Tampere University

Md Rayhan Al Islam

ADDING CONTEXT AWARENESS TO CHATBOTS WITH MICROSERVICES: CASE FOREST COMPANION

Master of Science (M.Sc Tech) Thesis
Faculty of Information Technology and Communication Sciences
May 2023

ABSTRACT

Md Rayhan Al Islam: Adding context awareness to chatbots with microservices: Case Forest Companion
Master of Science (M.Sc Tech) Thesis
Tampere University
Software, Web & Cloud
May 2023

Chatbots are now available on websites and apps where people are trying to sell some services or products. It makes the product owners' life easier by serving the customers automatically with some questions and answers. The concept of context awareness in chatbots is common in the development sectors for increasing user engagement and better response by the bot. Obtaining a complete context awareness is a challenge and needs different techniques such as natural language processing, intent recognition, etc. Also, when the chatbot has different modules like fetching some information from the backend and replying to the user or extracting intent from the conversation, the complete setup of the chatbot becomes complex and hard to maintain. Therefore, a good integration structure is which keeps it manageable from time to time.

Developers face difficulties because the chatbots currently available do not exactly provide the whole code to the consumer, rather they sell it as a solution. Certain chatbot frameworks provide an open-source license but still, those needs some planning and deployment procedure to make it a complete solution.

In this thesis, we have proposed an architecture model, based on microservices that shows how context awareness can be achieved using a Belief-desire-intention (BDI) model. Furthermore, it is also shown how an extra service can be added to the chatbot. For explaining this, a case study has been done where a location-based game database service is used to integrate with the BDI model and make the chatbot work as a whole. The main goal of this thesis is to explain how we can produce certain software architecture and make the integration process testable, scalable, and less complicated.

Keywords: chatbot, RASA, microservices, location based game, Belief-Desire-Intention based Agent

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

PREFACE

This thesis idea came when we started developing the location-based game during the summertime of 2022. I would like to thank my colleague Ville, my supervisors Timo and Samuli for their idea and contribution for making this thesis complete. When I started working on this thesis work, I did not have much idea about the chatbots. I had to study the complete system for a long time to get the necessary idea of it.

Finally, I want to thank my family and friends for supporting me during my hard times. I would also like to thank you, my reader: I hope you enjoy your reading.

Tampere, 31st May 2023

Md Rayhan Al Islam

CONTENTS

1. Introduction 1
1.1 Motivation 1
1.2 Research Question 3
1.3 Research Approach 4
2. Background 5
2.1 Context-Aware Chatbots 6
2.2 Microservice for context-aware bot 7
2.3 Popular Chatbot Solutions 8
2.3.1 IBM Watson Assistant 8
2.3.2 Hubot 9
2.3.3 Amazon Lex 10
2.3.4 RASA 11
2.4 Belief-Desire-Intention (BDI) Agent Model 12
2.5 Location Based Games (LBG) 14
2.6 Docker 14
3. Requirements and Overview of the Design 17
4. Implementation of a forest companion chatbot 20
4.1 RASA 20
4.2 Js-son BDI Agent 23
4.3 Data Source 24
4.4 Docker Orchestration 27
4.5 Slack Integration 28
4.6 Deployment 31
5. Discussion 33
5.1 Assessment of the research question 33
5.2 Contribution 34
6. Conclusion 36
References 37
Appendix A: Codes 41
A.1 RASA 41
A.2 BDI node server 49

LIST OF SYMBOLS AND ABBREVIATIONS

LBG	Location Based Game
BDI	Belief Desire Intention
GDPR	General Data Protection Regulation
NLU	Natural Language Understanding
BaaS	Bot as a Service
SaaS	Software as a Service
HCI	Human Computer Interaction
ASR	Automated Speech Recognition
AWS	Amazon Web Services

1. INTRODUCTION

1.1 Motivation

The first chatbot ever created, ELIZA [1], has a major role in the development of artificial intelligence and human-computer interaction (HCI). ELIZA was developed by MIT computer scientist Joseph Weizenbaum in the middle of the 1960s with the intention of simulating a Rogerian psychotherapist and involving users in text-based talks. By assessing user input and producing responses based on established patterns and templates, ELIZA used a rule-based approach. The chatbot's ability to understand words and phrases enabled it to give thoughtful, empathic responses that mirrored those of a human therapist. ELIZA could paraphrase comments, ask insightful questions, and carry on simple conversations to give the impression that she understood. ELIZA was special because it could appear intelligent while not actually comprehending the meaning of the speech. It depended on smart programming strategies to reroute user input and offer generalized replies that suggested empathy and participation. Even though ELIZA lacked actual comprehension or emotional capacities, users frequently felt compelled to express their thoughts and emotions to the chatbot.

In today's digital scene, chatbots are a technology that is becoming more common and prominent. These software packages replicate human interaction while offering prewritten responses to user questions or requests. Chatbots have developed throughout time from basic rule-based systems to sophisticated artificial intelligence (AI) models that can analyze and comprehend natural language.

We can see from Figure 1.1 how chatbot usage has increased from 2016 [2]. Also, figure 1.2 shows the diverse usage of chatbots. The chatbots from the '90s has limited application field but it has significantly changed from time to time.

The capacity of chatbots to improve customer care interactions is one of the key factors contributing to their rising popularity. AI and machine learning has advanced to the point where chatbots can now evaluate and comprehend user input to deliver precise and pertinent responses [3]. Due to this, they have become widely used in a variety of sectors, including e-commerce, banking, healthcare, and customer support.

Additionally, chatbots are available 24/7, which is quite convenient for users. Chatbots can

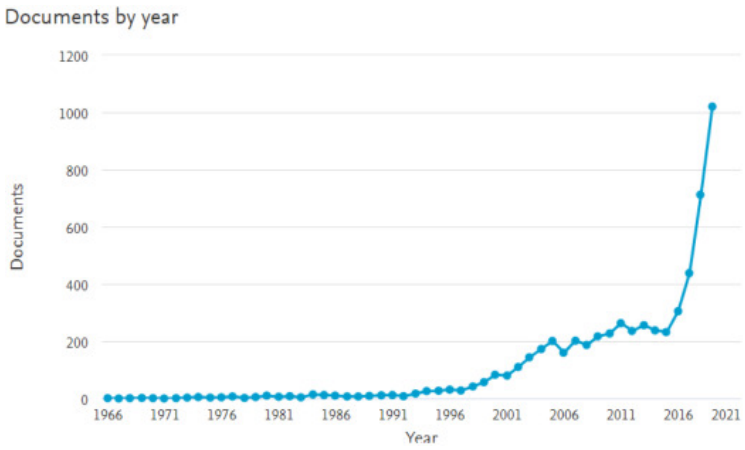

Figure 1.1. *Usage of Chatbot*

Chatbot Categories		
	Knowledge domain	Generic
		Open Domain
		Closed Domain
	Service provided	Interpersonal
		Intrapersonal
		Inter-agent
	Goals	Informative
		Chat based/Conversational
		Task based
	Response Generation Method	Rule based
		Retrieval based
		Generative
	Human-aid	Human-mediated
		Autonomous
	Permissions	Open-source
		Commercial
	Communication channel	Text
		Voice
		Image

Figure 1.2. *Chatbot Categories*

work around the clock, providing ongoing support and help in contrast to human agents who have set office hours [4]. Users may receive prompt responses to their inquiries without experiencing any delays thanks to its availability, which improves customer happiness and retention. The scalability of chatbots is another important benefit. It gets harder to adequately handle the volume of questions as organizations expand and encounter more client contacts. Because chatbots can manage numerous conversations at once, all users will receive prompt responses. Due to its scalability, organizations no longer need to hire and train additional customer care agents, saving money and streamlining processes.

Chatbots can assist in identifying common problems and opportunities for development by tracking patterns and trends in client questions [5]. With the help of these insights, it will be possible to improve the total customer experience by optimizing goods, services, and business procedures. Recent developments in AI and natural language processing have made it possible for chatbots to perform more complex tasks. Modern chatbots

are capable of comprehending context, having meaningful discussions, and even displaying personality traits. Chatbots can continuously learn from user interactions by utilizing machine learning techniques, which will lead to responses getting better over time and increasing in accuracy.

Nevertheless, despite all of their benefits, chatbots have certain drawbacks. Complex or ambiguous user queries may be difficult for chatbots to interpret [6], which might result in erroneous or irrelevant responses. They also lack empathy and human emotions, which can be important in customer service scenarios. Businesses must balance chatbot automation with human intervention while acknowledging these constraints to provide seamless client experiences. The potential for chatbots in the future is enormous. Chatbots will grow even smarter and more adept as AI and natural language processing technology progress [7]. We may anticipate chatbots integrating with a variety of platforms and gadgets, such as voice assistants, smartphone apps, and smart home gadgets, to increase their usability and accessibility.

The customer base for chatbots can vary based on the purpose of the bot. In general, there are a large group of people who will find chatbots supportive for finding information quickly. For statistics, it is found that 54% of consumers find the chatbots really helpful for resolving queries [8]. Another important feature of a chatbot can be multilingual service. The chatbots now can automatically process other languages and reply accordingly [9]. Moreover, if we think about faqs or serving very basic questions, a chatbot can serve the purpose of being an actual representative in this case. For making the bots smarter, context awareness can play a vital role. The bot needs to understand the user's intention and what he/she is really trying to achieve. There are some important contexts mentioned in this article which are location, identity, activity, and time [10]. Also, in another article, it is seen that contexts can be categorized, for example- computing context (processors, nearby available devices to connect, etc), user context (location, user profile, etc), and physical context (lighting, temperature, etc.) [11]. Therefore, it shows that contexts can vary a lot and that means the context requirement for a bot can change also.

1.2 Research Question

In this thesis, our goal is to enhance bots' context-understanding ability by following an architecture model. Also, besides context awareness, we want to make the chatbot capable of integrating independent software services with it. So, the research questions for this thesis will be-

1. How can context awareness be achieved by microservice-based architecture?

2. How external systems can be integrated with chatbots?

1.3 Research Approach

For this thesis, the experimental research method has been selected. We have proposed an architecture model and we have implemented that architecture to validate the feasibility. The experiment shows why this architecture will be better in the long run for application scalability and robustness. In the first phase of the research, we reviewed previous architecture models and chatbot implementation techniques. Section 3, shows our proposed implementation model and in section 4 we have discussed how this model is more versatile compared to the currently available ones. Also, we have discussed how we have verified the architecture with our case study of a location-based game. Last but not least, in the conclusion we have explained the challenges faced and what can be done in the future.

