION\_NAME). The by name

\* search is used to find collections that we created before a restart.

\* @param {Object} params - Object discribing the existing environment.

\* @return {Promise} Promise with resolve({discovery params}) or reject(err).

\*/

WatsonDiscoverySetup.prototype.findDiscoveryCollection = function (params) {

return new Promise((resolve, reject) => {

this.discoveryClient.listCollections(params, (err, data) => {

if (err) {

console.error(err);

return reject(new Error('Failed to get Discovery collections.'));

} else {

// If a DISCOVERY\_COLLECTION\_ID is set, validate it and use it (or fail).

// Otherwise, look (by name) for one that we already created.

const validateID = process.env.DISCOVERY\_COLLECTION\_ID;

const DISCOVERY\_COLLECTION\_NAME = process.env.DISCOVERY\_COLLECTION\_NAME || params.default\_name;

const collections = data.result.collections;

for (let i = 0, size = collections.length; i < size; i++) {

const collection = collections[i];

if (validateID) {

if (validateID === collection.collection\_id) {

console.log('Found Discovery collection using DISCOVERY\_COLLECTION\_ID.');

console.log(collection);

params.collection\_name = collection.name;

params.collectionId = collection.collection\_id;

return resolve(params);

}

} else if (collection.name === DISCOVERY\_COLLECTION\_NAME) {

console.log('Found Discovery collection by name.');

console.log(collection);

params.collection\_name = collection.name;

params.collectionId = collection.collection\_id;

return resolve(params);

}

}

if (validateID) {

return reject(new Error('Configured DISCOVERY\_COLLECTION\_ID=' + validateID + ' not found.'));

}

// No collection\_id added, but return params dict. Set the name to use to create a collection.

console.log('no collection found - must create');

params.collection\_name = DISCOVERY\_COLLECTION\_NAME;

return resolve(params);

}

});

});

};

/\*\* Create a Discovery environment if we did not find one.

\* If an environment is passed in, then we already have one.

\* When we create one, we have to create it with our known name

\* so that we can find it later.

\* @param {Object} params - Object describing the environment we found or need.

\* @return {Promise} Promise with resolve(environment) or reject(err).

\*/

WatsonDiscoverySetup.prototype.createDiscoveryEnvironment = function (params) {

if (params.environmentId) {

return Promise.resolve(params); // If we have an ID, then the env must exist.

}

return new Promise((resolve, reject) => {

// No existing environment found, so create it.

// NOTE: The number of environments that can be created

// under a trial Bluemix account is limited to one per

// organization. That is why have the "reuse" strategy above.

console.log('Creating discovery environment...');

const createParams = {

name: params.name,

description: 'Discovery environment created by ' + params.default\_name,

size: 'LT',

};

this.discoveryClient.createEnvironment(createParams, (err, data) => {

if (err) {

console.error('Failed to create Discovery environment.');

return reject(err);

} else {

console.log(data);

params.environmentId = data.environment\_id;

resolve(params);

}

});

});

};

/\*\*

\* Create a Discovery collection if we did not find one.

\* If params include a collection\_id, then we already have one.

\* When we create one, we have to create it with our known name

\* so that we can find it later.

\* @param {Object} params - All the params needed to use Discovery.

\* @return {Promise}

\*/

WatsonDiscoverySetup.prototype.createDiscoveryCollection = function (params) {

if (params.collectionId) {

return Promise.resolve(params);

}

return new Promise((resolve, reject) => {

// No existing environment found, so create it.

console.log('Creating discovery collection...');

const createCollectionParams = {

name: params.collection\_name,

description: 'Discovery collection created by watson-banking-chatbot.',

language\_code: 'en\_us',

};

Object.assign(createCollectionParams, params);

this.discoveryClient.createCollection(createCollectionParams, (err, data) => {

if (err) {

console.error('Failed to create Discovery collection.');

return reject(err);

} else {

console.log('Created Discovery collection: ', data);

params.collectionId = data.result.collection\_id;

resolve(params);

}

});

});

};

/\*\*

\* Load the Discovery collection if it is not already loaded.

\* The collection should already be created/validated.

\* Currently using lazy loading of docs and only logging problems.

\* @param {Object} params - All the params needed to use Discovery.

