Computer Science NEA – Chatbot

By Kazik Sobotnicki

Contents

[Analysis 3](#_Toc126681025)

[Problem identification 3](#_Toc126681026)

[Stakeholders 3](#_Toc126681027)

[Interview 4](#_Toc126681028)

[Interview: Tasnim 5](#_Toc126681029)

[Why should it have a computational solution 6](#_Toc126681030)

[Possible computational methods 6](#_Toc126681031)

[Decomposition 6](#_Toc126681032)

[Abstraction 7](#_Toc126681033)

[Research 8](#_Toc126681034)

[Chirpy Cardinal 8](#_Toc126681035)

[Kuki AI 10](#_Toc126681036)

[Features of my solution 13](#_Toc126681037)

[Limitations 13](#_Toc126681038)

[Developer requirements 14](#_Toc126681039)

[Hardware 14](#_Toc126681040)

[Software 14](#_Toc126681041)

[Stakeholder requirements 15](#_Toc126681042)

[Design 15](#_Toc126681043)

[Functionality 15](#_Toc126681044)

[Hardware 15](#_Toc126681045)

[Software 16](#_Toc126681046)

[Design 17](#_Toc126681047)

[Create Venv 17](#_Toc126681048)

[ML model 18](#_Toc126681049)

[Choosing model 18](#_Toc126681050)

[Implementing model 19](#_Toc126681051)

[User interface 20](#_Toc126681052)

[Create HTML web pages 21](#_Toc126681053)

[Use CSS styling for the web pages 21](#_Toc126681054)

[Create HTML forms for users to input text 21](#_Toc126681055)

[Link to python backend using Flask 22](#_Toc126681056)

[User interaction 23](#_Toc126681057)

[Text from user should be read by the model 23](#_Toc126681058)

[Response from model outputted to user 23](#_Toc126681059)

[Allow user to enter message using microphone 23](#_Toc126681060)

[Record Audio 23](#_Toc126681061)

[Decide on model to convert text to speech 24](#_Toc126681062)

[Fine-tune model 25](#_Toc126681063)

[Implement model 28](#_Toc126681064)

[Allow users to report messages 29](#_Toc126681065)

[Reported messages and the report reason should be logged 29](#_Toc126681066)

[Testing 29](#_Toc126681067)

[Development 30](#_Toc126681068)

[Development diagram 30](#_Toc126681069)

[Development practices 31](#_Toc126681070)

[Python version 31](#_Toc126681071)

[Git 31](#_Toc126681072)

[Type checking and Docstrings 32](#_Toc126681073)

[Flask algorithm 34](#_Toc126681074)

[Base structure 34](#_Toc126681075)

[HTML pages 37](#_Toc126681076)

[Home page – CSS grid 37](#_Toc126681077)

[Home page – adding elements 40](#_Toc126681078)

[Creating dark mode page 46](#_Toc126681079)

[Creating the messages page 51](#_Toc126681080)

# Analysis

## Problem identification

These days, people often go out and talk less, spending more time in doors and therefore meeting less people and engaging in conversation less with people. As a result of this, individual’s social skills, and their confidence in those, are worse off. This means that individuals create less meaningful connections, and conversate with less new people. This shows that people need a way to practice these skills in order to build up confidence. A way where there is no risk of embarrassment or awkwardness, and no one to make fun of you. People need an environment they are familiar with, to allow them to build up those skills, becoming more familiar with casual conversations, in particular with people they do not know.

This can also be an educational tool, to show users the uses and abilities of AI. But more simply, it can be a fun piece of software for the users to play around and experiment with.

## Stakeholders

The potential clients and users for this product are wide and varied. From people that want to develop their conversational skills, to people that just want someone to talk to when they are bored, many people can make use of this chatbot. A lot of people may also feel that they cannot talk to other real people and want to talk about private matters that they would want to keep themselves.