2. BACKGROUND

Most websites and different mobile applications are now integrating a bot to serve the customers' basic needs. Therefore, a good structure of the chatbot implementation is necessary. Ahmad et al.[12] showed in their research based on StackOverflow questions where they have shown what challenges the developers are facing while implementing chatbots. They mentioned that 61.8% of chatbot developers ask questions about implementations. They have considered the questions in three different parts- how, why and what. This percentage was based on 'how' types of questions. Another research was done regarding the programming challenges of chatbots. Rahman et al. explained in their research [13] how the multi-diversified modules of chatbot can be difficult to handle. Also, they have mentioned the challenges of training extensive datasets to make the bot understand different contexts of the conversation. Therefore, we have explored the different implementation possibilities of chatbots.

The most popular architecture to integrate a bot currently is the Bot as a service model (BaaS) which means the bot will be provided by another company and the customer will be charged in a pay-as-you-go manner. One of the good examples of BaaS is the Azure bot service. Here different logic and functions can be added from the Azure cloud panel. The main concept of this bot will be putting general questions based on the business and serving them to the customer but prompting and answering. Though this model is highly scalable and maintained automatically by azure cloud, the challenge comes, if the developers want to add artificial intelligence or questions which are outside of the domain. Also, all the programming languages are still not supported there. Moreover, it requires monitoring else it might get expensive for the users.

In addition to BaaS, there are also monolithic applications that use NLP libraries to process conversations and reply to them. These monolithic applications are good if the user base is small and if the deployment is simple. Moreover, under average load, monolithic and microservice-based applications behave almost similarly [14]. In this thesis, it is explained the architecture keeping in mind that the application will have a large amount of traffic. Also, the bot architecture I am showing here has multiple independent modules which can be tested and modified separately without any dependency measurements.

2.1 Context-Aware Chatbots

Chatbots' context awareness can be achieved in multiple ways. A proposed architecture including a context-aware manager has been proposed recently [15]. This manager works based on the context dimension tree. The context manager extracts data about who the user is and what type of location he/she might be interested in, and based on that the chatbot will suggest tourist destinations. The architecture of the bot is a mixed architecture model style. They have put some components together in a single module and some of them are independent. It is felt that, in the long run, this architecture model can be challenging to maintain the chatbot.

Another article [16] proposed a context-aware self-attentive NLU model which uses the user's previous intentions, dialogue acts, utterances, etc to extract context from the conversation. They focused only on the model, not on the integration level.

Furthermore, this article [17] shows how contexts have been extracted from an image. The bot is able to define the context and response by converting text to speech. Also, it has the ability to keep the conversation track which means, it can remember old conversation context which makes it even smarter.

What is the current challenge identified?

Context-aware chatbots are not without their difficulties. Context-awareness demands advanced algorithms and a large amount of training data. Techniques including intent identification, named entity recognition, sentiment analysis, and dialogue management are used. Furthermore, while dealing with personal information in the context of interactions, guaranteeing user privacy and data protection is critical. Despite that, context-aware chatbots have the potential to transform our interactions with technology. Their ability to comprehend and adapt to the context of a conversation allows for more natural and productive interactions, making them useful tools in customer service, support systems, virtual assistants, and a variety of other applications. As conversational AI research and development improve, we may expect context-aware chatbots to become even smarter, enabling increasingly smooth and human-like discussions. For enabling this, flexibility in implementation is necessary. The chatbot should be able to consume multiple context analysis sources and still keep its robustness and maintainability at a certain level.

The core model of our chatbot architecture is based on microservices. Therefore, literature has been reviewed to gather information regarding the microservices implementations and their advantages for the chatbot.

2.2 Microservice for context-aware bot

The way complicated systems are created, delivered, and maintained has been revolutionized by the rise of microservices as a common architectural style in software development. Large monolithic applications are divided into more manageable, loosely connected services using this method of software design. These services can be independently created, deployed, and scaled. Organizations can improve the agility, scalability, and resilience of their software systems by implementing a microservices architecture. The concept of breaking down an application into a number of independent services that may communicate with one another through well-defined APIs is at the heart of microservices. User identification, payment processing, and inventory management, among other business capabilities or functions, are handled by a different microservice for every microservice.

Advantages of microservices over monolithic solutions:

One of the key advantages of microservices is their ability to promote scalability and flexibility [18]. Since each service operates independently, it can be developed, tested, and deployed separately. This allows organizations to scale individual services based on demand without having to scale the entire application. For example, if a particular service experiences a surge in traffic, it can be horizontally scaled by adding more instances of that service, ensuring that the overall system remains responsive and available. Microservices also enable organizations to adopt a more agile approach to software development. With smaller, self-contained services, teams can work on different services simultaneously, using different technologies and programming languages as appropriate. This decentralized approach fosters innovation, accelerates development cycles, and allows teams to deliver new features and improvements quickly. Additionally, the modular nature of microservices simplifies maintenance and troubleshooting, as issues can be isolated to specific services rather than impacting the entire application [19, 20]. The robustness of microservices is another important advantage. One service failing or having problems does not cause the entire application to crash. Instead, other services can carry on operating independently, guaranteeing the availability of essential functionality. Well-defined APIs and service boundaries enable companies to achieve fault isolation and reduce the impact of failures on the system as a whole. Additionally, the effective deployment, scaling, and monitoring of microservices are made possible by the use of containerization and orchestration tools like Docker and Kubernetes.

Adopting a microservices architecture, however, presents a unique set of difficulties. To facilitate inter-service communication, organizations must spend on developing reliable communication channels like message queues and API gateways. It might be challenging to manage distributed transactions across several systems and ensure data consistency. That is why currently the microservices implementation regarding chatbots are not very

common yet. Organizations mostly try to avoid these maintenance difficulties. Still, in the long run, if we think about it on a large scale, microservices can be a better solution indeed.

There has been recent work done with chatbots and microservices. The article [21] shows how the chatbot was empowered to serve medical-related conversations with customers. They have divided the services into different modules and showed how the bot is prompted for information and stores those data in the database. They have used NLU libraries by Java to parse the conversation and get data out of it. So, the main challenge in this paper found was the data processing and training model. Throughout the architecture, they did not mention the core development of the chatbot and how it is developed. So, if there is no framework or services behind it, scaling and maintenance can be expensive and challenging to do. If there is a need to add more information regarding medical care and train the bot with new data, the procedure might be cumbersome. Therefore, for chatbot implementation, a framework will be always better for making it futureproof. Also, the extension of data source integration can be easier to implement. In my architecture model, I have tried to resolve these kinds of challenges.

In order to implement chatbots, there are several options available now. For example, IBM Watson Assistant, Hubot, Amazon Lex, RASA, etc. All of these applications have their own advantages [22]. Here the mentioned solutions are reviewed to check the feasibility of their integration with our architecture. This information has been taken from the official documentation of these services [23, 24, 25, 26].

2.3 Popular Chatbot Solutions

2.3.1 IBM Watson Assistant

IBM Watson Assistant is a strong artificial intelligence (AI) platform. It is intended to give organizations a sophisticated conversational AI solution capable of understanding and responding to user queries in a natural and intuitive manner. Watson Assistant enables enterprises to create chatbots, virtual assistants, and voice-enabled applications that can engage with users and deliver personalized experiences by leveraging natural language processing (NLP) and machine learning technology.

One of IBM Watson Assistant's important strengths is its capacity to perceive and interpret the meaning behind human inputs. It is capable of comprehending and analyzing text and audio input, identifying entities, intents, and context, and generating appropriate responses according on the user's requests. Watson Assistant has a number of tools and capabilities that make it easier to design and deploy conversational AI applications. Users can construct conversational flows, establish intents and entities, and create dialogues using the platform's graphical interface. It also integrates with a variety of messag-

ing services, including Slack, Facebook Messenger, and web chat interfaces, allowing for smooth deployment across numerous channels.

Watson Assistant can be coupled with other IBM services such as Watson Discovery, Watson Knowledge Studio, and Watson Natural Language Understanding to expand its capabilities. These services allow the assistant to access structured and unstructured data sources, do advanced language analysis, and extract insights from complex data, allowing for more sophisticated interactions and answers. Furthermore, Watson Assistant allows for customization and customization, allowing businesses to adjust the assistant's replies and behaviors to their own needs. It gives businesses options for training and fine-tuning the AI model, allowing them to increase accuracy and maximize the assistant's performance over time. Watson Assistant's versatility makes it suited for a wide number of industries and use cases, including healthcare, finance, e-commerce, and others.

Finally, IBM Watson Assistant is a comprehensive and powerful artificial intelligence platform that enables enterprises to construct intelligent conversational agents. Watson Assistant helps enterprises to offer excellent user experiences and realize the promise of conversational AI in their operations with its superior NLP capabilities, integration choices, customization tools, and support for many channels.

2.3.2 Hubot

GitHub created Hubot, an open-source chatbot framework. It was originally designed to automate processes and aid developers in their everyday workflows, but it has since grown in popularity and is now used for purposes other than software development. Hubot is a flexible and extensible platform for developing chatbots that can interface with numerous chat platforms and perform a variety of functions.

CoffeeScript, a programming language that is compiled into JavaScript, is used to create Hubot. It is based on the Node.js runtime environment and uses JavaScript to create dynamic and responsive chatbots. It employs the adapter pattern, allowing it to communicate with several chat services such as Slack, HipChat, Microsoft Teams, and others. This flexibility enables developers to deploy their Hubot instances to their preferred chat service, making it accessible to users within their existing communication channels.

One of the key features of Hubot is its extensive library of scripts, known as Hubot Scripts, which provide a wide range of capabilities out of the box. These scripts cover various categories, including deployment automation, monitoring, task management, and entertainment. Developers can leverage these scripts or create their own to extend Hubot's functionality and tailor it to their specific needs. Hubot scripts are written in CoffeeScript, making it easy for developers familiar with JavaScript to contribute and create new scripts. Hubot's extensibility is a major strength of the framework. Developers can create custom

scripts and plugins to add new features and integrate them with external services and APIs. This allows Hubot to perform tasks such as querying databases, retrieving information from web services, interacting with version control systems, and more. With its extensibility, Hubot can be customized to fit the unique requirements of different teams and organizations. Hubot's capacity to listen for and respond to natural language commands is another noteworthy feature. It makes use of regular expressions and pattern matching to comprehend user input and initiate suitable actions. Natural language processing makes Hubot interactions more intuitive and user-friendly.

Overall, Hubot is a robust chatbot framework that streamlines the process of developing and deploying chatbots. Hubot enables developers to create chatbots that automate jobs, optimize workflows, and improve collaboration with its extensible architecture, prebuilt scripts, and compatibility with numerous chat platforms. It's vibrant community and ecosystem add to its adaptability and ensure its relevance in the ever-changing world of chatbot development.