\* @return {Promise}

\*/

WatsonDiscoverySetup.prototype.loadDiscoveryCollection = function (params) {

console.log('Get collection to check its status.');

this.discoveryClient.getCollection(params, (err, data) => {

if (err) {

console.log('ERROR during: Get collection to check its status.');

console.error(err);

} else {

console.log('Checking status of Discovery collection:', data);

const docs = params.documents;

const docCount = docs.length;

const result = data.result;

const loadedDocs = result.document\_counts.available + result.document\_counts.processing + result.document\_counts.failed;

if (!Number.isInteger(loadedDocs)) {

throw Error('Unexpected Discovery document\_counts resulted in an unusable not-a-number.');

}

if (loadedDocs < docCount) {

console.log('Loading documents into Discovery collection.');

for (let i = 0; i < docCount; i++) {

const doc = docs[i];

const addDocParams = {

file: fs.createReadStream(doc),

fileContentType: 'application/msword',

};

Object.assign(addDocParams, params);

this.discoveryClient.addDocument(addDocParams, (err, data) => {

// Note: No promise. Just let these all run/log. Revisit this?

if (err) {

console.log('Add document error:');

console.error(err);

} else {

console.log('Added document:');

console.log(data);

}

});

}

} else {

console.log('Collection is already loaded with docs.');

/\* For testing:

discovery.deleteCollection(params, (err, data) => {

console.log('Deleting collection for testing!!!!!!!!!!!!!!!!!!!!!!!!');

if (err) {

console.log(err);

}

});

\*/

}

}

});

// Note: The collection was validated earlier and we are letting the docs lazy load.

// So this one will resolve fast, but might revisit that.

return Promise.resolve(params);

};

/\*\*

\* Validate and setup the Discovery service.

\*/

WatsonDiscoverySetup.prototype.setupDiscovery = function (setupParams, callback) {

this.findDiscoveryEnvironment(setupParams)

.then((params) => this.createDiscoveryEnvironment(params))

.then((environment) => this.findDiscoveryCollection(environment))

.then((params) => this.getDiscoveryConfig(params))

.then((params) => this.createDiscoveryCollection(params))

.then((params) => this.loadDiscoveryCollection(params))

.then((params) => callback(null, params))

.catch(callback);

};

module.exports = WatsonDiscoverySetup**;**

**USE CASES AND APPLICATIONS:**

Chatbots powered by IBM Watson Assistant have a wide range of use cases and applications across various industries. Here are some common use cases and applications for chatbots using IBM Watson Assistant:

1. **Customer Support and Service:**
   * Assist customers by answering frequently asked questions, resolving common issues, and providing information about products or services.
   * Offer support through web chat, mobile apps, or messaging platforms, reducing the load on human agents.
2. **E-commerce and Retail:**
   * Help users find and purchase products by offering product recommendations, assisting with the checkout process, and providing order status updates.
   * Handle inquiries related to product availability, returns, and refunds.
3. **Hospitality and Travel:**
   * Assist travelers with booking accommodations, flights, and activities.
   * Provide travel recommendations, weather updates, and information about local attractions.
4. **Healthcare and Telemedicine:**
   * Offer healthcare information and answer health-related questions.
   * Schedule medical appointments, provide medication reminders, and assist with telemedicine appointments.
5. **Banking and Finance:**
   * Provide account balance inquiries, transaction history, and fund transfers.
   * Assist with basic financial planning and answer banking-related queries.
6. **Human Resources:**
   * Assist employees with HR-related queries, such as leave requests, benefits information, and company policies.
   * Aid in the onboarding process for new hires.
7. **Education and e-Learning:**
   * Offer educational content, answer students' questions, and provide study resources.
   * Support e-learning platforms and help with course enrollment.
8. **Real Estate:**
   * Help users find and inquire about real estate properties.
   * Offer information about property listings, prices, and virtual property tours.
9. **Government and Public Services:**
   * Provide information on government services, procedures, and regulations.
   * Assist citizens with tasks like renewing licenses, reporting issues, or finding local government offices.
10. **Entertainment and Media:**
    * Offer recommendations for movies, music, or TV shows based on user preferences.
    * Provide information about upcoming events, concert tickets, and showtimes.
11. **Lead Generation and Sales:**
    * Engage with website visitors to capture leads and guide potential customers toward products or services.
    * Qualify leads and connect them with sales representatives.
12. **Food and Restaurant Services:**
    * Assist users in ordering food for delivery or pickup.
    * Provide restaurant recommendations, menus, and special offers.
13. **Technical Support:**
    * Guide users through troubleshooting technical issues with products or software.
    * Assist with installation, configuration, and basic IT support.
14. **Travel and Transportation:**
    * Assist travelers with transportation-related queries, such as booking tickets, checking schedules, and finding routes.
15. **Nonprofit and Social Services:**
    * Provide information about nonprofit organizations and their initiatives.
    * Assist individuals in finding community resources and support.
16. **Legal Services:**
    * Offer initial legal advice and guidance on common legal issues.
    * Assist with legal document preparation and contract review.
17. **Language Translation:**
    * Translate text between languages to assist users who speak different languages.
18. **Personal Assistants:**
    * Act as personal virtual assistants, helping with tasks like setting reminders, managing calendars, and providing general information.

