Alabama Liquid Snake

University of Pretoria

Epi-Use

Botic - Privacy aware chatbot
System Requirements Specification

Justin Grenfell - u16028440
Peter Msimanga - u13042352
Alicia Mulder - u14283124
Kyle Gaunt - u15330967
Lesego Mabe - u15055214

Contents

1 Introduction 3
- 1.1 Purpose . . . 3
- 1.2 Background . . . 3
- 1.3 Scope . . . 3
- 1.4 UML Domain Model . . . 4
- 1.5 Definitions, acronyms, and abbreviations . . . 5
- 1.6 Overview of Document . . . 5

2 Overall Description 5
- 2.1 Product Perspective . . . 5
  - 2.1.1 System Interfaces . . . 6
  - 2.1.2 User Interfaces . . . 6
  - 2.1.3 Hardware Interfaces . . . 6
- 2.2 Product Functions . . . 6
  - 2.2.1 Authentication . . . 7
  - 2.2.2 Information Scraper . . . 8
  - 2.2.3 Chatbot/Ticketbot . . . 9
  - 2.2.4 AI Training . . . 10
  - 2.2.5 Database Manager . . . 10
- 2.3 User Characteristics . . . 10
  - 2.3.1 Customer . . . 10
  - 2.3.2 Customer Support Representative . . . 11
  - 2.3.3 Administrator . . . 11

3 Specific Requirements 11
- 3.1 Functional Requirements . . . 11
  - 3.1.1 User Interface . . . 11
  - 3.1.2 Information Scraper . . . 11
  - 3.1.3 Query Classification . . . 12
  - 3.1.4 Response Generation . . . 12
  - 3.1.5 Chatbot . . . 12
  - 3.1.6 Chatbot Trainer . . . 12
  - 3.1.7 Data Persistence . . . 12
- 3.2 Organizing the Specific Requirements . . . 13
  - 3.2.1 Traceability Matrix . . . 13
- 3.3 Software System Attributes . . . 13
  - 3.3.1 Availability . . . 13
  - 3.3.2 Security . . . 14
  - 3.3.3 Reliability . . . 14
  - 3.3.4 Performance . . . 14
  - 3.3.5 Scalability . . . 15
  - 3.3.6 Maintainability . . . 15
- 3.4 Challenges . . . 15
- 3.5 Technology Decisions . . . 16
  - 3.5.1 Technologies . . . 16
  - 3.5.2 Justifications . . . 16

4 Architectural Design 17
4.1 System Type . . . 17
4.2 Architectural Tactics . . . 17
4.2.1 Availability . . . 17
4.2.2 Security . . . 18
4.2.3 Reliability . . . 19
4.2.4 Performance . . . 20
4.2.5 Scalability . . . 20
4.2.6 Maintainability . . . 20
4.3 Architectural Style . . . 21
4.3.1 Determine System Types . . . 21
4.3.2 Applying Architectural Styles . . . 21

5 References 24

1 Introduction

1.1 Purpose

The purpose of this document is to present a detailed description of Botic- the privacy aware chatbot. It will explain in good detail the purpose and the features of the system, the interfaces of the system, what the system will do, the constraints under which it must operate and also how the system will react to user and external stimuli. This document is intended for the stakeholders, that is the COS 301 staff and lectures as well as the CS department lectures and our client EPI USE Labs– represented by Mrs. Jhani Coetzee and Mr. Tiaan Scheepers, and the developers, Alabama Liquid Snake, of the system and will be proposed to all of our stakeholders for their approval.

1.2 Background

A crucial part of any business in today's economic climate is customer service. Those companies that are willing to go the extra mile for their customers are seen as being a cut above the rest. With superior customer service, a company can not only bring in new clients who want an experience that seems to cater to them as an individual, but also successfully retain existing clients by dealing with their issues efficiently and effectively.

In order to do so, there needs to be a system that can record customer feedback and act on it in as soon as possible. In the past, this has been achieved by employing a large number of people around the clock that sat and waited for queries, handled them and sent back the results.

While this works, it is not only inefficient (different employees may respond better or worse than others, employees may not follow protocols, mistakes may be made regularly) but financially costly as well. On top of that, when dealing accounts and queries, customers may inadvertently divulge private information that is not applicable to their case, but may leave them vulnerable should that information become public knowledge.

What if one central system could seamlessly record, interpret and act on the requests of multiple users 24/7 and prevent them from transmitting sensitive data unless absolutely necessary?

1.3 Scope

Botic is the solution! One system that can not only record user queries, but sanitize their content by filtering out any "data risks" and act on the provided information, returning the appropriate response. Trained on historical data, the system uses artificial intelligence to analyze requests and act accordingly. It scrapes all data before transmission to ensure that no sensitive information is sent to or from the client without clearance from the sender.

Should the system be unable to find a suitable response, the request will be handed off to a customer support representative who will then deal with the request. Once that case has been handled, the system will have learned how to deal with future requests of that type and will be able to return a response based on this learning. This will ensure a high level of efficacy for the system as a whole.