2.3.3 Amazon Lex

Amazon Lex is an Amazon Web Services (AWS) tool that allows the building of conversational interfaces, also known as chatbots or virtual assistants. It understands and responds to user inputs in a conversational manner by leveraging advanced natural language understanding (NLU) and automatic speech recognition (ASR) capabilities. Amazon Lex is built to be highly scalable, cost-effective, and simple to incorporate into a wide range of applications and platforms.

Amazon Lex's excellent natural language understanding engine is one of its standout characteristics. It analyzes and comprehends the meaning of user inputs using machine learning methods, allowing Lex-powered chatbots to accurately read and extract intents, entities, and context from dialogues. Developers can utilize this to construct chatbots that interpret user inquiries in a human-like manner, making interactions more intuitive and natural. Amazon Lex can engage via text as well as voice. It includes automated speech recognition (ASR) capability, allowing developers to create voice-enabled chatbots that can interpret and process spoken words. This functionality is very valuable for voice assistants, call center automation, and speech-controlled products.

Amazon Lex's connectivity with other AWS services is another noteworthy aspect. It interacts smoothly with AWS Lambda, allowing developers to build custom business logic and run backend actions in response to user inputs. Lex also supports Amazon DynamoDB, Amazon S3, and other AWS services. Amazon Lex has a graphical user interface for creating and constructing chatbot conversation processes. Developers can design intentions, which represent the goals or actions that the user wishes to carry out, and then develop related utterances that reflect the many ways in which users communicate those

intents. Lex allows developers to create dynamic and engaging chatbot experiences by quickly defining and managing conversational exchanges, specifying prompts, and handling complex dialogues. Furthermore, Amazon Lex allows the multi-platform deployment, allowing Lex-created chatbots to be delivered across several channels and devices. It offers a software development kit (SDK) for interaction with messaging services such as Slack, Facebook Messenger, and Twilio, making it simple to reach people through their chosen channels.

In summary, Amazon Lex is a powerful and versatile service that enables the development of chatbots and virtual assistants. With its advanced natural language understanding capabilities, integration with AWS services, and support for various deployment channels, Lex empowers developers to create intelligent and conversational interfaces that can enhance customer experiences, automate tasks, and improve overall efficiency.

2.3.4 RASA

Being an open-source framework, RASA provides different components to build to chatbot completely. Here we have mentioned which features of RASA have been used in the architecture model-

Natural language understanding: RASA has this feature of training itself with the basic dataset. This dataset helps it to understand the conversation and extract the intent from it. Now the question can be, what is an intent? Intent is the key concept of the conversation what the user is trying to find.

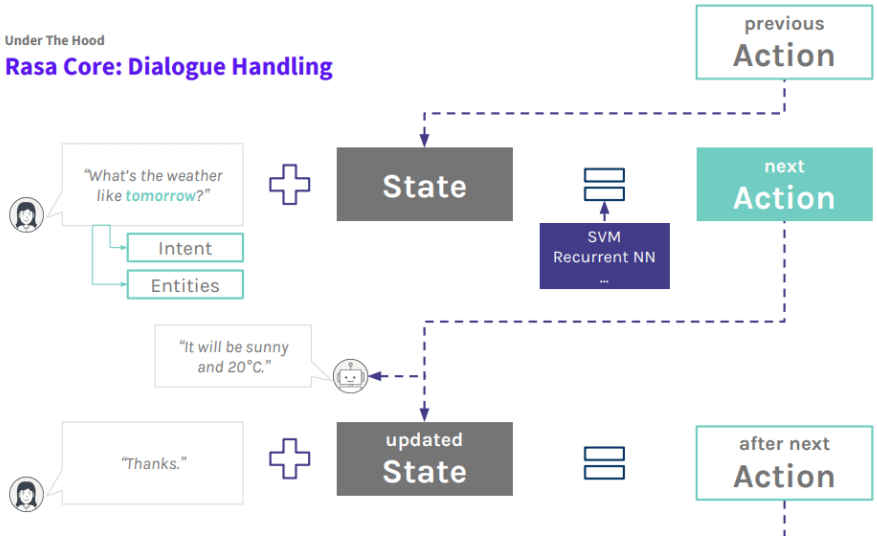

Figure 2.1. *RASA dialogue Handling [27]*

For example, What is the current weather in Tampere? Here as we can see the sentence is starting with 'what' so, the general intent of this sentence will be getting information from the bot.

Action Server: Calling other APIs, searching databases, or carrying out sophisticated computations are operations that cannot be handled entirely by the NLU and Core components. The RASA Action Server is useful in this situation. We can specify custom actions in the chatbot that can carry out these functions and give the user dynamic responses using the Action Server. Alongside the RASA Core, a separate process called the RASA Action Server operates. It uses a REST API to communicate with the Core component. The Action Server receives a request from the Core when it anticipates a custom action, executes it, and returns the outcome. To create a custom action, a Python class is required that inherits from the rasa_sdk.Action class. This class contains a method called run() which is responsible for executing the action. Within the run() method, we can implement desired logic, such as making API calls, performing calculations, or interacting with databases. We can also use the extracted entities and slots from the conversation to customize the behavior of the action.

The RASA Action Server provides a flexible and extensible framework for adding custom actions to the chatbot. It allows us to integrate external systems, perform complex computations, and create dynamic responses based on the user's input. With the Action Server, we can enhance the capabilities of the RASA chatbot and create more interactive and powerful conversational experiences.

Third-party integration: The framework also has the option to integrate this in several other third-party platforms for example- Facebook messenger, telegram, slack, discord, and even in custom websites. The particular integration processes will be determined by the third-party product or service. Rasa provides detailed documentation and examples to navigate the process of integrating with other platforms. Furthermore, the Rasa community is active and helpful, so seeking assistance or direction from the community forums or chat groups if any special integration issues is encountered.

This feature plays a vital role because most chatbots are added to a specific application for supporting purposes. Therefore we have created a channel in Slack for the location-based game(LBG) players and the bot can run conversations with them in that channel. The integration process is simple and easy to make. It needs the API credentials to have access to API calls and then minor configuration is needed on the vendor end. That way it can reply to conversations and also the RASA bot can start a conversation by itself in the channel.

2.4 Belief-Desire-Intention (BDI) Agent Model

Artificial intelligence and agent-based systems employ the Belief, Desire, and Intention (BDI) agent model as a framework to represent and analyze the thoughts and actions of autonomous agents. The BDI was designed by Michael Bratman for explaining future-directed intentions [28]. The software model was done based on this design. It is a

cognitive architecture that offers a way to represent an agent's beliefs, goals, and intentions, which are important elements of thinking and making decisions in a manner that is similar to that of a person.

The three elements of the BDI model are broken down as follows:

Belief: Beliefs are an agent's knowledge or understanding of the outside world. The agent's perspective of its surroundings, internal states, and current situation are all included. The most common way that beliefs are expressed is as a knowledge base or a collection of propositions that the agent believes to be true.

Desire: The goals, preferences, or motivations of the agent are reflected in their desires. They stand in for the goals or objectives the agent has. Desire is known as the best state of the agent which the agent always wants to reach. The desires work as the goal of the agent and to meet that goal, the agent uses intentions and plans its way towards the desires.

Intention: The agent's plans or future actions are captured by their intentions. They stand for the agent's dedication to reaching particular objectives or aspirations. An intention is a decision to carry out a specific action or group of activities in accordance with one's views and desires. As they guide the agent's behavior and decision-making, intentions can be considered as the link between desires and actions.

The BDI agent architecture model is possible to integrate with any type of software to make an intelligent agent. Integrating this model with a chatbot for creating context awareness is a thing that we found will be interesting to see. There are researches on storytelling software to add BDI agents [29]. The main idea of this was, how the implemented characters behave when the intelligent agent is included in the system. Also, an article has been found regarding the BDI agent model to design a personal assistant [30]. The article shows how multiple backend applications have their own agents and each agent has a special task for example calendar management. So, here a coordinator agent is integrated with the BDI model to supervise all the existing agents. Furthermore, another interesting research has been done about BDI agents to see how the agent reacts in the case of mental health care [31]. This agent keeps showing compassion towards the user until the user feels okay.

Another BDI agent implementation was explained in this research [32]. They have implemented their own BDI model without any external libraries. They have used the RASA framework to manage the bot, the database is PostgreSQL. For containerizing the app, they used docker. The overall architecture looks similar to ours. They have mentioned in the article that, the challenge they faced was starting the conversation by the bot itself to make it more engaging.

Analyzing the recent works, it is seen that there has not been a large number of appli-

cations or libraries available to support the BDI model. It is found that there has been an implementation called js-son library which is a javascript-based implementation [33]. Another implementation was done in Python called spade-bdi.

After considering the documentation, It is felt that js-son will be easy to implement. Also, the js-son developer wrote an article about the whole implementation so it has been easy to follow [33]. In this paper, they have given an example of an agent who has a nature of claustrophobia. Therefore, when the agent believes the door of the room is closed, it becomes active to reach its desire which is opening the door. So, here the desire is an 'opened door' and the intention is making a plan to 'open the door'. The belief is continuously changing based on the current state of the door.

The BDI model describes reasoning and decision-making as a process of choosing the best course of action to take in order to attain the agent's goals while taking the agent's beliefs and intents into account. The agent continuously modifies its desires, updates its beliefs in light of new information, and creates intentions as a result. The approach supports flexible, goal-directed behavior that can change with the environment.

2.5 Location Based Games (LBG)

Location-based games are digital versions of the game where real-world gps location is used with gamification. To be precise, the user needs to move to the real world to find out points of interest and collect points. There is always gamification behind this which motivates the user to move more. The most popular location-based game is Pokemon Go [34]. The game we have worked on to collect data for this bot has a similar motive. We planned that game to increase user movement in the forest area of Tampere. Data collection from LBGs is quite a common concept [35, 36] and this collection process is always done by keeping in mind the GDPR rules and regulations.

Also, it is seen from different research that, LBGs can change the user movement pattern in the outer world [37]. This means people like to walk outside while playing this game. Therefore, apart from data collection, we were also motivating people to walk nearby the forest area. In the case of our user's data security, we are not collecting any kind of personal credentials of the user. There are only unique users and the geolocation is recorded within 2-3mins.

2.6 Docker

Docker is an open-source technology that enables developers to automate application deployment and administration within software containers. It provides a lightweight, portable, and scalable environment for application execution, making it easier to create, ship, and deploy software across many computer systems. Docker's main technology is container-

ization, which allows for the development and isolation of software packages known as containers. Each container contains all of the dependencies, libraries, and customizations necessary to execute a given application. These containers are self-contained and can be readily transferred across platforms without causing compatibility concerns.

Docker Compose is yet another powerful Docker ecosystem product that helps developers to create and manage multi-container applications. YAML file can be used with Docker Compose to provide the setup of the application's services, including their dependencies, networks, volumes, and environment variables. This makes it easier to execute complicated applications that require numerous linked containers.