People with mental health issues or terrible illnesses may not feel like they can talk to others. Therefore, an online chatbot would allow them to talk about their problems and express themselves, in a safe environment where they should have none of the worries about expressing themselves that they usually would when conversating with real people. This should hopefully help them to become comfortably with expressing themselves and their problems and insecurities to real people.

## Interview

My interview questions are:

1. Have you ever used an online chatbot before?

* This question should show whether they have experience with chatbots, as previous experience may influence their opinions, either in a more positive or negative light.

1. If you have used online chatbots, or other used other language-centred applications that use AI, what are your opinions on how comprehensible, understandable, and natural current AIs are?

* This question is can only really be answered if they answered yes to question one, but this gives me a good view of the general opinion on current AI language generation, what could be improved and what has worked in the past. It gives me ideas for what techniques my bot should use, and what problems other AIs have struggled with.

1. What can make discussing personal things with people uncomfortable?

* This question can show what they may have to gain from the chatbot, and how it can help them. This will help give me a better idea of what the program will be used for, allowing me to have a more precise set of aims and requirements for the chatbot, so I can tailor the chatbot to the exact uses of the user

1. How often do you message people online or through text?

* This shows how comfortable they are with a messaging-based interface, hopefully backing up the previous evidence about how frequent online messaging is. It may also be an indicator for how much they could benefit from the chatbot.

1. Is it easier to talk over text than over phone calls or in person?

* This shows how much they can benefit from the chatbot and supports my previous knowledge that many people prefer to text over in person or on the phone conversation. This, combined with some of the other answers will give me a good idea on what some potential users are like, and what they would get out of using my chatbot.

1. What features (current features or ones you wish for) can make messaging easier

* This will give me a good idea of what features users would want to message with. For example, some users (particularly those with disabilities) may find speech-to-text (entering a message by speaking into the microphone) very useful. Others may put a lot of emphasis on privacy features or different themes.

1. If you wanted to talk about yourself, would you find it easier for the other person to lead the conversation, asking you questions, or if they just let you talk and react to what you say.

* This helps decide how the bot should communicate. Whether the bot should be starting off and controlling the conversation in a way, or if the users want to be able to enter things about any topic, and have the bot react to what I am saying. Each method would require a different approach to creating the artificial intelligence, so it would be useful to get a second opinion on which one any potential users find the best.

### Interview: Tasnim

1. Have you ever used an online chatbot before?

* Yes, a few years ago. Mostly to play around with.

1. If you have used online chatbots, or other used other language-centred applications that use AI, what are your opinions on how comprehensible, understandable, and natural current AIs are?

* Whilst some were understandable and comprehensible, none felt at all natural. It was easy to tell that I was talking to an AI and not an actual human.

1. What can make discussing personal things with people uncomfortable?

* When you are talking face to face, you can see immediate reactions to what you are saying, through tone of voice (of any response), facial expressions and vocal expressions, which might make you feel uncomfortable if those are not the reactions you intended (even if the person does care and is trying to help).

1. How often do you message people online or through text?

* Very regularly, every day.

1. Is it easier to talk over text than over phone calls or in person?

* It depends on the situation, but it can be easier to talk over phone calls as it is easier to get across the meaning of what you are saying.

1. What features (current features or ones you wish for) can make messaging easier

* Reactions, emojis, emotes and direct replies. A search bar to search through older conversations. Maybe customisable interface (such as light and dark mode)

1. If you wanted to talk about yourself, would you find it easier for the other person to lead the conversation, asking you questions, or if they just let you talk and react to what you say.

* I would find it easier to lead the conversation when talking about myself, as it allows me to talk about what I want to talk about, and not rely on hoping the other person is talking about something relevant to me, or that I want to discuss

#### Analysis

This interview shows that there would be interest in an AI chatbot, however that interest is strongly dependent on the quality of the chatbot. Poor, unintelligible chatbots would quickly lose the user’s interest.