1.4 UML Domain Model

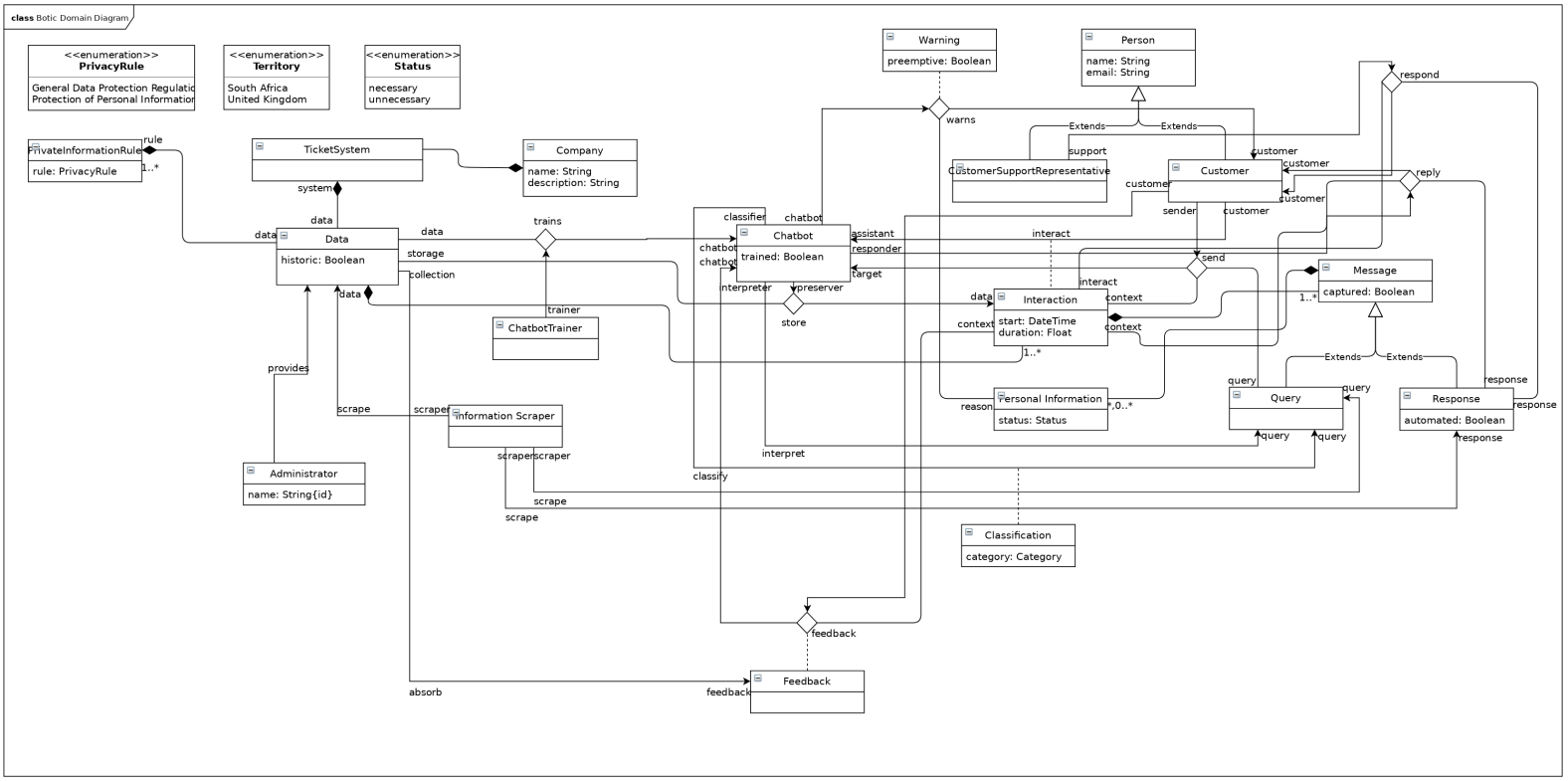

Figure 1: UML Domain Model of the Botic System

1.5 Definitions, acronyms, and abbreviations

Chatbot	A program that is designed to simulate a conversation as if it were a human, in order to assist one who queries with some task or inquiry. In the context of this project, the chatbot will be used to assit customers in a ticket system (customer service system).
AI	Artificial Intelligence; this refers to programmed intelligence, or rather, intelligence that is demostrated by computers usually to mimick human intelligence in some specific area or even in general terms.
Historic Data	Data that is collected and stored over some considerable period of it time, especially, in the context of this project, with the purpose of being used to train a chatbot.
POPI Act	Protection of Personal Information Act; an South African piece of legislature aimed at protecting the right of its citizens to privacy, especially in the Internet and its periphery. It aims to "provide for the rights of persons regarding unsolicited electronic communications and automated decision making; to regulate the flow of personal information access the borders of the Republic."[1]
Ticket System	An online platform, used by a business, made for processing customer queries and issues on products, services and the like.
Personal Information	Sensative and identifying information; the likes of which permission should be asked before sharing or processing.
CS	Computer Science, as in the academic discipline of Computer Science.
Scrub	Detect private information and highlight it according to severity.
SPA	Single Page Application; a web application that dynamically changes a signle page to display all of its contents.
Heroku	Deployment platform.
Docker	Containerization platform*.
Auth0	Authorization server; third party software/service; provides authorization and authentication as a service.*

1.6 Overview of Document

The next chapter, the Overall Description section, of this document gives an overview of the functionality of the project. It describes the 'informal' requirements and is used to establish a context for technical requirements specification in the next chapter.

The third chapter, the Requirement Specification section, of this document is written especially for the developers and describes in terns the details of the functionality of the product.

Both sections of the document describe the same software product in its entirety, but are intended for different audiences and thus use different languages, but seeing as virtually everyone reading this document has a CS background, the main focus ought to be the third section and it is given priority as a result.

This document is structured according to the IEEE 830-1998 SRS Standard[2], as recommended by:[3].

2 Overall Description

2.1 Product Perspective

This system was originally meant to be used in an already existing customer support ticketing system, or ticket system in short. This is to say that the ticketbot- or rather, chatbot, was meant to interface with the ticket system and also be trained with the ticket system's historic data without the risk of exposing customer personal information.

This is system is meant to process a ticket system's messages or tickets, let the users know if they are exposing personal information, and respond intelligently to the messages otherwise divert the queries to a client representative if no sufficient response can be found.

However, for the purposes of this project, we will use a simple SPA to mimick the ticketing system.

2.1.1 System Interfaces

The deployment diagram for the Botic system:

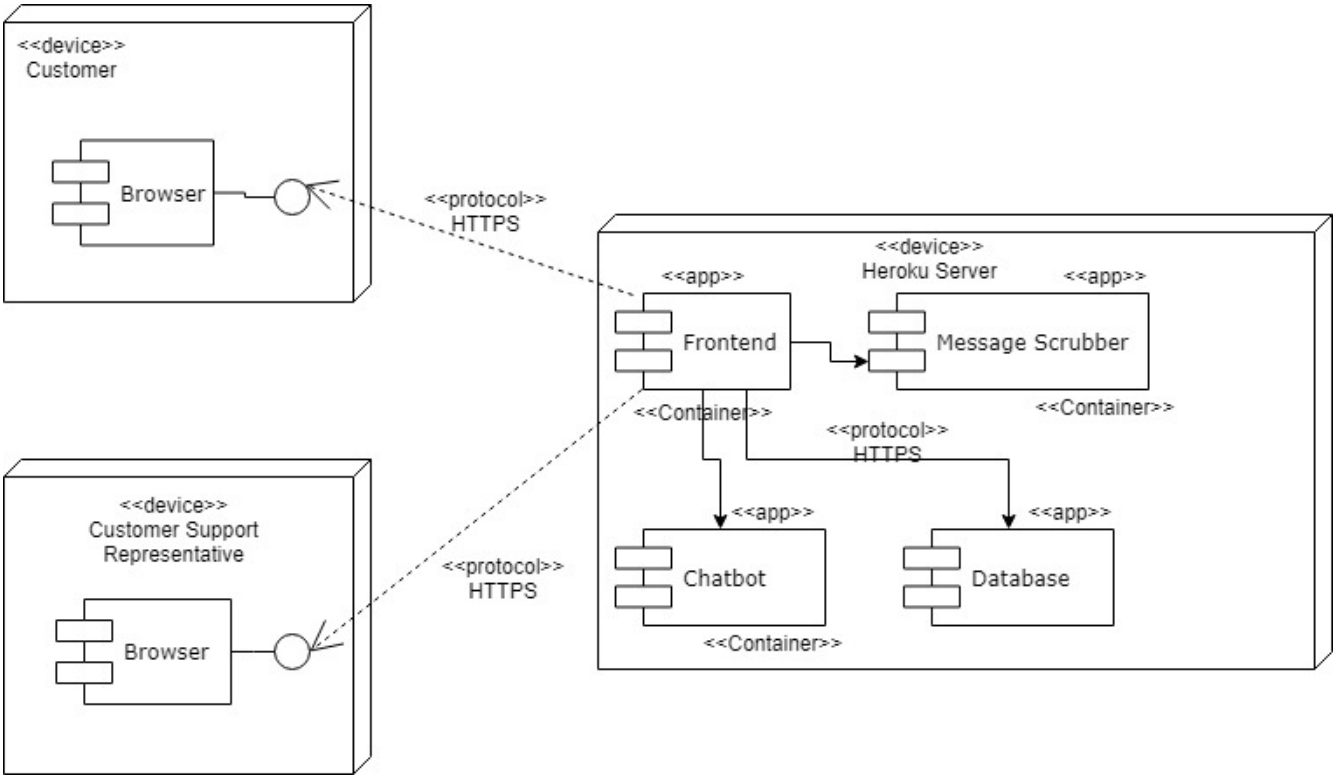

Figure 2: UML Deployment Diagram

2.1.2 User Interfaces

The user interface, which is ultimately the chatbot ticket interface, is meant to be a SPA– Single Page Application. This should be made available online in a manner that supports all major web browsers.

2.1.3 Hardware Interfaces

Each subsystem will be containerized into a Docker container, which in and of themselves use Linux 64-bit architecture. Ports will be mapped to port 80; this is the standard port for API's and is the port that is exposed on our deployment platform, Heroku.