Orchestration of docker containers:

The control and synchronization of large Docker containers in a distributed environment is referred to as "Docker orchestration." A cluster of machines must be used to launch, scale, and manage containerized applications. The management of complicated containerized applications is made simpler by the tools and functionality offered by Docker orchestration frameworks for automating the deployment, scaling, networking, and monitoring of containers. This helps to create network bridges and also put network rules to make communication between containers. Also, inside an orchestration, it is possible to make volumes that can be shared between different containers.

The benefits of orchestration-

Scalability and load balancing: By adding or removing instances (replicas) in response to demand, orchestration frameworks enable us to scale containerized applications horizontally. Additionally, they offer integrated load-balancing systems to divide traffic among the containers. Networking and service discovery are required for communication between containers in an orchestration cluster. Orchestration systems frequently offer techniques for service discovery to assign containers with dynamic IP addresses or DNS names, facilitating communication. In order to support container networking across several hosts, they also provide network overlay capabilities.

Container scheduling: Taking into account resource availability, load balancing, and limits provided in the service specification, orchestration systems handle the scheduling and placement of containers on host computers. They guarantee that containers are dispersed around the cluster in an ideal way.

Monitoring: Continual monitoring of the health and condition of containers is done through orchestration frameworks. To keep the application in the correct state, the framework automatically restarts or replaces any failing or unresponsive containers.

Maintaining updates: Rolling updates and rollbacks are supported by orchestration platforms, enabling to upgrade of containerized applications without experiencing any down-

time. They allow us to maintain application availability while gradually deploying new versions. Rollbacks to earlier versions are possible in case of problems.

Resource Management: Orchestration frameworks offer ways to effectively manage and distribute resources. They can impose resource restrictions, order workloads, and maximize resource usage throughout the cluster.

The orchestration can be done with different technologies now. Currently for large-scale applications the most popular technology is Kubernetes. Also, there are docker compose, docker swarm, open shift, and other solutions available. Each of them has its own advantages and disadvantages [38]. A brief idea of them will be explained here-

Kubernetes: Kubernetes is designed by Google and it is now vastly used for maintaining containers. It contains a variety of deployment patterns and features to meet various application designs and scalability needs, and it offers an API-driven interface for communicating with the cluster. Although Kubernetes is a strong platform, it is also complicated and has a long learning curve.

Swarm and other Docker-native tools: Although considered to be less powerful than Kubernetes, the Docker ecosystem provides a number of orchestration solutions that are simple to use and easily accessible to Docker users. These include Docker Compose, which automatically launches applications made up of several containers, and Docker Swarm, a compact container orchestrator.

Other Cloud container platforms : Simpler than full-scale orchestrators like Kubernetes, such as Amazon Elastic Container Service (EKS), Google Cloud Run, or Azure Container Instances (ACI), offer rudimentary orchestration capabilities. Still, these are less popular compared to Kubernetes and docker native solutions.

3. REQUIREMENTS AND OVERVIEW OF THE DESIGN

From the previous literature, a similar work was found which was for making a chatbot for patient support [21]. They have mentioned about 3 major requirements for a good chatbot architecture-

1. Scalability that enables easy expansion of the bot

2. Standard data paradigms that encourage data sharing and reuse

3. Basic conversational modeling that helps the user communicate with the chatbot

In this architecture, we will follow these requirements. In addition to these, the architecture will enhance the bot in such a way that it will become a context-aware forest companion bot. The main contexts this bot is following are the weather and the player's movement while playing the location-based game. In this architecture, the RASA framework has been used as the main engine.

Why we have selected RASA?

The IBM Watson and Amazon Lex have several ties with their own other services and the chatbot integration documentation shows that they perform better if those services are taken. The learning curve of these platforms is not that smooth and might take some time to understand the setup. In terms of Hubot, it is found very similar to RASA and has almost the same functionalities but based on online resources and support by other developer communities, it is seen that RASA has more support and is more frequently used than Hubot. Therefore, measuring these factors, we decided to keep RASA in our architecture.

For making the bot aware of the weather, we have used weather API from AccuWeather. Also, the PostgreSQL server is providing the player and POI location to let the bot know about the player movement context.

Microservice style design

For reaching the mentioned requirements, this architecture is implemented in a microservice-style design. The requirements of this design are based on the principles of microservices which will make the solution better maintainable, scalable, and expandable [39]. The requirements it is planned to fulfill are-

1. Each service has separate responsibilities and it does not overlap with others.

2. The services should be loosely coupled so that they can be developed independently.

3. Fault isolation and scalability of the services need to be kept in mind.

4. Possibility to automate the CI/CD pipeline

5. The orchestration of the microservices is necessary to increase maintainability and networking.

6. Each service should be reusable which means outside of this environment, the services can act as they should.

7. The architecture should follow robust security practices. Communication between the services should be highly secured and not exposed to the outer world.

8. Continuous improvement of the services needs to be possible based on future changes and requirements.

For fulfilling these requirements, the whole application is divided into different independent parts and used the idea of containerizing each service. We have selected a case study to validate the architecture flow, so the diagram is created based on the case study of a location-based game. A top-level view of the implementation will be similar to Figure 3.1.

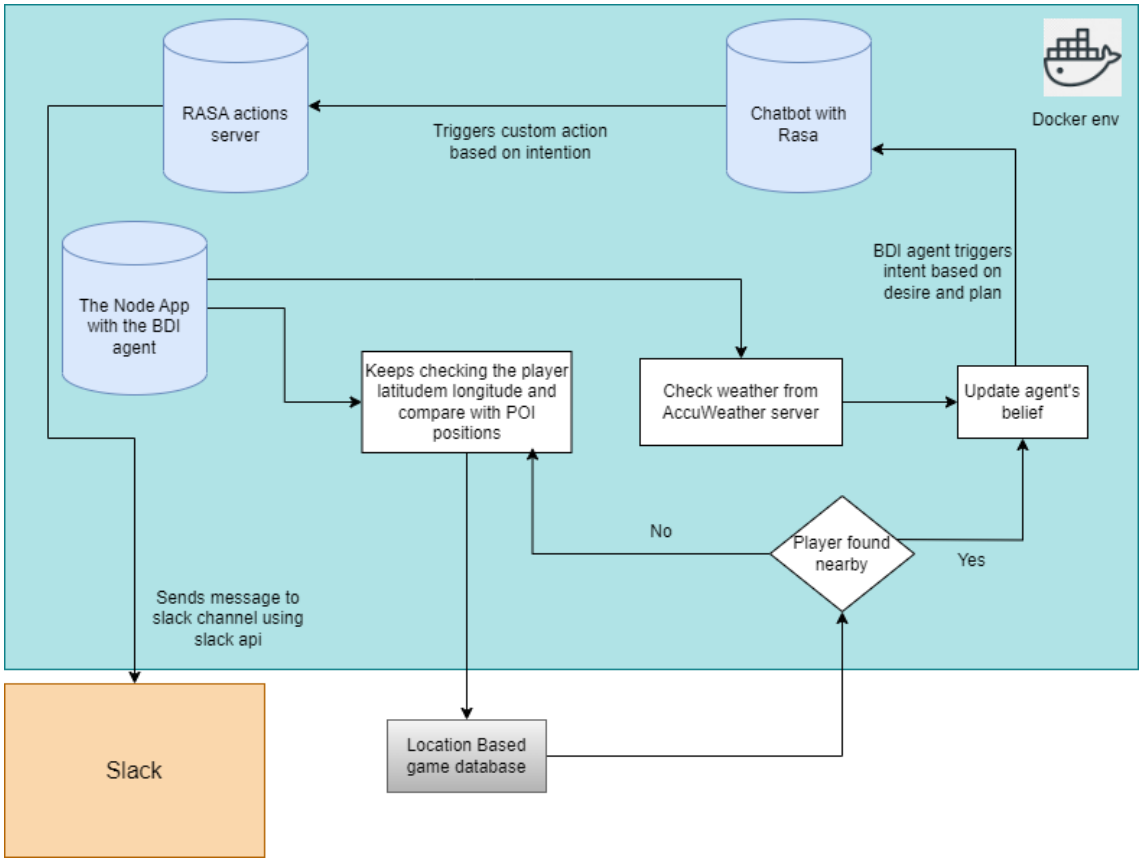

Figure 3.1. *Full architecture diagram*

The square large box is the docker environment. Slack and the LBG's database are outside of this environment. The BDI with the node server is continuously checking the player's position and weather. Based on the response, the belief of the agent is updated and the agent triggers RASA intent and the custom action server sends a message to the slack channel of the players.

The architecture plan implementation is explained in detail in the next chapter.

4. IMPLEMENTATION OF A FOREST COMPANION CHATBOT

4.1 RASA

In the implementation, we have used the RASA framework to design the chatbot's basic functionalities. RASA trains the bot based on three different files which are-

i) nlu.yml: The nlu file consumes different examples and tries to extract the entities from the conversation. Based on those entities, we can trigger different intents to complete the action cycle and dispatch the response from the bot. For example-

```
- intent: greet
  examples: |
    - hey
    - hello
    - hi
    - hello there
- intent: weather_update
  examples: |
    - what is the weather in [tampere](place)?
    - what is the weather in [helsinki](place)?
    - what is the weather like in [turku](place)?
    - what is the weather like in [espoo](place)?

- lookup: place
  examples: |
    - Akaa
    - Alajärvi
    - Alavus
    - Espoo
    - Forssa
    - Haapajärvi
    - Haapavesi
```

```
- Hamina
- Hanko
```

The first example is given to make the bot understand that the user is trying to trigger the greeting intent. Therefore, the response from the bot should be similar. The second intent was more dynamic and it can change based on the user input. Here we have pointed the placename to train the chatbot so that when these questions come, it will automatically go to the action server to find the weather of that specific city.

ii) actions.py: RASA action server works based on the class name. For example, if we want to trigger the checkWeather action, the class in the action file should be named as ActionCheckWeather. The code looks similar to this-

```
class ActionCheckWeather(Action):

def name(self) -> Text:
    return "action_check_weather"

def run(self, dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
    current_place = next(tracker.get_latest_entity_values("place"),
    None)
    #sending city name to weather server
    #return response
```

This way we have picked the city name from the conversation. we have used AccuWeather API to get updates about the weather. Also, there are two other action classes defined in the action server-

```
class GetFeelings(Action):
def name(self):
    return 'action_get_feelings'
class GetExcitement(Action):
def name(self):
    return 'action_get_excitement'
```

The feelings show if the bot is feeling happy or sad based on nearby companion activity. If there is no company nearby and the weather is bad, the bot becomes sad and sends a response to the channel to give a visit to it. Also, if there are users nearby the points of interest, the bot becomes excited.

iii) rules.yml : This file is used to train the Rule Policy (conversation path following a fixed path) by small pieces of conversation which always follow the same path. This file is

mostly used to know about facts from the chatbot where the question and the answer will be always the same. If we consider this code-

```
- rule: Say goodbye anytime the user says goodbye
  steps:
  - intent: goodbye
  - action: utter_goodbye
```

We can see, whenever the user triggers goodbye, the bot will say goodbye in return and this conversation path will always be the same.

iv) stories.yml: On the contrary, the stories file helps to find out patterns from conversations. Here we need to declare the steps of the conversation flow. If we take the default example of the RASA bot where it shows two different sad paths, the code is-

```
- story: sad path 1
  steps:
  - intent: greet
  - action: utter_greet
  - intent: mood_unhappy
  - action: utter_cheer_up
  - action: utter_did_that_help
  - intent: affirm
  - action: utter_happy

- story: sad path 2
  steps:
  - intent: greet
  - action: utter_greet
  - intent: mood_unhappy
  - action: utter_cheer_up
  - action: utter_did_that_help
  - intent: deny
  - action: utter_goodbye
```

First, the greeting intent is triggered and then if the user tells it that he/she is unhappy, it tried to cheer them up. If that works, the bot is happy else, it just closes the conversation. So, from these two different conversation flows, the RASA bot will train the machine learning algorithm to find out similar patterns and act as we have declared in this file.