The suggestion to have the user lead the conversation is interesting and would allow more interesting conversations. Conversations lead by the bot would have a risk of all being very similar, so this would help to add some variety and make the conversations feel more natural, and less repetitive. This would however mean the AI model would have to be more complex, as it would have to be able to understand a wider variety of topics and user inputs, more of which could be unexpected.

It was also pointed out that talking can feel more natural and take less effort than typing. This could mean an option of speech-to-text input may be useful, as it would allow the user to feel more like they are having a normal conversation, whilst still be understood by the bot.

## Why should it have a computational solution

Text conversations are becoming more common these days. Statistics have shown that young adults think text conversations are just as meaningful as phone calls. Furthermore, a not insignificant portion of the adult population prefers texts over all other forms of conversation. This shows that online messaging and texting are a very familiar way of communicating, one that people are comfortable with. It would therefore be best if the way for people to practice and build up their social skills should be through a text like interface, on a phone or computer. They would already know how to use this and would not need to spend any time becoming familiar with the interface, so they could spend all their time focusing on the conversations.

Furthermore, as I have already discussed, people do not always feel comfortable discussing freely with another person, so the anonymity of a computer screen, and the knowledge you are not talking to a real person can, in some ways, be comforting. The use of an AI allows this to be achieved, and on a large scale, where the number of users would make it near impossible to have dedicated staff or moderators watching over or even controlling the conversations. When purely using software for this, artificial intelligence is the only way to truly accomplish this complex natural language processing task.

I could just have a dataset of conversational responses and use basic comparison techniques to find the most similar one to the user’s input, and output that. However, these responses may feel unnatural as there is no way to have a response for every possible user input, so some of the responses may be off topic. Also, if the program is just selecting responses from a list, there is a risk that the same response will be printed twice, which would feel very unnatural to the user, and make the program feel more automated then human.

## Possible computational methods

### Decomposition

Decomposition is the process of breaking down the problem into smaller parts, each one can be solved individually. Then, when all the solutions to the smaller problems are combined, you have the solution to the main problem. This process makes the problem less daunting, and uncomplicates it. As my program will be very complex, being comprised of multiple features, each one having a host of problems, my program needs to be decomposed into many subroutines.

The project can be broken down into a set of smaller steps, such as:

1. Train an AI model for conversational intelligence
2. Build algorithms for interacting with the model
3. Have a simple, easily accessible interface
4. Take in user input, either through them typing it into a box or perhaps saying it into their microphone
5. Have the AI read the input and output a text response
6. Print out this response onto the interface

This should happen smoothly, and as quickly as possible so it could imitate a normal online conversation between real people in an as natural way as possible. As well as inappropriate responses, long response time and a clunky UI can take away from the immersion, and make the conversation feel less natural. This would take away from the familiarity and comfortability I am trying to make the users feel when using the program. This means that the program would no longer be able to achieve its aims of imitating a natural conversation and would not help them build up their confidence in real conversations.

Also, the AI model would only have to be trained once, and could then be stored in the software files, to be accessed by the program later on. This would decrease the size of the program and prevent long loading times when it is first run (loading times which would be heavily dependent on the speed of the user’s computer).

For the most part, decomposing my problem is not necessary, and I could just use minimal subroutines and just have long blocks of code, attempting to solve most problems at once. However, this makes the development process more complicated and time consuming. Trying to solve every problem at once also makes the code more convoluted and harder to understand, which would make it harder to go back and update the program.

### Abstraction

Once the problem has been decomposed into subroutines, it becomes easier to see what parts of the problem can be abstracted.