2.2 Product Functions

The Botic system consists of the frontend, AI backend which includes the chatbot API, message scrubber API, the classifier API and their neural networks. The frontend is a Single Page Application which houses the main user interface of the system. The Chatbot API is responsible for recieving messages and returning responses.

The Message Scrubber API checks whether or not information provided to it is private and if so, determines the severity of it. The neural networks are trained using Machine Learning and provide the system with its intelligences-privacy classification, responses to queries and queries classification.

2.2.1 Authentication

This subsystem is to be responsible for the authentication and the authorization of customers and customer representatives. This involves the creation and update of user profiles of the mentioned roles. Auth0 will be used as an Authorization server and will be configured to make us of the OAuth 2.0 authorization framework in order to provide us with enough flexibility to create different grants/permissions to easily authorize different users for different activities in the system.

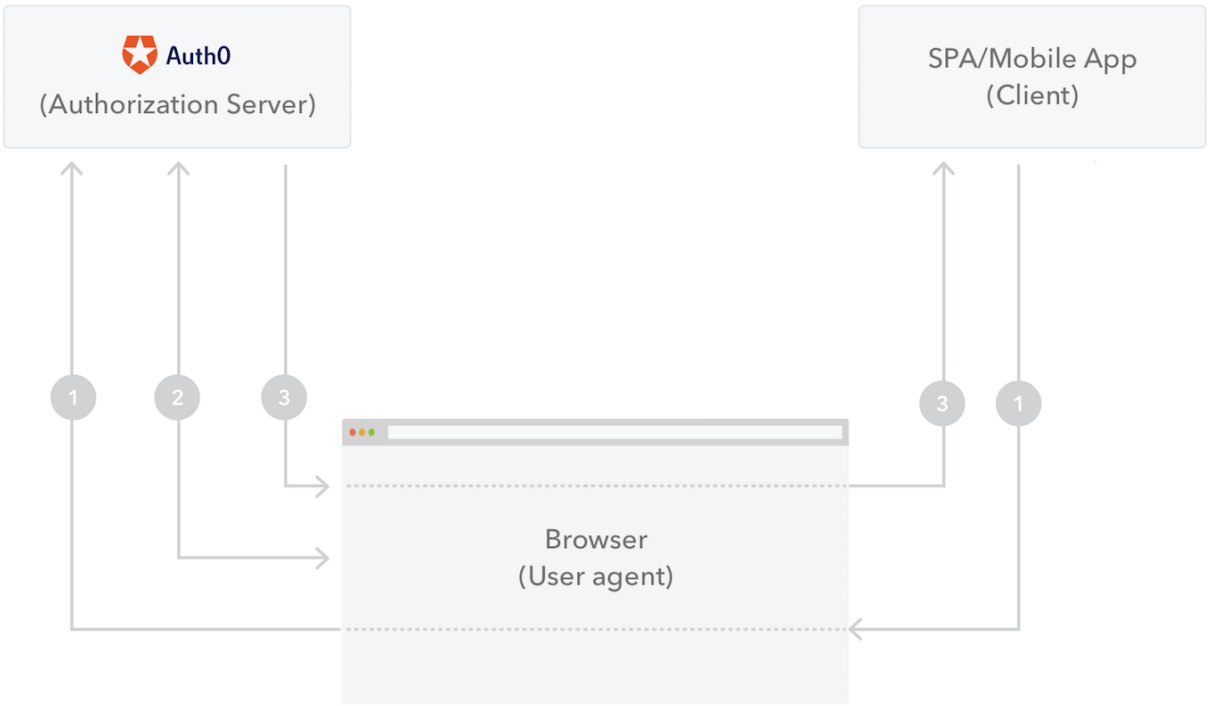

Figure 3: Auth0 Solution Diagram[4]

1. The frontend initiates the flow when the user clicks on the sign in button and redirects the browser to the Auth0 /authroze endpoint so that the user can be authenticated.

2. Auth0 then authenticates the user. If the user is using one of the supported social logins, they will be shown a consent page where there are permissions, which will be given to our system- Botic, that a listed and they would have to give their consent for Botic, through Auth0, to use.

3. Auth0 then redirects to Botic with an Access Token in the hash fragment of the URI, which the authentication module consumes. The happens because Auth0 calls an endpoint on the Botic fronend which processes the token from the URI. Botic can now extract the token, and the user is logged in.[4]

2.2.2 Information Scraper

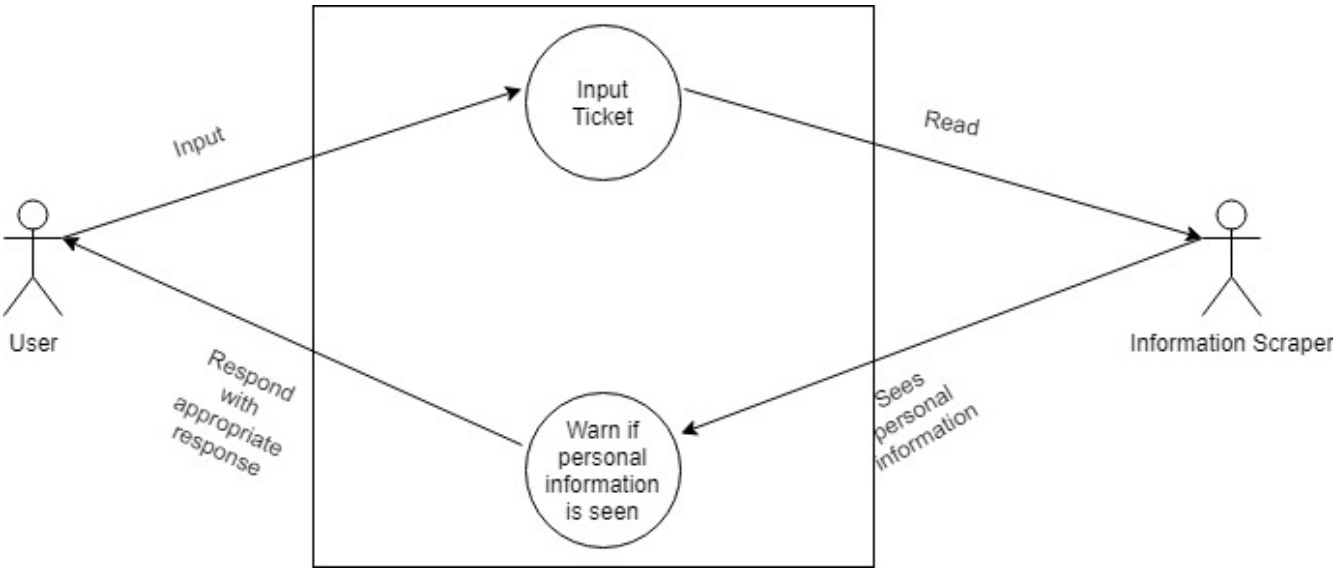

Figure 4: UML Use Case of the Information Scraper

The Information Scraper, or Message Scrubber, is a subsystem that is responsible for identifying personally identifying information as the user types in their query. It uses the PrivateInfoClassifier to classify words given to it and to produce a severity index. It is called by an interface that is implemented as an Angular service in the Botic frontend. The API itself is implemented in Python and served using gunicorn. It is containerized using Docker to make it more portable, and also to make it possible to run it and other subsystems in the same infrastructure without much difficulty.