4.2 Js-son BDI Agent

The next component we have used to trigger RASA intentions is the js-son BDI agent. This is a javascript-based agent where we tried to put modifications to make the agent aware of its nearby environment. The js-son library supports different kinds of agent models but for our case, we found that the Belief-Desire-Intention-Plan fits most. The way these components are changed will be explained down below-

Belief: In this object we have set initial beliefs for the bot and then based on iteration every time these beliefs are updated. Based on the beliefs, agents try to trigger the intentions based on a defined plan to reach its desires.

```
const beliefs = {
  ...Belief('weather', 'sunny'),
  ...Belief('haveCompanionNearby', false),
  ...Belief('feelings', { general: "neutral",
  happiness: "neutral", excitement: "excited"})
}
```

Desire: Desire is the ideal state of the bot agent. The ideal state is a type of state that the bot always wants to reach. For reaching the desired state, the agent tries to follow the intended path with plans. The desired state I added to the bot is

```
const desire = {
  ...Desire('checkWeather', beliefs => {
    return "sunny"
  }),
  ...Desire('companionNearby', beliefs => {
    return true
  }),
  ...Desire('tellStory', beliefs => {
    if (beliefs.haveCompanionNearby) {
      return true
    }
    return false
  })
}
```

As we can see, the best state for the agent will be, if there are companions nearby, the weather is at its best condition and the agent is able to tell stories to the nearby player.

Intention: Intention component shows the plans by which the agent will try to reach the desired state. The plans we have added for this agent bot is-

```
const plans = [
  Plan(intentions => intentions.checkWeather,
    () => [{ respondRequest: 'checkWeather' }]),
  Plan(intentions => intentions.companionNearby,
    () => [{ respondRequest: 'companionNearby' }]),
  Plan(intentions => intentions.tellStory,
    () => [{ respondRequest: 'tellStory' }])
]
```

From the above code, we can see that, the bot keeps checking the weather and player positions to find out if it could reach the desired state of it.

After declaring these three objects, we have created a setInterval function in the node app which continuously triggers the agent's iteration. Before triggering the agent intentions, there are two different checks through REST API calls- the first one is the current weather update and the second one is player positions data from the location-based game. Afterward, the responses update the default belief of the agent and the iteration starts. From the iteration, the agent triggers the intentions of the RASA bot and starts sending messages to the players' Slack channel.

4.3 Data Source

The third module of the architecture is the location-based game we have developed. The game is developed using Unity and in the backend, we have implemented a REST API endpoint. The API endpoint is working as a data collector. Therefore, when the game is played, from time to time the users' location is saved in a data table. Other than user locations, there is also an option for storing points of interest data by users. The modules of the game will be explained here-

Login and Registration: The first step of the game is to register and log in. We are using a username and password model for the registration and login process. We kept storing this very basic information to keep up with the right GDPR for the user. The user login information is saved to the database using the rest endpoint.

Movement and storing POIs: After logging in, the user can show the character on the map. During the walk, the idea is, if the user finds thing interesting, he/she will take a picture, write information about the POIs and upload it to the server. The ER diagram of the backend server-

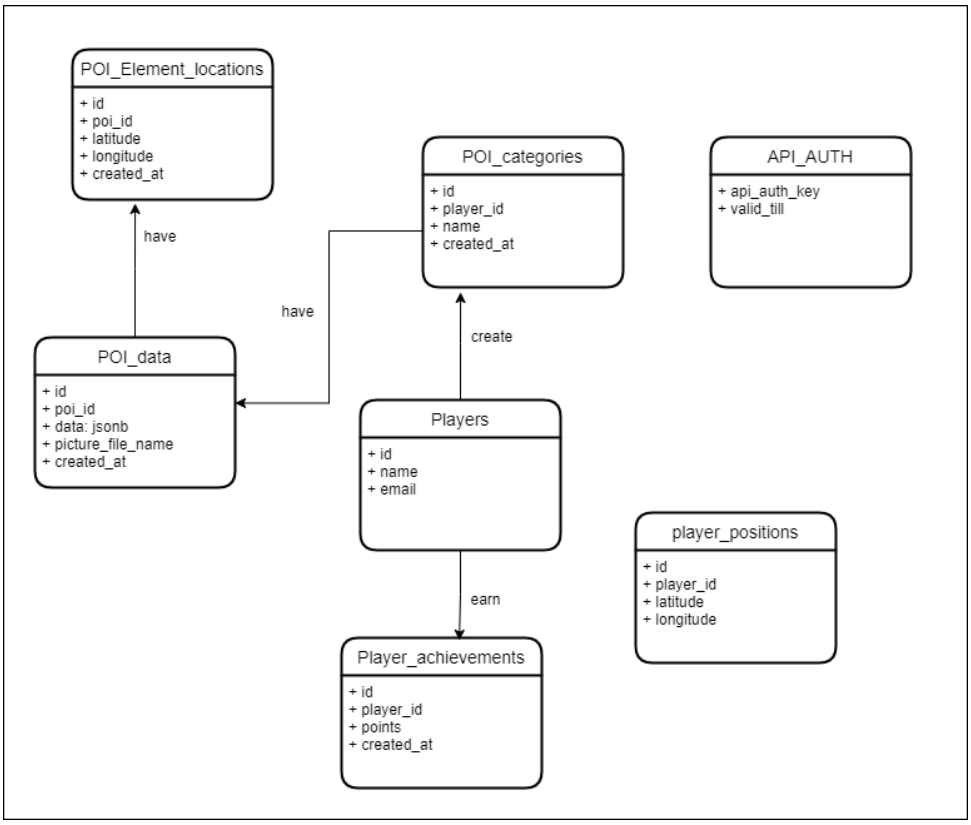

Figure 4.1. *Location Based Game database ER diagram*

The diagram shows the database tables and which information we are collecting from the game. The primary table of the game is the player table. In every 2 minutes of the gameplay, we store the players' location data in the player_positions table. When the player adds a new point of interest, the POI image is stored in the server and the file name is added to the table. Also, the POI geo-location data is stored which is later used by the bot to compare player position with it.

When a user clicks the add POI button, this pop-up comes up. Here user can add the image and information about the POI and upload it.

Figure 4.2. *Add Points of interest*

So, in the field of 'What is this place?', the user can add what is special about this place

and what he/ she should capture using the camera. This POI details is picked by the bot and it tells it as a story when another player comes near that POI.

By clicking the image button, the user can pick an image from the existing gallery or there is also another option to open the camera to capture and upload a picture immediately. Also, there is another feature called capturing the POI. When the user walks near a POI and stays there for 15 seconds, the user captures it. This is just a gamification mechanism to create regional capturing competition among different users.

Figure 4.3. *Points of interest*

The uploaded POIs show on the map like a tree. If another user goes nearby, there will be a popup with images and details about the points of interest. The stored data in the database looks like this-

created_by integer	name character varying	data jsonb
80	person	{"geometry": {"location": {"lat": 61.49352264404297, "lng": 23.77736473083496}, "viewport": {"northeast": {"
81	Rantakasvillisuutta	{"geometry": {"location": {"lat": 61.4889030456543, "lng": 23.789499282836914}, "viewport": {"northeast": {"
81	Sinisorsat järven jäällä	{"geometry": {"location": {"lat": 61.489784240722656, "lng": 23.78626823425293}, "viewport": {"northeast":
82	cute animal	{"geometry": {"location": {"lat": 61.493526458740234, "lng": 23.777372360229492}, "viewport": {"northeast"

Figure 4.4. *Points of interest table*

As mentioned earlier, the node server keeps checking these points of interest positions and compares them with the players' position table. The distance between player and POI latitudes and longitudes are calculated using the haversine [40] formula which is-

a = $\sin^2$(latDifference/2) + cos(lat1).cos(lt2).$\sin^2$(lonDifference/2)

c = 2.atan2(a, (1a))

distance = R.c [R is the radius of the globe]

This equation calculates the distance and when any player position is within 200 meters of points of interest, the chatbot agent triggers an intention and sends a response to RASA. That way, the agent starts communicating in the Slack channel.

The application has three different modules which are tied to each other. The RASA framework needs two different servers. The first one is the RASA default server for the basic conversation. Secondly, for processing the weather update and expressing its feelings based on user activity, it needs a separate action server. Furthermore, there is another component which is the BDI js-son agent. In my architecture, I tried to containerize these components separately in order to decrease the implementation dependency. Therefore, we have implemented docker in this architecture model.

4.4 Docker Orchestration

The docker implementation was done using a simple docker-compose file. we have added the specifications in the yml file along with the environmental requirements. As RASA is Python based, there is complexity between dependency versions. This issue was resolved by the docker container.

The docker yml file looks like this-

```
version: '3.7'
services:
  chatbot:
    build: forestbot
    networks:
      - chatbot-network

  rasa_actions:
    build: rasa-actions
    networks:
      - chatbot-network

  jsson_server:
    build: jsson
    networks:
      - chatbot-network
```

```
networks:
  chatbot-network:
    driver: bridge
```

The containers are connected to the same docker network called chatbot-network. This way the containers are able to communicate with each other inside the internal network.