These use cases represent only a fraction of the potential applications of chatbots powered by IBM Watson Assistant. The versatility and adaptability of chatbots make them valuable tools for enhancing customer service, improving user experiences, automating routine tasks, and engaging users across various industries.

**EVALUATION AND PERFORMANCE ANALYSIS:**

Evaluating and analyzing the performance of a chatbot using Watson Assistant is crucial to ensure that it meets its objectives and provides a positive user experience. Here are key steps and metrics for evaluation and performance analysis:

**1. Define Objectives and Key Performance Indicators (KPIs):**

* Clearly define the objectives of your chatbot. What do you want it to achieve?
* Determine KPIs that align with your objectives, such as user satisfaction, completion rate, response time, and cost savings.

**2. User Testing and Feedback:**

* Conduct user testing to gather feedback from real users. They can provide insights into the chatbot's strengths and weaknesses.
* Use surveys and feedback forms to collect user opinions and suggestions.

**3. Performance Metrics:**

* Measure and monitor key performance metrics, including:
  + **User Satisfaction**: Gather user ratings and feedback on their chatbot experience.
  + **Completion Rate**: The percentage of user interactions that were successfully completed without errors.
  + **Response Time**: Measure the time it takes for the chatbot to respond to user queries.
  + **Error Rate**: Track the rate of errors or misunderstandings in user interactions.
  + **Fallback Rate**: Measure how often the chatbot falls back to a human agent for assistance.

**4. Natural Language Understanding (NLU) Analysis:**

* Evaluate the chatbot's NLU capabilities by analyzing its ability to correctly identify user intents and entities.
* Monitor and fine-tune the NLU model to improve accuracy.

**5. Conversation Flow Analysis:**

* Review the chatbot's dialog flow and conversation paths to identify bottlenecks and areas for improvement.
* Ensure that the conversation remains natural and contextually relevant.

**6. Error Handling and Recovery:**

* Assess how well the chatbot handles errors, misunderstandings, and unexpected user input.
* Implement improvements in error-handling strategies and recovery mechanisms.

**7. A/B Testing:**

* Conduct A/B tests to compare different versions of the chatbot to determine which performs better in terms of user engagement and completion rates.

**8. Cost Analysis:**

* Analyze the cost savings achieved by using the chatbot, such as reduced customer support inquiries and increased operational efficiency.

**9. Security and Compliance Review:**

* Ensure that the chatbot complies with security and privacy standards, especially when handling sensitive data.
* Conduct security assessments and penetration testing as necessary.

**10. Scalability and Performance Testing:**

* Test the chatbot's performance under heavy loads to ensure it can handle a large number of simultaneous users.
* Monitor resource usage and scale the chatbot infrastructure as needed.

**11. Continuous Improvement:**

* Use the insights gained from the evaluation and analysis to make continuous improvements to the chatbot.
* Regularly update the chatbot's skills, dialog nodes, and responses based on user feedback and evolving requirements.

**12. Documentation and Reporting:**

* Document the results of the evaluation and performance analysis.
* Provide reports and recommendations to stakeholders for decision-making and future improvements.

**13. User Education and Training:**

* Educate users on how to interact effectively with the chatbot by providing clear instructions.
* Develop user guides or FAQs to help users understand the chatbot's capabilities and limitations.

Evaluating and analyzing the performance of a chatbot is an ongoing process. Regularly revisit the metrics and objectives to ensure that the chatbot continues to meet user expectations and business goals. By actively monitoring and improving the chatbot's performance, you can deliver a more valuable and satisfying user experience.

**Implications and Future Outlook**:

During the past years, there has been a significant advancement in chatbot technology. Chatbots are now replacing the role of humans in many domains, and they are able to perform the tasks sufficiently well. However, a chatbot is much more beyond a system that can perform machinelike interactions, and increasing the trustworthiness or the credibility of the chatbots has now attracted a lot of research attention.

**CONCLUSION:-**

In simple terms, chatbots powered by IBM Watson Assistant are smart assistants that help users, save time, and make services more accessible. They're like helpful digital friends that can answer questions, provide support, and get things done faster. With the right setup and ongoing improvements, they make life easier and businesses more efficient.

In conclusion, chatbots powered by IBM Watson Assistant are valuable tools that offer a wide range of applications across various industries. They enable businesses and organizations to provide efficient and personalized interactions with users, automate tasks, and enhance customer service.