2.2.3 Chatbot/Ticketbot

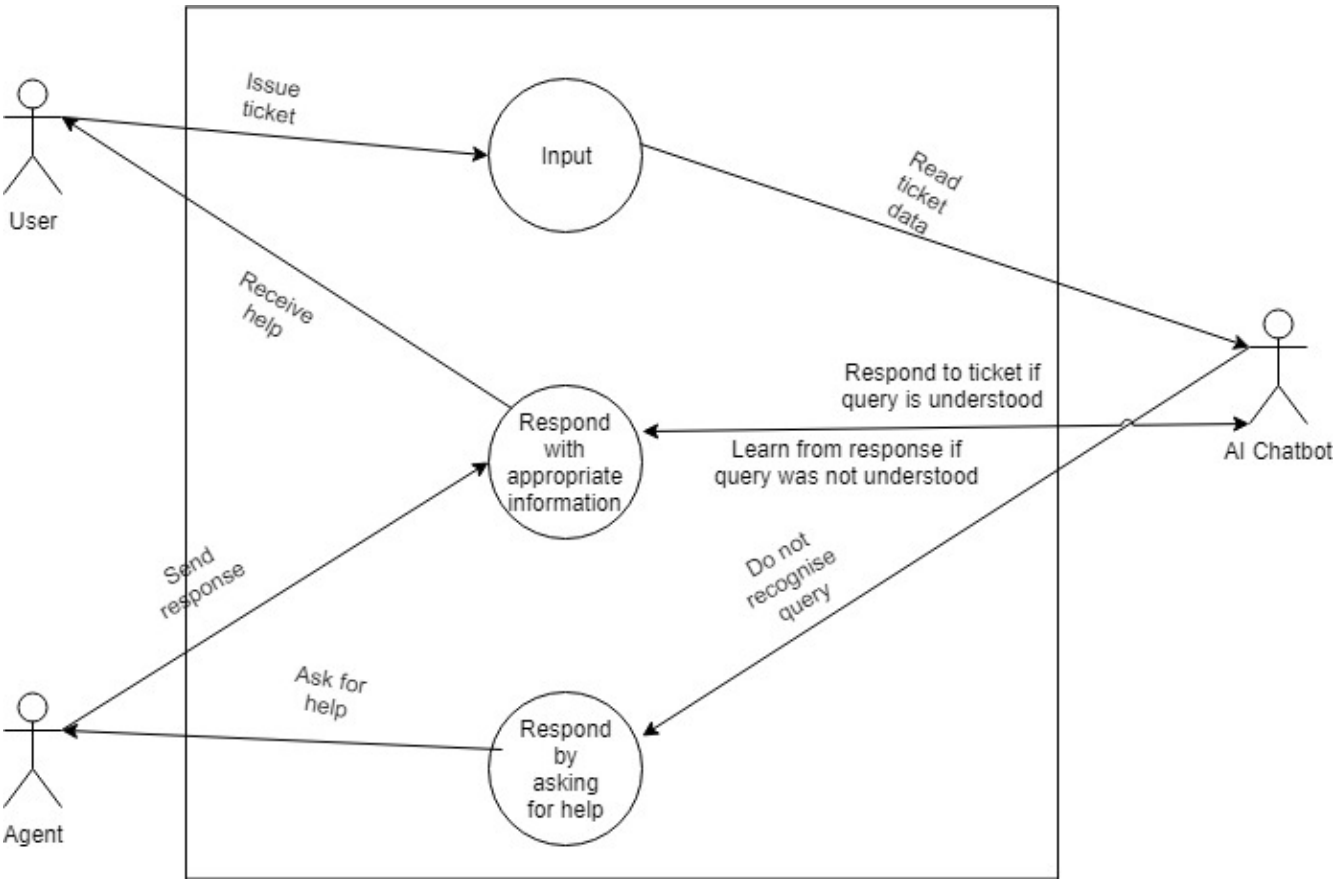

Figure 5: UML Use Case of the Chatbot

The Chatbot or 'Ticketbot' is the subsystem that is responsible for taking in customer queries and answering those queries. The Chatbot consults a QueryClassifier with a query to classify it i.e. whether it is a password, configuration, or other query for example. The classification and the query are then used to find the best response for the query using a neural network of responses, which was initially trained using historical data. The Chatbot is implemented using Nodejs and it is also containerized, for the same reasons as the previous subsystem.

2.2.4 AI Training

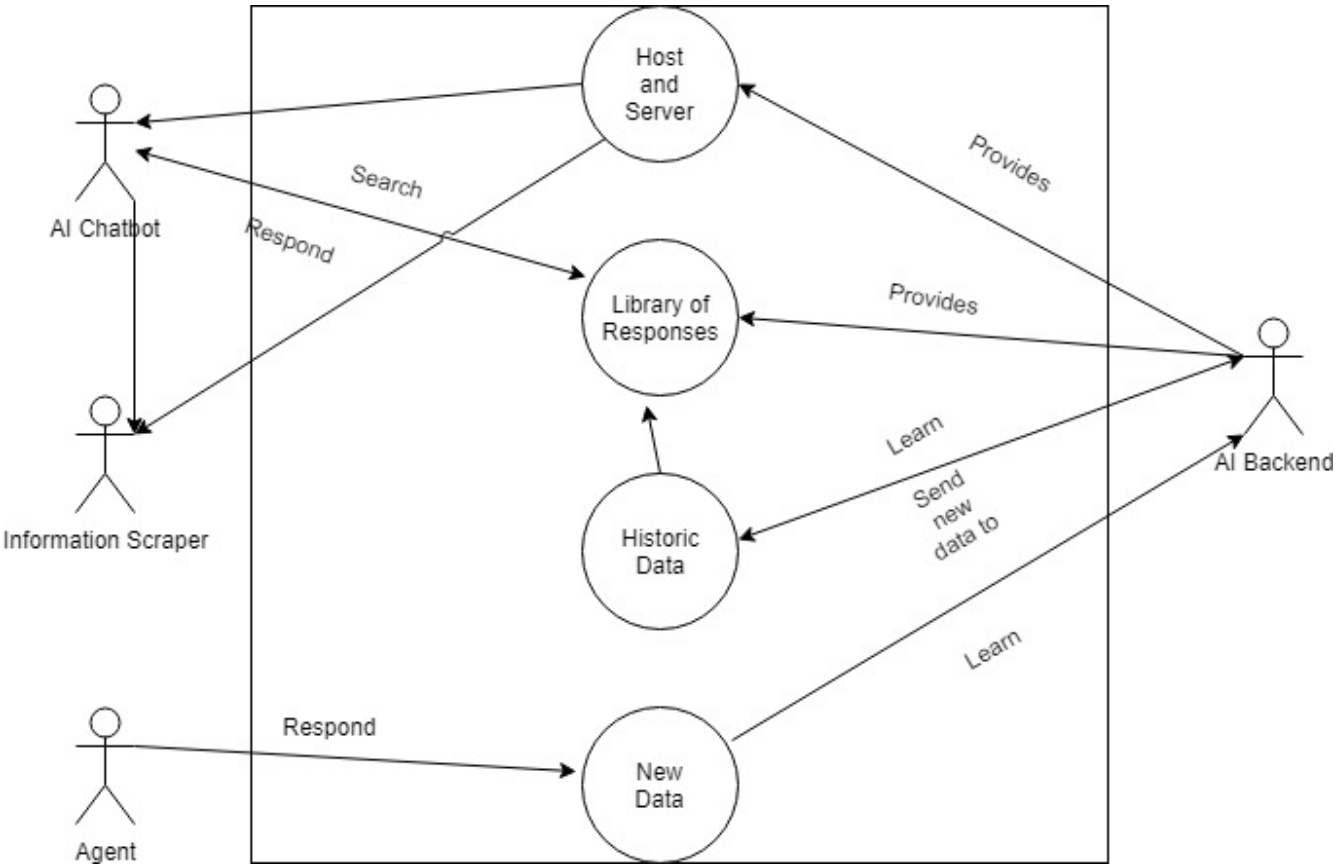

Figure 6: UML Use Case of the AI Backend

The AI Training subsystem is responsible for training the PrivateInfoClassifier, the QueryClassifier and the Responses neural networks. This is done using historic data which belongs to the ticketing system.