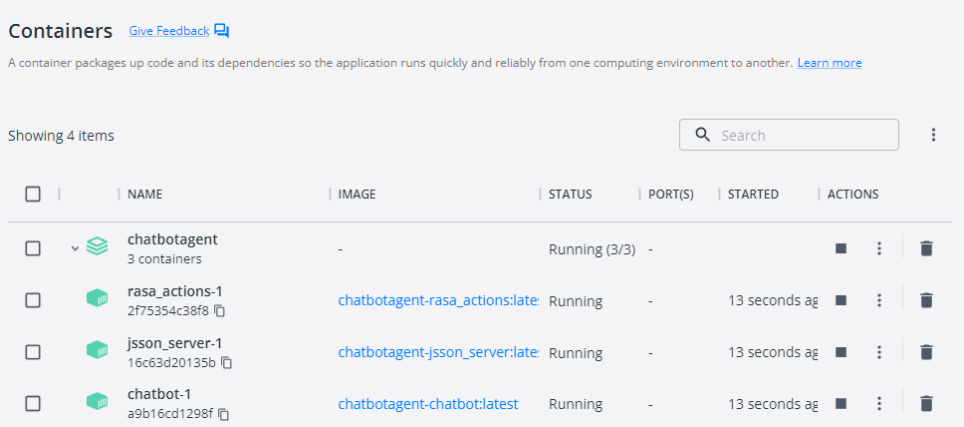

Figure 4.5. *Docker orchestration*

Internal Ports Usage: RASA by default uses two different ports 5005 and 5055. The 5055 is used by the RASA action server. In the case of the node container where the BDI agent is running, that is using port 3000. There was no necessity to expose any container port to the outer world. Inside the orchestration, all the containers are connected to the docker network bridge. Through this, all the containers can communicate with each other within their internal ports.

4.5 Slack Integration

After this integration, the last but not least part was the configuration of Slack. The configuration was done in such a way so that, the agent can send/ initiate a conversation at the common Slack channel. The messages are based on the BDI agent which means, how the bot is currently feeling about its surroundings. If there are players, the agent becomes excited to tell stories about the nearest points of interest of that player.

The slack configuration option was available inside RASA framework. We had to update the credentials yml file with our Slack channel credentials. The format of the credentials is-

```
slack:
  slack_token: "slackSecretToken"
  slack_channel: "channelID"
```

```
slack_signing_secret: "signinSecret"
```

After this configuration, inside the slack api interface, we had to declare the webhook in order to allow the chatbot to be interactive.

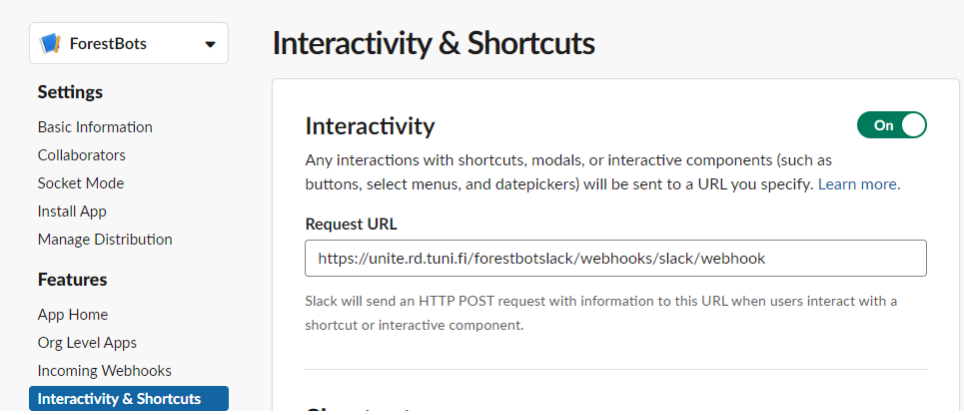

Figure 4.6. *Webhook for Slack interactivity*

Interactivity works based on allowed event subscriptions of the application. Therefore, we have added these scopes and events for the app we created inside Slack.

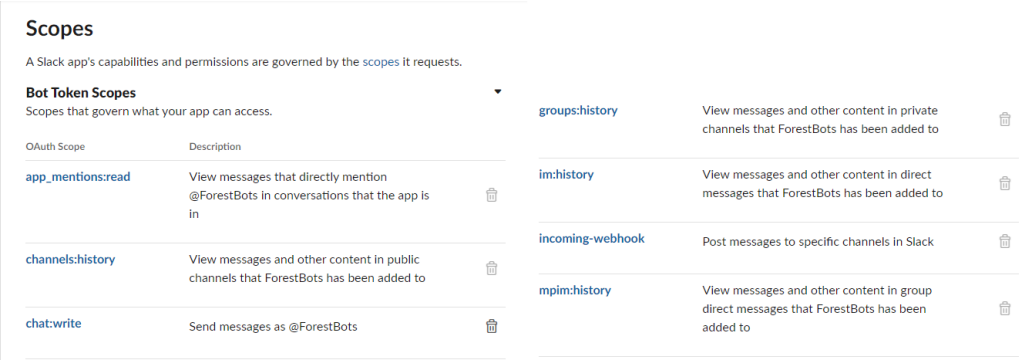

Figure 4.7. *Scopes*

These defined scopes show which parts of the channel are allowed to access by the app. This increases the credibility of the application.

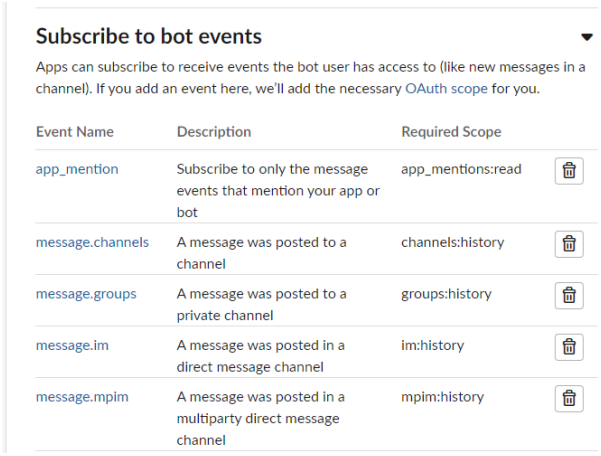

Figure 4.8. *Bot events*

The bot events are used to interact with the bot. For example, inside the conversation in the channel, if the bot is mentioned using '@' by any user, the bot will try to respond to that specific comment. These events are more like trigger points of the bot to write back the user.

After these configurations, the components are working independently and the chatbot is functional in the slack channel. Here are examples of the interaction with the bot-

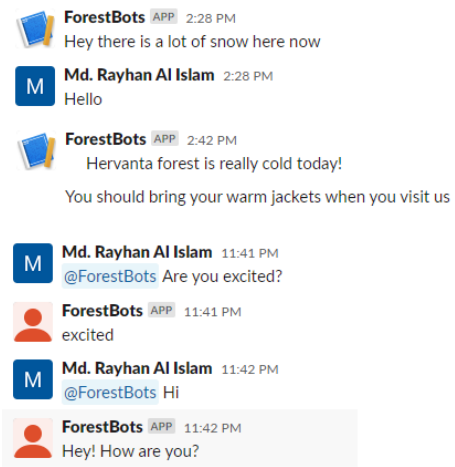

Figure 4.9. *Bot conversation*

This example shows the bot is able to handle the basic conversation. Also, it can provide updates regarding the weather as well.

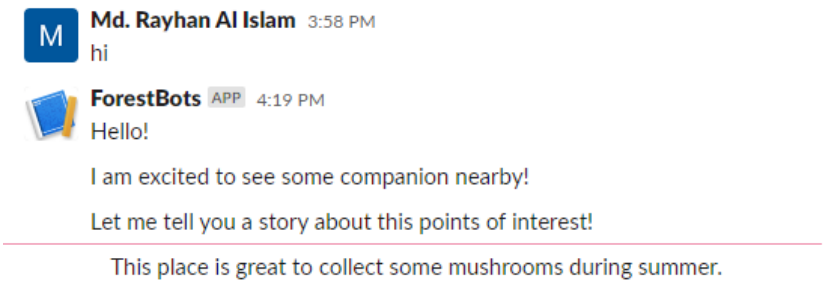

Figure 4.10. *Bot telling stories*

This example shows that the bot believes there are companions nearby and for that reason, it is trying to give information about the nearby place. These points of interest data are coming from that location-based game POI table. The details of the POI are added by other users while playing the game.

The main idea of this implementation is to show how a microservice-style architecture can be implemented regarding chatbots. This architecture also proves that the interactivity and the ability of chatbots can be extended by making the components well organized.

4.6 Deployment

Docker containerized applications are easy to deploy because the environment is taken care of by Docker itself. We just need to make sure that the docker is installed in the server and the existing services of the server are updated to the latest state.

```
sudo yum check-update
```

As we are using a Linux Redhat server, we used this command to update the current applications. After that, we need to prepare the server for installing docker. The first step of the preparation is removing if there are any old docker engines or dependencies available where in the server. In order to do that-

```
sudo yum remove docker \
              docker-client \
              docker-client-latest \
              docker-common \
              docker-latest \
              docker-latest-logrotate \
              docker-logrotate \
              docker-engine \
              podman \
              runc
```

After the cleaning process, we need to use yum-config-manager utility for setting up the docker repository.

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo
https://download.docker.com/linux/rhel/docker-ce.repo
```

After that, the repository will be added to the config manager and we will be able to start the installation process. For installing docker it is required run the next command-

```
sudo yum install docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin
```

After running this, there was no error and it shows the docker was installed completely. For checking if the installation was completed properly and the docker command works we run the next command-

```
docker --version
docker compose --version
```

these commands will show the docker version and also the installed docker-compose version along with it. So, now we have our docker installed. The next step will be pulling the container code from GitLab. We were using GitLab for version control. So, the next command will pull the code from Gitlab and put it into the server-

```
git clone <repository link>
```

This command takes time based on internet speed. As the repository is quite large, it might take up to a minute or so to download all the files on the server. When the download process is done, the next thing we need to do is deploy the orchestration-