Abstraction is the process of removing unnecessary details to create a representation of reality. It can help to remove unnecessary complexity from the program, such as:

* As the conversational AI model would not be able to take images or emojis as input, only text input needs to be considered and coded for. The only way images could be taken as input is if another AI model was created, trained, to be used for image classification. However, this is a whole other complex program in and of itself, which is way beyond the scope of this program. I could ask users to give captions to images they want to send to the model, however, at that point the images are useless, as the user has to explain them, and the model only pays attention to the explanation.
* Any reactions to the AIs messages can be ignored, as these would be unlikely to give any extra information that the AI could not already gain from the user’s messages. The user should still be able to send reactions, but these are purely for the user, as the AI would not be able to understand these and there is nothing it could gain from it.

## Research

### Chirpy Cardinal

#### Link: https://stanfordnlp.github.io/chirpycardinal/

#### Overview:

Graphical user interface, text, application, chat or text message

Description automatically generatedChirpy Cardinal is a chat bot developed by a natural language processing research group at Stanford university, with the aim to research and advance conversational AI. It is seen very highly and won 2nd place in the Alexa prize. This is good, as it means it is a great example of the best of current conversational AI. It has a simple, easy to use interface, which although bare bones, allows the user to simply chat with the bot without needing to navigate through clutter.

Figure

Graphical user interface, text, application, chat or text message

Description automatically generatedIn terms of the AI and other algorithms behind it, it seems to control the conversation, asking the user questions and then reacting to their answers, continuing on the conversation. It is quite good at continuing the conversation, for instance, going from talking about cooking, to being vegetarian, to having pets.

It is generally relatively good at reacting to the user’s inputs, though this is partially because it is good at using less precise responses. For example, in Figure 2, its response to “I’m not vegetarian”, is “Oh okay, I’m actually vegetarian”. This is a clever response, as the “Oh, okay” gives the impression of a reaction to what I said, but could in fact be a perfectly understandable response whether the user said that they are vegetarian or not. It also has a pattern of never directly referencing what the user said. So when it asked for food reccomendations and I gave one, it didn’t directly include the name of that food in the response, it just said “That sounds really good”, which would make sense whatever I put in.

Figure

#### What am I going to implement in my own Chatbot?:

I like the clean, simple user interface, simply showing the text boxes and a bar to enter your response, with a button to submit the response. This simplifies the user experience, allowing them to easily interface with the product, therefore making them less stressed out by the interface and other things distracting them, and focusing entirely on the bot. I want to do something similar in mine, though I may add one or two other features, but I will try to make those as less overbearing and distracting as possible.

I also like the text message style of interface, with the send icon being common in messaging apps, and the text bubbles being familiar. This adds a familiarity to the program, as the users are already comfortable with the interface and how it works, so they feel more at home. This also gives the chat app a more casual feel, as the users likely use other messaging apps regularly, so using my chatbot will feel similar to just texting a friend.

Text

Description automatically generatedFurthermore, I like how casual the conversation is. Although it is led by the bot, who is the one asking questions, it feels much more like a casual conversation with a friend and less like an interrogation.

Moreover, I liked how it could occasionally refer back to something you said earlier in the conversation, for example in Figure 3, when it refers to the user mentioning school earlier and starts a conversation.

Figure

#### What am I not going to implement:

Although I like the simple, clutter-free interface Chirpy Cardinal uses, I would like to add one or two extra features. These could go a long way to help the user interact with the chatbot, and more varied ways the user and chatbot can communicate. These may also make it easier for the user to use, as they may be accessibility features such as speech-to-text.

Also, a lot of the bot’s conversations are similar, and I have found only three different starting conversation prompts once I entered my name, all asking me what food I like. I feel like a bit more freedom, and the user having more control over the conversation, should help them express themselves a bit more.

### Kuki AI

#### Link: https://www.kuki.ai/

#### Graphical user interface, text, application, chat or text message Description automatically generatedOverview:

Figure

Graphical user interface, text, application

Description automatically generatedKuki is a conversational AI that users can chat with over a text messaging like platform. Kuki has been given an avatar and has used this to expand to other media such as video games. This bot is also seen very highly, winning 5 Turing competitions, and is widely used with 25 million people chatting with it. Kuki’s interface has slightly more features to it, whilst still keeping the text message like appearance. It has features like mini games and quizzes, and a whole currency that can be earned by chatting with it and spent on gifts for it.