2.2.5 Database Manager

This subsystem is responsible for managing persistance in the system. It is called when data needs to be stored, updated, and deleted. Each endpoint will require that the subsystem calling it, identify itself. We intend to have this API secured using JSON Web Tokens. It will be implemented using Java, which when correctly uses, tends to outperform Nodejs is speed[5] and is a much more rebust language than that of NodeJs's JavaScript[5]. This language choice is also made with consideration to the data analytics what will come about when analyzing and processing logs and their data.

2.3 User Characteristics

2.3.1 Customer

The customer will be submitting information to the system in order to deal with account related queries and, in doing so, may unintentionally submit private/sensitive information that could lead to a breach of confidentiality. Any information sent through by the customer will be sanitized by the system and cleaned up before it is transmitted.

2.3.2 Customer Support Representative

This user will be notified when the automated system is unable to interpret the customer's request and said request will be forwarded. The user will have the ability to respond with the result, whereby the system will sanitize the data once more and send it through to the customer.

2.3.3 Administrator

The system administrator is a super user who will be able to manage the system as well as register customers and customer support representative. The administrator will be able to feed the chatbot historic interactions so that it can be trained with the data to improve the correctness of response generation.

Users with Administrator credentials will have the ability to change the settings and operational preferences of the program to better suit the needs of its company's requirements. Admin will have control over the program as a whole including access control, privacy preferences and overall system operation.

3 Specific Requirements

3.1 Functional Requirements

Requirements are labelled with "R" and constraints with "C."

3.1.1 User Interface

R1.1 The system must allow a customer to enter a query and click on a button to send it.

R1.2 The system must warn a customer of personal information included in a query.

R2.2 The system must be able to highlight personally identifying information according to severity index.

R3.3.2.2 The system must be able to highlight personally identifying information according to severity index.

R3.3.2.3 The system must be able to warn the client representative if they have entered identifying information.

R4.1 The system must allow customers to thumbs up a query response.

R4.2 The system must allow customers to thumbs down a query response.

R5 The system must allow queries to be sent to customer support representatives if not answered satisfactorily.

C1 The system must use an Angular Single Page Application for the user interface.

3.1.2 Information Scraper

R2.1 The system must be able to attach a severity to the personally identifying information.

R3.1 The system must scrape its customer query responses for personal information.

R3.3.2.1 The system must be able to identify personal information in a customer representative's response.

C3.1 The system must use word2vec for identifying personal information in customer queries.

C5.2 The system must determine if the response contains personally identifying information.

3.1.3 Query Classification

R3.2 The system must be able to classify the user queries.

C3.2 The system must use word2vec for classifying customer queries.

3.1.4 Response Generation

R3.3.1 The system must generate a response if it certain that it can.

C5.1 The system must generate an automated response based on the query classification.

3.1.5 Chatbot

R1.3 The system must be able to recieve customer queries.

R3.3.2 The system must be able to send the query to a customer support representative if it cannot obtain an appropriate response.

R3.4 The system must be able to send a query response back to a customer.

R4.3 The system must be able to recieve customer feedback.

R5 The system must allow queries to be sent to customer support representatives if not answered satisfactorily.

R8 The system must interface with the currently existing ticket system.

C2 The system must provide an API for the SPA to interact with.

3.1.6 Chatbot Trainer

R6.1 The system must store previous customer interactions with positive feedback.

R7 The system must must be trained with previous customer queries and responses.

C4 The system must use Machine Learning or Deep Neural Networks in order to be trained with previous customer queries and responses.

3.1.7 Data Persistence

R9.1 The system must scrape customer interation data for personal information before storing.

3.2 Organizing the Specific Requirements

3.2.1 Traceability Matrix

R	UI	Info Scraper	Query Classification	Resp Generation	Chatbot	Chatbot Trainer	Data Persistence
R1.1	X						
R1.2	X						
R1.3				X			
R2.1		X					
R2.2	X						
R3.1		X					
R3.2			X				
R3.3.1				X			
R3.3.2					X		
R3.3.2.1		X					
R3.3.2.2	X						
R3.3.2.3	X						
R3.4					X		
R4.1	X						
R4.2	X						
R4.3	X						
R5	X				X		
R6.1						X	
R7						X	
R8					X		
R9.1							X
C1	X						
C2					X		
C3.1	X						
C3.2			X				
C4						X	
C5.1				X			
C5.2		X					

Key: UI = User Interface, Info Scraper = Information Scraper, Resp Generation = Response Generation.

3.3 Software System Attributes

The non-functional requirements below will be listed by priority.

3.3.1 Availability

R1.1. The system must have high availability to handle customer queries.

R1.1.1. The system should be available at least 99 percent of the time.

R1.2. The system must ensure that errors that occur throughout the system are handled appropriately and provide sufficient information.

R1.2.1. The system must provide error messages when errors occur.

R1.2.2. The system must ensure to keep a traces that show what led to errors.

R1.3. The system must ensure that errors are localized and that their effect is minimized throughout the system.

3.3.2 Security

R1.1. The system must be able to authenticate users and authorize them to access system features.

R1.1.1. The system must be able to identify and authenticate customers.

R1.1.2. The system must be able to identify and authenticate customer support representatives.

R1.1.3. The system must be able to deny users who haven't been authenticated to access system features

R1.2. The system must be able to allow new users to register for user profiles for authentication.

R1.3. The system must be able to allow users to update their password.

R1.4. The system must ensure that confidentiality of customer and customer support representative interactions are ensured and maintained across the system.

R1.4.1. The system must ensure that customers can interact with the system in a secured manner.

R1.4.2. The system must ensure that customer queries are sent in a secured manner.

R1.4.3. The system must ensure that customer support representatives care interact with the system in a secured manner.

R1.4.4. The system must ensure that customer support representative response are sent in a secured manner.

R1.4.5. The system must ensure that all queries and responses are processed in a secured manner.

R1.5. The system must ensure that information disclosed during error management is not revealing of internal architecture, design, and configuration information.

3.3.3 Reliability

R1.1. The system must ensure that responses to customer queries are done in a reliable manner.

R1.1.1. The system must ensure that customer support representative are authorized to respond to customer queries.

R1.1.2. The system must ensure that queries are responses sent throughout the system are complete and consistent.

R1.2. The system must ensure that it is at least 80 percent certain that an autogenerated response is correct before responding to a query.

3.3.4 Performance

R1.1. The system must ensure that personal information is highlighted according to severity in real-time.

R1.1.1. The system must ensure that a severity of a word is recieved within a second of it being typed.

R1.1.2. The system must ensure that a word or set of words containing personal information are highlighted in less than a second after recieving the severity.

3.3.5 Scalability

1. The system should be able to scale appropriately to accommodate additional/growing customer queries, especially during peak work hours; it would be useful if the resources scaled down as well during "off peak" hours.

2. We have chosen to deploy our system to Docker, it is used in part to allow for efficient and easy scaling.

3.3.6 Maintainability

1. The system structure will be modular to adhere to the concept of low coupling and high cohesion. This would help to make it maintainable since updated systems result in localized changes instead of changes everywhere throughout the system.

2. We will create a coding standards document which we will also adhere to throughout the system in order to increase readability.

3.4 Challenges

1. Being able to understand what the user is asking even if they use 'slang' abbreviations. For example, using "R U" instead of "are you".

2. Not being able to respond adequately to a new user query due to limited or insufficient training.

3. Time it would take for a representative to get back to a user if the bot does not know what to respond.

4. Being prepared for unexpected inputs.

5. Understanding who the users are, where they are coming from and what they talked about in the past without being able to store/capture and train on data related to specific user data.