```
cd chatbot-architecture
docker compose up
```

The first command is for entering to the container folder and the next one is for starting the deployment process. This will take time to set up app the containers, install all the dependencies and that's it. The application should be up and running. It will keep checking the player positions and keep sending messages in the Slack channel.

5. DISCUSSION

From the background review, it is seen that context awareness can be achieved in multiple ways. The implementations showed [15, 32, 16, 17] the usage of monolithic architecture or mix architecture of monolith and microservices. These architectures are feasible for the specific applications they have developed but, if we want to improve the application scalability and maintenance, microservice is a better option [41]. Also, the architecture we proposed, creates a wider region of context awareness implementation possibilities.

In this chapter, we reflect on our entire research project. Here we will discuss how the implementation answers our research questions. Also, we will explain our contribution towards a scientific and practical level. Furthermore, we will mention the challenges we faced and also what can be done in the future on this work.

5.1 Assessment of the research question

As we can see in Figure 3.11 that the bot is able to read the user's location and understands that the bot the nearby points of interest. In the background, the bot is continuously checking the weather update from the AccuWeather server. Measuring these two contexts, the bot is updating its belief. At this point, the js-son agent takes over and the interation of the js-son BDI starts. The bot looks at its desire and tries to find a plan to reach there. When the bot sees a player within a POI radius, it sends a message to the Slack channel. This act explains that the bot is context-aware and it can take actions based on its nearby environment.

In the case of architecture, figure 3.1 explains the service containers and their communication. To verify this, the benchmark requirements of being a microservice need to be fulfilled [42].

Here some of the mentioned requirements checks have been done from the article-

Easy Access from a Version Control Repository: The RASA bot, action server and the BDI agent can stay in different repositories and still will be able to run as it is.

Continuous Integration: Gitlab's continuous integration can run for building and testing the new changes and integrate it if the tests are passed. Each component of this architecture has the ability to build and run tests independently.

Automated Testing: Automated test cases can be written for the available components in this architecture. The unit tests can run separately without any dependency on each other.

Dependency Management: The RASA bot runs in Python and Python has issues with dependencies. Therefore, we have used docker containers to resolve this dependency issue.

Reusable Container Images: In this project, docker compose is used to containerize the modules. The compose yml file consists of all required information to build and run the application. This containers can run as a container app outside of this orchestration also. So, it is reusuable.

Automated Deployment: As mentioned earlier, the gitlab CI can run for building and testing. When that passes, a deployment can be also done using that pipeline.

Container Orchestration: All the containers are connected into a docker orchestrated cluster. Here containers can communicate with each other and create network bridges to access the local ports also.

These requirements show that the architecture proposed here verifies itself as a microservice-based architecture. By measuring these factors, it can be said that the first research question is answered.

In terms of integrating the external system into the bot, the data source is completely coming from outside of this environment. The game and the game's backend are deployed on a different server. We are making RESTful calls from the bot backend to fetch the data and analyze it on our end. Moreover, the actual context awareness is proved by the game activities themselves. Therefore it answers the second research question.

5.2 Contribution

We have seen that the developers face difficulties in implementation and integration procedure [12], therefore, this architecture will help the developers to modularize the implementation and make the debugging and testing process easier. Also, in this article [32] it is mentioned that the authors faced challenges to make the chatbot start a conversation by itself to increase engagement. Through the architecture implementation, we have also resolved this challenge. Furthermore, from the background review, we have seen that the context of chatbots can vary based on situations. As the modules of our architecture are totally separated, it is possible to implement multiple context awareness through this architecture. Now the chatbot agents trigger the intention when there is player movement near points of interest. If we want to make a context for tracking and creating a leaderboard of the players who walked most in the forest, that can be done by creating another

small service. That service will map through the user movement positions and send it to the bot. This way, several modifications, and implementations can be done with this architecture and improve the context awareness of the bot.

What are the challenges we have faced?

Firstly, designing the data model and plan about what kind of questions might be asked by the users. We also decided that we do not want the bot to be like prompting continuously and answering based on user replies. Rather, we wanted to keep the conversation as natural as possible. Also, about the weather update, the current version of API is provided by AccuWeather which gives a limited amount of API calls for the free version. So, if this bot is used on a large scale we have to change that. Usually, there is not much free data available for weather forecasting, so this issue is quite common. The next challenge I faced was, listing the city names for data training. The training of the dataset gets better if comparatively, we can feed a good amount of data to it. Therefore, I had to put a large list of city names of Finland in the nlu file of RASA to train it and make it capable to show weather updates based on city names.

So in brief, with this microservice-based architecture, application scalability improvement, BDI agent integration with a bot, and bot integration with a location-based game have been done successfully.

What can be done in the future on this?

I believe this chatbot architecture has huge potential in the chatbot development sector. The next thing which can be done is, rather using a microservice, this kind of implementation can be tested in a monolith or other architecture to find out the efficiency. Also, other chatbot platforms can be used to test the architecture, for example- the azure bots and such. Also, this chatbot integration can be done inside a window of the location-based game we designed.

6. CONCLUSION

Finally, this research has shed light on the critical function of chatbots' context awareness across multiple disciplines. A survey of the literature and practical studies found that chatbots have immense potential in a variety of applications spanning from customer service to healthcare and education. Chatbots have emerged as essential tools for businesses looking to improve customer service, shorten response times, and provide personalized experiences. Furthermore, they have the ability to improve internal processes, increase productivity, and produce cost savings.

The findings also highlighted the importance of chatbot context awareness in terms of customer pleasure and engagement. It may provide great user experiences by responding quickly and efficiently, resulting in increased user engagement, loyalty, and overall happiness. Chatbots' capacity to adapt and learn from user interactions allows them to provide personalized recommendations, targeted help, and a sense of conversational involvement, boosting the user experience even more.

However, the implementation of the context-aware bot is not without challenges. Implementation complexity, maintenance, and scalability need to be addressed throughout the process. As chatbot technology continues to evolve, it is crucial for organizations and developers to ensure transparency, accountability, and user empowerment in the design and deployment of chatbot systems. This research study adds to the existing body of knowledge by offering a thorough examination of the present status of context-aware implementation technologies and their impact on user experience. It provides practitioners, researchers, and policymakers with useful insights into the opportunities and pitfalls related to chatbot implementation.

In conclusion, chatbots have emerged as a promising technology with the potential to revolutionize user experiences across various domains. By harnessing their capabilities effectively and addressing associated challenges, organizations can create meaningful interactions, improve customer satisfaction, and drive business success in the digital era.

REFERENCES

[1] Joseph Weizenbaum. “ELIZA—a Computer Program for the Study of Natural Language Communication between Man and Machine”. In: *Commun. ACM* 9.1 (Jan. 1966), pp. 36–45. ISSN: 0001-0782. DOI: 10.1145/365153.365168. URL: https://doi.org/10.1145/365153.365168.

[2] Eleni Adamopoulou and Lefteris Moussiades. “Chatbots: History, technology, and applications”. In: *Machine Learning with Applications* 2 (2020), p. 100006.

[3] Naz Albayrak, Aydeniz Özdemir, and Engin Zeydan. “An overview of artificial intelligence based chatbots and an example chatbot application”. In: *2018 26th signal processing and communications applications conference (SIU)*. IEEE. 2018, pp. 1–4.

[4] Asbjørn Følstad, Cecilie Bertinussen Nordheim, and Cato Alexander Bjørkli. “What makes users trust a chatbot for customer service? An exploratory interview study”. In: *Internet Science: 5th International Conference, INSCI 2018, St. Petersburg, Russia, October 24–26, 2018, Proceedings 5*. Springer. 2018, pp. 194–208.

[5] Ahmed Fadhil and Gianluca Schiavo. “Designing for health chatbots”. In: *arXiv preprint arXiv:1902.09022* (2019).

[6] Terrance Lopez and Meer Qamber. *The benefits and drawbacks of implementing chatbots in higher education: A case study for international students at Jönköping University*. 2022.

[7] Mubashra Akhtar, Julia Neidhardt, and Hannes Werthner. “The potential of chatbots: analysis of chatbot conversations”. In: *2019 IEEE 21st conference on business informatics (CBI)*. Vol. 1. IEEE. 2019, pp. 397–404.

[8] *Chatbot Statistics*. https://bloggingwizard.com/chatbot-statistics/. Accessed: 2023-05-29.

[9] Mahesh Vanjani, Milam Aiken, and Mina Park. “Chatbots for multilingual conversations”. In: *Journal of Management Science and Business Intelligence* 4.1 (2019), pp. 19–24.

[10] Gregory D Abowd et al. “Towards a better understanding of context and context-awareness”. In: *Handheld and Ubiquitous Computing: First International Symposium, HUC’99 Karlsruhe, Germany, September 27–29, 1999 Proceedings 1*. Springer. 1999, pp. 304–307.

[11] Wei Liu, Xue Li, and Daoli Huang. “A survey on context awareness”. In: *2011 International Conference on Computer Science and Service System (CSSS)*. IEEE. 2011, pp. 144–147.

[12] Ahmad Abdellatif et al. “Challenges in chatbot development: A study of stack overflow posts”. In: *Proceedings of the 17th international conference on mining software repositories*. 2020, pp. 174–185.

[13] AM Rahman, Abdullah Al Mamun, and Alma Islam. “Programming challenges of chatbot: Current and future prospective”. In: *2017 IEEE region 10 humanitarian technology conference (R10-HTC)*. IEEE. 2017, pp. 75–78.

[14] Omar Al-Debagy and Peter Martinek. “A Comparative Review of Microservices and Monolithic Architectures”. In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. 2018, pp. 000149–000154. DOI: 10.1109/CINTI.2018.8928192.

[15] Fabio Clarizia et al. “A Context-Aware Chatbot for Tourist Destinations”. In: *2019 15th International Conference on Signal-Image Technology Internet-Based Systems (SITIS)*. 2019, pp. 348–354. DOI: 10.1109/SITIS.2019.00063.

[16] Arshit Gupta et al. “Casa-nlu: Context-aware self-attentive natural language understanding for task-oriented chatbots”. In: *arXiv preprint arXiv:1909.08705* (2019).

[17] Ping-Huan Kuo et al. “Multi-sensor context-aware based chatbot model: An application of humanoid companion robot”. In: *Sensors* 21.15 (2021), p. 5132.

[18] Nathan Cruz Coulson, Stelios Sotiriadis, and Nik Bessis. “Adaptive microservice scaling for elastic applications”. In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 4195–4202.

[19] Hanzhong Zheng, Justin Kramer, and Shi-Kuo Chang. “Auto-Modularity Enforcement Framework Using Micro-service Architecture.” In: *J. Vis. Lang. Sentient Syst.* 2020.2 (2020), pp. 17–22.

[20] Dharmendra Shadija, Mo Rezai, and Richard Hill. “Towards an understanding of microservices”. In: *2017 23rd International Conference on Automation and Computing (ICAC)*. IEEE. 2017, pp. 1–6.

[21] Surya Roca et al. “Microservice chatbot architecture for chronic patient support”. In: *Journal of Biomedical Informatics* 102 (2020), p. 103305. ISSN: 1532-0464. DOI: https://doi.org/10.1016/j.jbi.2019.103305. URL: https://www.sciencedirect.com/science/article/pii/S1532046419302242.

[22] S Shah and Seema Shah. “A comparison of various chatbot frameworks”. In: *J. Multi-Criteria Decis. Anal* 6 (2019), pp. 375–383.

[23] *Amazon Lex documentation*. Accessed: 2023-06-06. URL: https://docs.aws.amazon.com/lex/index.html.

[24] *Hubot documentation*. Accessed: 2023-06-06. URL: https://hubot.github.com/docs/.

[25] *IBM watson documentation*. Accessed: 2023-06-06. URL: https://cloud.ibm.com/developer/watson/documentation.

[26] *RASA documentation*. Accessed: 2023-06-06. URL: https://rasa.com/docs/.

[27] Naveen Kumar. *Understanding the Basics of Rasa - Open source conversational AI*. 2020. URL: https://smazee.com/blog/basics-of-rasa.

[28] M Bratman. “Intention, plans, and practical reason, Harvard Un”. In: *Cambridge, MA* (1987).

[29] Federico Peinado, Marc Cavazza, and David Pizzi. “Revisiting character-based affective storytelling under a narrative BDI framework”. In: *Interactive Storytelling: First Joint International Conference on Interactive Digital Storytelling, ICIDS 2008 Erfurt, Germany, November 26-29, 2008 Proceedings 1*. Springer. 2008, pp. 83–88.

[30] Wayne Wobcke et al. “A BDI agent architecture for dialogue modelling and coordination in a smart personal assistant”. In: *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*. IEEE. 2005, pp. 323–329.

[31] Minha Lee et al. “Caring for Vincent: a chatbot for self-compassion”. In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–13.

[32] Sharon Grundmann. “A bdi-based virtual agent for training child helpline counsellors”. In: (2022).

[33] Timotheus Kampik and Juan Carlos Nieves. “JS-son-a lean, extensible JavaScript agent programming library”. In: *Engineering Multi-Agent Systems: 7th International Workshop, EMAS 2019, Montreal, QC, Canada, May 13–14, 2019, Revised Selected Papers 7*. Springer. 2020, pp. 215–234.

[34] Janne Paavilainen et al. “The Pokémon GO experience: A location-based augmented reality mobile game goes mainstream”. In: *Proceedings of the 2017 CHI conference on human factors in computing systems*. 2017, pp. 2493–2498.

[35] Sampsa Rauti and Samuli Laato. “Location-based games as interfaces for collecting user data”. In: *Trends and Innovations in Information Systems and Technologies: Volume 2 8*. Springer. 2020, pp. 631–642.

[36] Irene Celino et al. “Urbanopoly–A social and location-based game with a purpose to crowdsource your urban data”. In: *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*. IEEE. 2012, pp. 910–913.

[37] Ionut Andone et al. “Impact of location-based games on phone usage and movement: A case study on Pokémon GO”. In: *Proceedings of the 19th International Conference on Human-Computer Interaction with Mobile Devices and Services*. 2017, pp. 1–8.

[38] Marek Moravcik and Martin Kontsek. “Overview of Docker container orchestration tools”. In: *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE. 2020, pp. 475–480.

[39] *Top 10 Microservices Design Principles*. Accessed: 2023-06-08. URL: https://www.developer.com/design/microservices-design-principles/.

[40] Akshay Upadhyay. *Haversine formula - Calculate geographic distance on earth*. June 2018. URL: https://www.igismap.com/haversine-formula-calculate-geographic-distance-earth/.

[41] Davide Taibi et al. “Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages”. In: *Proceedings of the XP2017 Scientific Workshops*. 2017, pp. 1–5.

[42] Carlos M Aderaldo et al. “Benchmark requirements for microservices architecture research”. In: *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. IEEE. 2017, pp. 8–13.

APPENDIX A: CODES

A.1 RASA

RASA action file:

```
    # This files contains your custom actions which can
       be used to run


from typing import Any, Text, Dict, List

from rasa_sdk import Action, Tracker
from rasa_sdk.executor import CollectingDispatcher
from rasa_sdk.events import SlotSet
import requests
import json
#
#
class ActionCheckWeather(Action):

    def name(self) -> Text:
        return "action_check_weather"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text,
               Any]]:
        current_place = next(tracker.
           get_latest_entity_values("place"), None)
        #sending city name to weather server
        find_city = requests.get('https://dataservice.
           accuweather.com/locations/v1/cities/search?
           apikey=<api key>&q='+current_place)
```

```
        if find_city.status_code == 200:
            city_json_data = find_city.json()
            print(city_json_data[0]['Key'])
            weather_update = requests.get('https://
               dataservice.accuweather.com/forecasts/v1/
               daily/1day/'+city_json_data[0]['Key']+'?
               apikey=<API key>').json()
            dispatcher.utter_message(text=weather_update
               ["Headline"]["Text"])
        else:
            dispatcher.utter_message(text="Are you sure
               about the city name? Please check the
               spelling")



        return []


class GetFeelings(Action):
    def name(self):
        return 'action_get_feelings'
    def run(self, dispatcher, tracker, domain):
        headers = {
        "Content-Type": "application/json",
        "Accept": "application/json",
        }
        data = {
            "fields": {
                "conversation": tracker.sender_id,
                "what":"feeling"
            }
        }
        url = "http://jsson_server:3000/agent/
           get_feelings"
        status = requests.post(url, headers=headers, data
           =json.dumps(data)).json()
        dispatcher.utter_message(text=status["0"]["
           feelings_report"])
```

```
class GetExcitement(Action):
    def name(self):
        return 'action_get_excitement'
    def run(self, dispatcher, tracker, domain):
        headers = {
        "Content-Type": "application/json",
        "Accept": "application/json",
        }
        data = {
            "fields": {
                "conversation": tracker.sender_id,
                "what":"excitement"
            }
        }
        url = "http://jsson_server:3000/agent/
           get_feelings"
        status = requests.post(url, headers=headers, data
           =json.dumps(data)).json()
        dispatcher.utter_message(text=status["0"]["
           feelings_report"])
```

RASA nlu yml file:

```
    version: "3.1"

nlu:
- intent: greet
  examples: |
    - hey
    - hello
    - hi
    - hello there
    - good morning
    - good evening
    - moin
    - hey there
    - let's go
    - hey dude
    - goodmorning
```

```
    - goodevening
    - good afternoon

- intent: goodbye
  examples: |
    - cu
    - good by
    - cee you later
    - good night
    - bye
    - goodbye
    - have a nice day
    - see you around
    - bye bye
    - see you later

- intent: affirm
  examples: |
    - yes
    - y
    - indeed
    - of course
    - that sounds good
    - correct

- intent: deny
  examples: |
    - no
    - n
    - never
    - I don't think so
    - don't like that
    - no way
    - not really

- intent: mood_great
  examples: |
    - perfect
    - great
    - amazing
```

```
    - feeling like a king
    - wonderful
    - I am feeling very good
    - I am great
    - I am amazing
    - I am going to save the world
    - super stoked
    - extremely good
    - so so perfect
    - so good
    - so perfect

- intent: mood_unhappy
  examples: |
    - my day was horrible
    - I am sad
    - I don't feel very well
    - I am disappointed
    - super sad
    - I'm so sad
    - sad
    - very sad
    - unhappy
    - not good
    - not very good
    - extremly sad
    - so saad
    - so sad

- intent: bot_challenge
  examples: |
    - are you a bot?
    - are you a human?
    - am I talking to a bot?
    - am I talking to a human?

- intent: confirm_exist
    - is there a [tree](poi_name)?
    - is there a [rock](poi_name)?
    - is there any [animal](poi_name)?
```

```
- lookup: poi_name
  examples: |
    - palm tree
    - rock
    - reindeer

- intent: weather_update
  examples: |
    - what is the weather in [tampere](place)?
    - what is the weather in [helsinki](place)?
    - what is the weather like in [turku](place)?
    - what is the weather like in [espoo](place)?

- intent: real_feelings
  examples: |
    - how do you really feel?
    - what are your real feelings?
    - what are your true feelings?

- intent: excitement
  examples: |
    - are you excited?
    - what is your excitement level?


- lookup: place
  examples: |
    - Akaa
    - Alaj rvi
    - Alavus
    - Espoo
    - Forssa
    - Haapaj rvi
    - Haapavesi
    - Hamina
    - Hanko
    - Harjavalta
    - Haukipudas
    - Heinola
    - Helsinki
```

```
    - Huittinen
    - Hyvink
    - H meenlinna
    - Iisalmi
    - Ikaalinen
    - Imatra
    - Pietarsaari
    - Joensuu
    - Juankoski
    - Jyv skyl
    - J ms
    - J rvenp
    - Kaarina
    - Kajaani
    - Kalajoki
    - Kankaanp
    ...
    ...
```

RASA rules file:

```
    version: "3.1"

rules:

- rule: Say goodbye anytime the user says goodbye
  steps:
  - intent: goodbye
  - action: utter_goodbye

- rule: Say 'I am a bot' anytime the user challenges
  steps:
  - intent: bot_challenge
  - action: utter_iamabot

- rule: agent info
  steps:
  - intent: EXTERNAL_agent_info
  - action: utter_agent_info

- rule: real feelings
```

```
  steps:
    - intent: real_feelings
    - action: action_get_feelings

- rule: excitement
  steps:
    - intent: excitement
    - action: action_get_excitement
\end{verbatim}

RASA stories file:

\begin{verbatim}
    version: "3.1"

stories:

- story: happy path
  steps:
  - intent: greet
  - action: utter_greet
  - intent: mood_great
  - action: utter_happy

- story: sad path 1
  steps:
  - intent: greet
  - action: utter_greet
  - intent: mood_unhappy
  - action: utter_cheer_up
  - action: utter_did_that_help
  - intent: affirm
  - action: utter_happy

- story: sad path 2
  steps:
  - intent: greet
  - action: utter_greet
  - intent: mood_unhappy
  - action: utter_cheer_up
```

```
  - action: utter_did_that_help
  - intent: deny
  - action: utter_goodbye

- story: provide info path 1
  steps:
    - intent : inform
    - action: utter_inform

- story: weather update
  steps:
    - intent: weather_update
    - action: action_check_weather
```

A.2 BDI node server

Agent sending info for RASA intention trigger:

```
    function send_info(arg) {
  console.log(`arg was => ${arg}`);
  let postData = {"name": "EXTERNAL_agent_info", "
     entities": {"agent": "ForestAgent"}};
  var clientServerOptions = {
    uri: `http://localhost:5005/conversations/${
       conversationid}/trigger_intent?output_channel=
       latest`,
    body: JSON.stringify(postData),
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    }
  }
  request(clientServerOptions, function (error, response)
      {
    if(response!= null && response.body!= null){
     // console.log("Received from RASAS: ",error,
        response.body);
    }
      return;
  });
```

```
}
```

Getting BOT feelings after BDI iteration:

```
    app.post('/agent/get_feelings', (req, res) => {
  console.log('received: %s', JSON.stringify(req.body))
  console.log("Forest:" + JSON.stringify(forest))
  environment.state.userRequests.push(req.body.fields.
     what)
  environment.run(1)
  const jsonString = Object.assign({}, response);
  conversationid = req.body.fields.conversation;
  console.log("conversation id: "+conversationid);
  res.json(jsonString);
  response = [];
})
```