Figure

The quizzes can be personality quizzes or pop quizzes and are presented through the same text message-like format as the rest. The coins slowly build up over time, with the user getting small amounts when they talk with Kuki. Users also have the option to buy coins to supplement their funds. These coins can be used to purchase gifts like food items to give to Kuki. There are also other games, like small, simply games that work just with text and emojis, like tic-tac-toe and connect 4.

Figure

The chatbot itself also controls the flow of the conversation, asking questions for the user to answer, though unlike Chirpy cardinal, Kuki’s questions are a lot more open ended allowing for a wider range of user responses, which the bot is not too bad at doing. It has more direct references to what the user said, for example in Figure 4, where the user mentions liking bike riding and finding it refreshing, Kuki, comments on that, directly talking about bike riding and talking about it being refreshing in a different. It is also quite good at call backs, as shown in Figure 6.

This is a useful feature as it makes this conversation feel less random. It allows the bot to switch topics, but still not in a completely unexpected way. It also gives the feeling that the bot is truly listening and knowing what you say, rather than just giving off premade responses.

The bot also has further features, like video calls. These, whilst in theory make the bot more personable and human like, in reality often draw the user out, as reinforces to them that the bot is not real and can often lead to an uncanny valley-like effect, as the conversation may be realistic and human-like, but the bot is obviously not human. However, it can be useful as it allows users to talk using a microphone and hear responses as the bot says them, rather than only being able to write and read responses. This gives the user more ways to communicate with the bot and is more accessible.

#### What am I going to implement in my own Chatbot?:

Similar to Chirpy Cardinal, I like the familiar feeling of the interface. It draws the user in and makes them feel comfortable. It also means that they don’t have to spend time becoming familiar with the interface, as it is already similar to services that they use every day. However, I also like the visual flair added to separate it from other chat apps: the different chat bubbles and the avatar next to the bot’s messages.

I like how the bot directly references what the user says, either in direct responses or call backs to topics that were discussed earlier in the session. These makes the user feel heard, and make it think the bot is smarter as it is not just giving off generic responses that could work for anything the user inputs but is tailoring the user’s responses to exactly what was inputted. This also makes the user feel like the bot is genuinely listening and caring about what is said, instead of just sending out a response seemingly randomly picked from a list of responses.

#### What am I not going to implement:

Graphical user interface, application

Description automatically generatedI am not going to use the quizzes. I feel like they take away from the casual feel of the conversation that the bot has been good at creating, and instead feels more like the user is taking some internet quiz. It doesn’t help that the actual questions feel clunky. From the “Type START to begin the quiz”, to the “Please answer either A, B or C for question 1 or Quit to finish the quiz” (as shown in Figure 5), It feels less like a causal conversation and more like entering commands into a computer (which yes, is technically kind of what the user is doing anyway, but the bot is better at masking that at other times).

Figure

I am also not going to use the currency feature, shown in Figure 7. I don’t want to introduce a monetary or economic part to my program. I like the feeling of just having a casual conversation with the bot, either to pass time or build up conversational skills. I think introducing a monetary aspect to it can make the user feel more stressed on what to spend the money on, and to earn as much as possible. This becomes more like a game, and less like a conversation with a friend. Also, as you earn coins by chatting, users may feel pressured to talk with the bot as much as possible, perhaps giving shorter responses, with less thought put into them, so the responses will be quicker to enter, allowing them to earn more money. They may care less about having a good conversation, and more about gaining money, which takes away from the purpose of the bot.