6. Context sensitivity: Handling the users' possible propensity to change topics in the middle of an interaction which could lead to a possible breakdown in communication (the risk of which is also possible if the AI chooses the wrong context from an ambiguous line of conversation).

7. Ability to determine intent, especially when what is said does not quite match what is meant. (Natural language processing). Examples:

   (a) Misused phrases
   (b) Intonation
   (c) Double meanings
   (d) Passive aggression
   (e) Poor pronunciation
   (f) Speech impairments
   (g) Slang
   (h) Non-native speakers

8. Users requesting multiple tasks and the chatbot being able to respond to those multiple queries.

3.5 Technology Decisions

3.5.1 Technologies

1. Angular
2. Fasttext
3. Docker
4. TravisCI
5. NodeJS
6. Python
7. Git
8. Gitkraken

3.5.2 Justifications

1. Angular: A highly portable framework that allows for rapid prototyping, instancing and general development of highly modular, portable applications and components thereof. Chosen because it was specified for by the client but supported by us because of the large amount of support the framework has, being built on nodejs and javascript as a whole. We as a team enjoy the benefits of wide-sweeping platform support.
2. FastText: An open source library that allows users and developers to create and use text classifiers and text representation. This is the main interface that we are using to train our AI chatbot to recognise the existence of personal information in a given string as well as recognise the type of query that the user is entering for the purposes of giving a suitable response.
3. Docker: A main goal for this project is to be portable across server implementations and client configurations. Hence the use of web technologies. To solve the server implementations docker provides invaluable aid in the form of containers that can house our backend systems without reck or concern for specific idiosyncrasies across azure, ubuntu and other services.
4. TravisCI: Travis is a powerful service that we use to stage integration for our projects. The industry has moved to a standard wherein constant integration, deployment and updating have become the expected norm. To this end, Travis provides tools to integrate new builds on the fly.
5. NodeJS: Fast server-side javascript by way of Google Chrome's runtime implementation is exactly what we need to create a networked application such as this. Because Angular is built on Nodejs, the backwards compatibility with existing Nodejs packages is invaluable to us as they mean that we do not have to implement any functionality that hasn't already been considered to be a solved problem.
6. Git, GitHub, ZenHub and GitKraken: Git is an industry standard tool for version control so it is obvious that we would use it. Furthermore, because of its ubiquity, git integrates with a large number of tools that make development, version control and project management easier or larger teams.
    (a) Github: Hosts out code online to trigger hooks in other pieces of software (i.e. travis, docker, heroku) for the purposes of building, deployment and integration. It also keeps track of the versions and iterations of the code that each member of the group has created.

(b) ZenHub: This combines the Kanban method of organising seen in applications like trello and the code integrated version control of github to create a project management environment that when managed well, allows for efficient creation and handling of short and long term issues in the project including but not limited to coding and documentation

(c) GitKraken: The main problem with git is its aged command line interface that, while intuitive if one is proficient with it, does little for those who have an understanding of the methodology but lack the time required to master a piece of software that is meant to make the main job easier. Enter GitKraken, this piece of software offers a graphical interface for git that is easy to understand, as well as additional functionality such as a visualisation of the software tree and Kanban boards for organisation. We chose this as a team partly because it streamlines the problem of version control but also partly because we are granted a free professional license by the github student developer pack.

4 Architectural Design

4.1 System Type

Our system type is an interactive system, as it is focused on the customer's interaction with the Chatbot. This means that the interaction begins and ends with the customer. The N-tier architecture is useful for the design of interactive systems. This architecture is breaks the system down to a number of relatively independent and loosely coupled layers.[3]

4.2 Architectural Tactics

In order to achieve and adhere to our quality attributes, we use architectural tactics. These are design decisions that influence the achievement of a quality attribute as they directly affect the system's response to some stimulus[6].

Below we list the architectural tactics that will be used satisfy our system's quality attributes (non-functional requirements), by order of the nonfunctional requirements listed above:

4.2.1 Availability

In order to fascilitate the application of tactics that improve availability, we would have to include the following into our system:

Logging of faults

Notifying appropriate entities of the faults

Disable the source of events causing the fault

Be temporarily available if need be

Fix or mask the fault/failure

or Operate in a degraded mode

In order for these to be effected, faults need to be detected, the system needs to be able to recover from faults, and it is appropriate to have a measure of preventing faults before they happen.

In order to log faults, the fault detection mechanism or tactic need to be able to store the detected 'faulty' state. The memento design pattern would clearly be useful here. We can even go as far as to have types of faults being associated with the types of mechanisms set out to identify each. For these, we may

apply the Abstract Factory Pattern in order to produce specific fault logs, and store them in our fault logging database through the memento pattern. Event's can be stored in this way, so that we may be able to have a trace of operations that led to an error.

Once that is stored in the caretaker, which in and of itself is a state change, we notify the module responsible for notifying the relevant parties of faults in the system. In order to effect these responsibilities, we would have to assign functionality that stores fault logs, and notifies users. These will be done in reference to use cases, because availability is an attribute that affects users - which are the actors in the use cases. The fault detection mechanisms (architectural tactics) depend on the particular operations of the modules, thus each of the modules or subsystems stand to have their own fault detection mechanisms (availability architectural patterns). This logic applies to tactics responsible for recovering from faults, and preventing faults.

The table below provides the subsystems yeilded from functional clusters for high cohesion, and their system types as well as our chosen fault detection tactic, recovery tactic and fault prevention tactic. These are only subsystems that are necessary, and ordered according to priority. The architectural tactics are solution specific, therefore we expect to add many more of them as the development of the project continues; software engineering is a wicked problem afterall:

Subsystem	Subsystem Type	Fault Detection	Recovery from Faults	Fault Prevention
Chatbot Subsystem	Transformational Subsystem	ping/echo, Retry	Exception Handling	Removal from Serv

Please note that the Chatbot Subsystem consists of the Message Scraper, Classifier and the Response Generator subsystems; thus all the mentioned tactics actually apply to each of this subsystem's components.

Two modules will be produced here: one for log management, and another for notification. These require that data be persisted.

4.2.2 Security

Looking at the quality requirements involving security in following the design checklist specified by [3], we find that it is useful to use physical security as a model for software security as indicated by [6]. This results in the four categories of focus for this quality attribute: detect, resist, react, and recover.

In order to support the design and the analysis of security in our system we will pay attention to the following, and ensure that these responsibilities are assigned and met:

Identify the actor

Authenticate the actor

Authorize actors

Grant or deny access to data or services

Record attempts to access or modify data or services

Encrypt data

Recognize reduced availability for resources or services and inform appropriate personnel and restrict access

Recover from an attack

Verify checksums and hash values

With regards to our data we shall ensure that the following is observed:

Seperation of data of different sensitivities

Ensure different access to data of different sensitivities

Ensure that access to sensitive data is logged and that the log files is suitably protected

Ensure that data is suitably encrypted and that keys are separated from the encrypted data

Ensure that data can be restored if it is inappropriately modified

This attribute modifies or enhances modules introduced in the availability requirement analysis above, this includes the assignment of new responsibilities in particular, certain logs have to be stored securely– those logging access to sensitive information. It also makes use of the notification module to let the appropriate personnel know that their system is being attacked, when these attacks are detected.

The table below includes the architectural tactics we will employ with respect to each of the four categories (detection, resistance, reaction, and recovery) we defined earlier, the subsystems and their subsystem types. These are in order of priority:

Subsystem	Subsystem Type	Detection	Resistance	Reaction	Recovery
User Interface Subsystem	Interactive Subsystem	Detect service denial, Detect message delay	Identify actors, Authenticate actors, Authorize actors, Encrypt data, Separate entities	Revoke access, Lock computer, Inform Actors	Maintain Audit Trail
Chabtot Subsystem	Transformational Subsystem	Verify Message Integrity, Detect service denial	Identify actors, Authorize actors, Encrypt data	Revoke access, Inform Actors	Maintain Audit Trail
Chatbot Trainer Subsystem	Transformational Subsystem		Authorize actors, Encrypt data	Inform Actors	Maintain Audit Trail, Data Model checklist above*
Database Subsystem	Database subsystem		Authorize actors, Encrypt data, Separate entities	Inform Actors	Maintain Audit Trail, Data Model checklist above*

Note: we mention the UI subsystem, but what we really mean are the operations performable by the roles of Administrator and Customer Representative. The UI is how they interact with the system.

These are not meant to be exhaustive; they are meant to be revised according to future needs and adapted with feedback.

4.2.3 Reliability

All of these concerns are encapsulated in the availability analysis above as well as the security analysis above. The requirements have nonetheless been associated with the revelant use cases.

4.2.4 Performance

Here we will look at design decisions that affect the performance of the system. Here we will consider two contributors to the response time: processing time and blocked time[6]. In these our architectural tactics will be concerned with the control of resource demand as well as the management of resources.

A design checklist proposed by [6] suggests useful design considerations for the sake of ensuring performance; other than just following this checklist as in, on pages 142 to 144, we will note especially that under resource management, it is advised that we manage and monitor important resources under normal and overloaded operation; resources such as process and thread models– a detail that involves the availability quality attribute.

The table below includes the architectural tactics that we will employ with respect to the control of resource demand and the management of resources in our system. This will only be for the performance critical components, and ordered by importance. The list contains the following columns: subsystem, subsystem type, control resource demand, and manage resources. This is not an exhaustive list, but alas software engineering is a wicked problem:

Subsystem	Subsystem Type	Control Resource Demand	Manage Resources
Chatbot Subsystem	Transformational Subsystem	Prioritize Events, Increase resource efficiency, Reduce overhead	Increase resources*, Introduce currency, Load balancer, Sch resources
Message Scraper	Transformation Subsystem	Prioritize Events, Increase resource efficiency	Increase resource*, Introduce currency, Load balancer, sch resources

Note: With regards to "Increase resources" we mean to say that a recommended deployment environment ought to be a cloud platform such as PaaS (Platform as a Service) which would dynamically increase or reduce resources based on demand and bill the customer accordingly.

4.2.5 Scalability

Looking into design decisions affecting scalability, we have to consider the impact of adding or of removing resources, and these measures will reflect on associated availability as well as the load that will be assigned to existing and new resources[6]. Here, two kinds of scalability will be dealt with: horizontal scalability (adding more resources to individual units), vertical scalability (adding more resources to individual nodes).

4.2.6 Maintainability

Maintainability may encapsulate some aspects of modifiability as well as testability and availability. All three of these will be observed as well as that which is required for maintanence management from [3].

In as far as modifiability is concerned, our architectural tactics in this regard, concern all of our system and its subsystems; they are reducing the size of modules through spliting modules, increasing cohesion by increasing semantic coherence, reducing coupling by encapsulation, using an intermediary, restrict dependencies, refactoring and abstract common services.

Looking at architectural tactics for testability we are mostly going to rely on language specific testing tools, especially those that can allow us to follow state, pause and playback tests. It is also useful to follow documentation. The high priority subsystems, in terms of availability and functional responsibilities are

high priorities for testing.

Availability has already been covered above. As far as preparing the system for maintainability is concern, during development we will pay close attention to the application of software design principles as well as the application of software design patterns.

"An aspect of testing in that arena is logging of operational data produced by the system, so that when failures occur, the logged data can be analyzed in the lab to try to reproduce the faults."[6]

4.3 Architectural Style

4.3.1 Determine System Types

From a top view of the entire system, we have observed that the system is more of an interactive system type. Our system is focused on the customer's interaction with the Chatbot, the customer support's interaction, and the administrator's interaction. This means that the interaction begins and ends with each user. The results in the ordering of all subsystems into a number of layers. This has the consequence of creating a clear and well-documented separation of concerns[6].

The layers that result are the following:

1. The User Interface Layer
2. The Chatbot Layer
3. The Training Layer
4. The Persistance Layer

*This comes from the application of a 4-Tier Architecture.

These four layers we treat as the total system's four main subsystems. Each of these layers, or subsystems, produce exhibit system types of their own. Going into each, we will apply the architectural design process until the we find that the individual leaf node elements are relatively easy to design and implement[3].

Now we have a look at the "User Interface Layer." This layer deals heavily with actor requests and responses; all actors primarily interact with this subsystem in order satisfy business processes. This is clearly an interactive subsystem.

Looking at the Chatbot layer, we observe that the layers in a transformational subsystem. The Support layer, or rather the Training layer we observe a transformational subsystem. The Perisistance Layer, however, is very clearly an object-persistence subsystem.

4.3.2 Applying Architectural Styles

Entire System The whole system adheres to the 4-Tier architectural style. It's layers or subsystems include: The User Interface Layer, the Chatbot Layer, the Support/Training Layer, and lastly the Persistence Layer. This architectural style ensure that all the subsystems and modules can be seperate and thus developed seperately with minimal interaction between them (changes), ensuring adherance to the software design principles of high cohesion and low coupling[6]. This makes our entire system more portable and modifiable (and thus maintainable). Here is the solution:

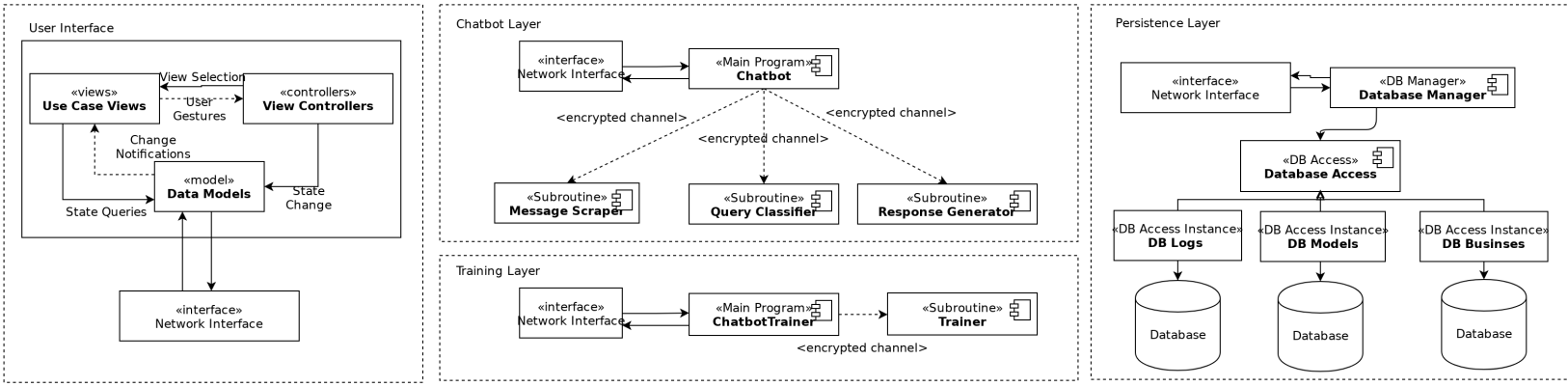

Figure 7: Architectural Stye of the System

The allowed-to-use relation here is denoted by the geometric adjacency of the layers from left to right– layers on the left use layers on the right and the layers on the right are used by the layers on the left ONLY. The connections between each layer are made to be through encrypted channels. An appropriate technology will be used to implement this during the implementation phase e.g. SSL.