Furthermore, I will not implement a video chat feature. This is for a few reasons. Firstly, it is a very large addition, which would end up taking way too much time to implement, especially compared to the algorithm behind the chatbot, which should be the main focus. But secondly, it introduces an uncanny valley like effect, drawing attention to the fact that the bot is not real, which would be easier to avoid with just the messaging conversation. However, I will take some parts from it. The ability for the user to interact with, and enter messages, using their voice is a good feature, both to add more ways to interact with the bot, but also to cater for people that may use more accessability features.

## Features of my solution

My solution will be a chatbot that you can communicate with through a text message like interface. It will use a conversational AI instead of a rule-based approach, as this generates more natural feeling messages. It will use a sequence-to-sequence model with a separate encoder (to analyse the users input and find the semantic information about the user’s input) and decoder (which takes in the semantic information from the user’s input, and returns the next word in the sequence, in order to generate the response).

It will use a messaging like interface, with a bar to enter a message, and text bubbles to show previous messages. It will also have a speech-to-text option, which will allow users to enter a response by talking into their microphone and use automated speech recognition to convert that into a text input.

The interface would have the option for the user to react to messages with emoji-style reaction bubbles, as well as report messages that they believe to be incorrect (to allow for better debugging and improvement)

If possible, I may include the option to have different personalities for the user to choose from, which would allow the user to choose a bot that would be best for them, as well as have some variety if they get bored of the bot they are talking to.

Finally, users will have the option to report messages, adding a reason onto the report. This allows for easier bug reporting and checking, making future improvements to the program easier.

### Limitations

Part of the limitations will stem from the conversational AI model. If this cannot generate very natural responses, the conversation will feel unnatural and less human-like. Also, having multiple pretrained models stored as part of the program will dramatically increase the disk space it uses. The option to have multiple personalities would be heavily dependent on finding a good enough dataset, preferably one with multiple different people’s conversations.

## Developer requirements

### Hardware

* **A fast enough computer.** This must be able to train machine learning algorithms, in a reasonable time, which requires a reasonably fast processor. The standard peripherals of a mouse, keyboard and monitor are also needed for using the software.
* **Microphone.** As the program will have a speech-to-text feature, A microphone is needed, although the standard microphone built into most computers and headphones is good enough.

### Software

* **Windows, Mac, or Linux operating system.** These are supported by both python and PyTorch (the python library used for creating and training machine learning models).
* **Python interpreter.** This is the programming language my chatbot will be developed in, as it is good for machine learning, and support many useful pieces of machine learning software, such as PyTorch.
* **Pip.** This is python’s package manager/installer, which will be used to install all the packages required to develop the chatbot. By default, this usually comes installed with python.
* **PyTorch.** This is a library for python that is the most widely used for developing and training machine learning algorithms, with support for many different types of algorithms. This is important for creating and training the machine learning model used for the chatbot. Whilst TensorFlow, the other popular machine learning library, is an option, I am going to use PyTorch as it is more pythonic. This means that common debugging tools, even simply print statements, work with PyTorch, whereas TensorFlow requires you to use its own debugging tool alongside normal python ones.
* **Transformers.** This library has pre-trained models for many different typed of transformers, and will be used to access the model, and fine-tune it.
* **NumPy.** This is a python library for doing complex mathematical operations and is also a pre-requisite for all of the other libraries used.
* **External Datasets.** Appropriate datasets of conversations are needed to train the model.
* **Flask.** A web application framework for python that can be used to create the web app for the chatbot. It allows the html files for the website to interact with the python code for the chatbot, sending data in between. This will allow users to enter messages into a html form, which will be processed by the python code, the output from which will be displayed on a new html page.
* **Any other possible packages.** This may not be all the necessary packages needed to develop the chatbot, more may be discovered during development, and they will need to be installed.
* **A web browser.** A web browser is needed to open the web app, as it will be run out of html files. This can be any mainstream web browser, such as Google Chrome or Firefox.