The User Interface Subsystem The User Interface Layer that is also an interactive subsystem uses an MVC architectural style. Using this architectural style, we ensure that user interface interface functionality, something that tends to change frequently especially in response to knew design trends *Quote IMY theory notes, be kept seperate from the application functionality and yet be ever responsive to user input, the most important thing in an interactive system[6]. This is the solution:

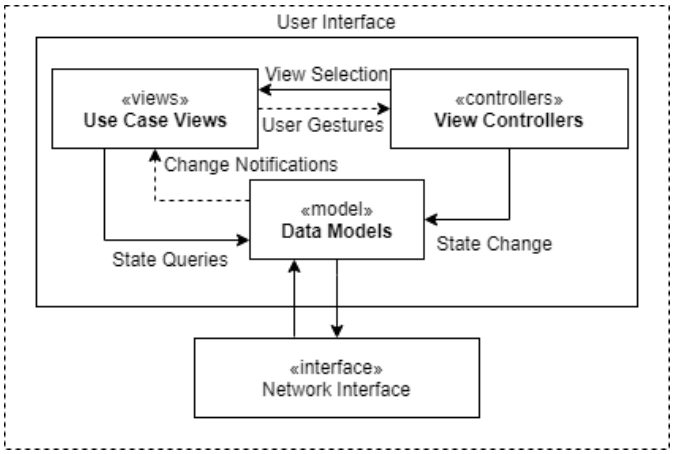

Figure 8: Architectural Style of the User Interface

The Chatbot Layer The Chatbot Layer is a transformational subsystem, and will use a Main Program and Subroutines architectural style.

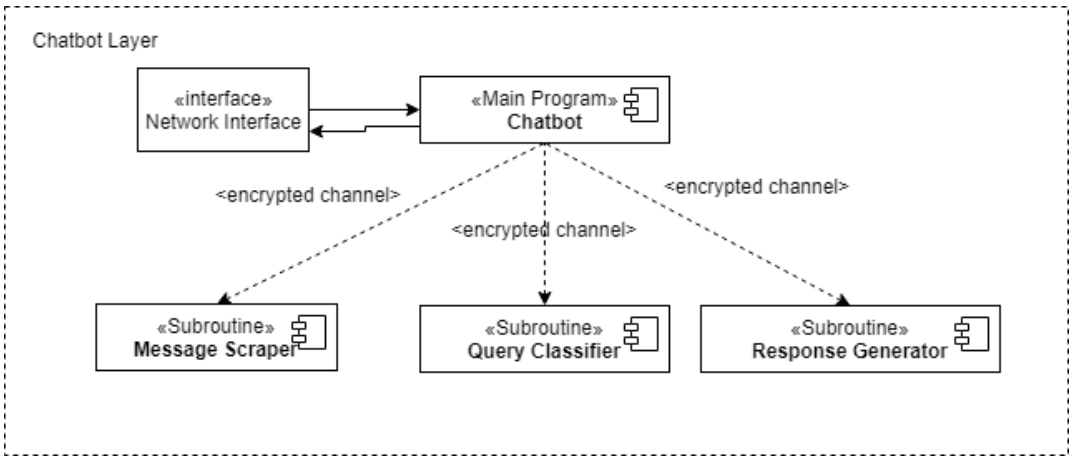

Figure 9: Architectural Style of the Chatbot Layer

The Training Layer This layer is a transformational subsystem and will also use a Main Program and Subroutines architectural style.

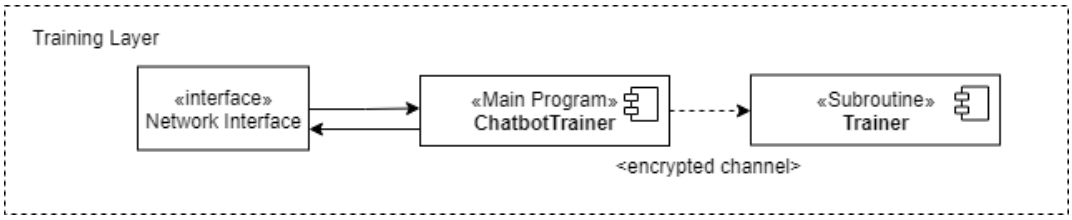

Figure 10: Architectural Style of the Training Layer

The ChatbotTrainer is a single container that has 4 instances: these are to service requests from the user interface (requests which are sent by the Administrator user), one to train the Message Scraper, one to train the Classifier, and one to train the Response Generator. It uses the Strategy Desgin Pattern to switch different training strategies/methods to suit each subsystem it services. This results in multiple instances of the same subsystem, but each using a different training method and data on a different chatbot subsystem.

The Persisitance Layer This layer is a clear candidate for the Persistence Framework architectural style, and it's architectural style is just that. Here is our solution:

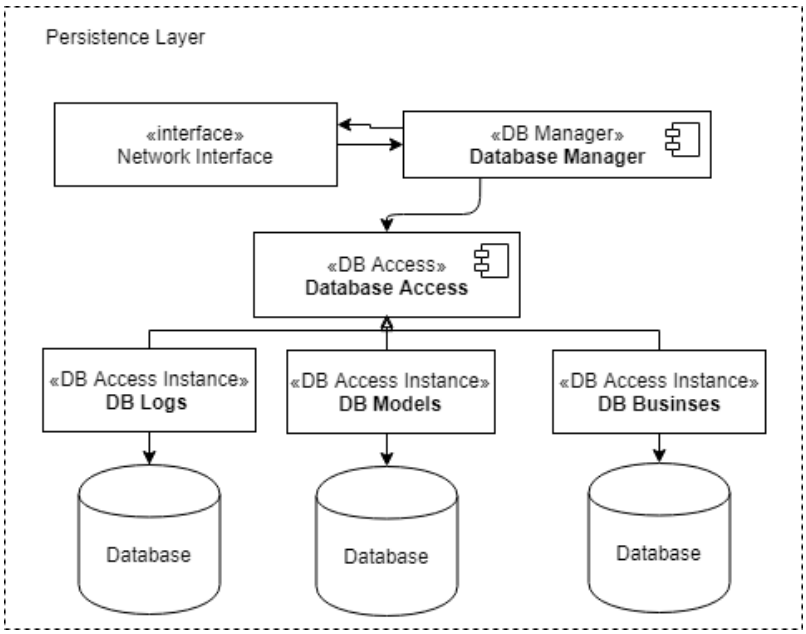

Figure 11: Architectural Style of the Persistence Layer

In accordance to the architectural tactics we have chosen to observe to achieve availability and security quality attributes and requirements, in particular the security tactic to seperate access and data of different sensitivities, we have made the decision to have three databases. The first is for logs and audit trails, the second for the data models what would be used for the chatbot subsystems, and the third for recording queries and responses– data that is concerned with the functional business processes.

5 References

[1] Information Regulator of SA, "Protection of personal information act," http://www.justice.gov.za/inforeg/docs/InfoRegSA-POPIA-act2013-004.pdf, 2013, accessed on 2019-05-21.

[2] IEEE, "Ieee recommended practice for software requirements specifications," IEEE Std. 830-1998, 1998, accessed on 2019-05-22.

[3] D. C. Kung, Object-Oriented Software Engineering An Agile Unified Methodology. McGraw-Hill, 2014.

[4] Auth0, "Architecture scenarios," https://auth0.com/docs/architecture-scenarios/spa-api/part-1, accessed on 2019-05-22.

[5] R. Clayton, "Speaking intelligently about "java vs node" performance"," https://rclayton.silvrback.com/speaking-intelligently-about-java-vs-node-performance, March 2016, accessed on 2019-05-23.

[6] R. K. Len Bass, Paul Clements, Software Architecture in Practice. Addison-Wesley, 2013.