## Stakeholder requirements

### Design

* **Intuitive, simple interface.** An easy-to-understand interface should be used to allow the user to quickly understand how to use the program without the need for complex instructions. This interface should be similar to a messaging app, as this is a type of interface users are already familiar with.This reduces the barrier to entry for users to access and use the program.
* **Customisable display.** Users should be able to customise their display, to make it suitable for their needs and preferences. This should at least consist of light mode or dark mode options.

### Functionality

* **A working AI model.** The model should be able to produce understandable text, that makes sense in the wider context of the conversation currently going on between the user and the chatbot.
* **Reactions.** Users should be able to react to the AI’s messages with icons/emojis. This adds to the user’s interaction, and makes the conversation feel more organic, and more like a conversation the user may have on a chat application like Discord with a real person.
* **Reporting.** Users should be able to report messages the bot has sent that make little sense or are inappropriate for the context of the conversation. This could allow the users to add a message explaining why they reported it. This should allow for me to test and debug the program easier.
* **Speech to text.** Users should be able to enter messages into the model using their voice, with the program recording them through their microphone and converting the audio to text. This feature allows more ease-of-use, as users do not have to be directly sitting at their computer to talk to the chatbot. It is also better for those who find it hard to use a keyboard and would prefer to simply talk to the bot instead. Therefore, whilst I could only let the user enter text through typing into a text box, allowing the user to enter text through their microphone makes the program more accessible.

### Hardware

* **A decent computer.** A computer with decent specifications will be needed to run the chatbot, as the models will take some computing power to use. The chatbot technically can be run on lower powered computers but will be incredibly slow. The standard peripherals of a mouse, keyboard and monitor are also needed for using the software.
* **Microphone.** As the program will have a speech-to-text feature, A microphone is needed, although the standard microphone built into most computers and headphones is good enough.

### Software

* **Windows, Mac, or Linux operating system.** These are supported by both python and PyTorch (the python library used for creating and training machine learning models).
* **Python interpreter.** This is the programming language my chatbot will be developed in. It will need to be installed on the user’s computer to allow them to run python programs.
* **All packages used in the program.** All the packages used in the program will be listed in a requirements text file, allowing them to be easily downloaded using pip (pythons package manager and installer).
* **Pip.** This is python’s package manager/installer, which will be used to install all the packages listed in the requirements text file. By default, this usually comes installed with python.
* **A web browser.** A web browser is needed to open the web app, as it will be run out of html files. This can be any mainstream web browser, such as Google Chrome or Firefox.

# Design

Figure

## Create Venv

A virtual environment is a folder structure that allows the developer to use a lightweight, but isolated environment. This means that the current python version, plus all modules and libraries used in the development of the program will be stored with the program, independently of all other python modules installed globally.

I could just store all packages globally, however there are a number of issues with this approach, including dependency conflicts. When programming for this chatbot, I would be using a certain version of each of the packages, whilst the packages may be updated in this time – updates which could have conflicts with the versions I am currently using, potentially breaking the program if used. For example, if I were to install these updated libraries for use in another project and hadn’t used a virtual environment for the chatbot, these updated libraries (which would overwrite the old ones) could cause conflicts with the chatbot, which had been programmed for earlier versions of the package.

This is not only an issue that would affect me, but also other users who install the program, as these users might have other versions (older or newer) of the packages installed. Users will need to be able to reproduce the environment on their own computers, which will need to not conflict with any possible packages they already have installed.

There is also the possible case that users may be using a computer that does not have administrator privileges, and so won’t be able to install packages into the python directory. Therefore, these users should also make use of a Venv.

## ML model

### Choosing model

#### Why transformers?

An important part of creating a good chatbot is to choose the right machine learning model. One of the best performing types of machine learning models for natural language processing tasks is the Transformer, introduced in 2017.

Transformers are a type of neural network architecture that has seen wide use in the field of natural language processing, performing better much better than other types of machine learning models like recurrent neural networks. RNN’s loop through the language data inputted, finding relationships between data points that it can connect together to gain a better understanding of the input data.

Whilst I could create my own model using TensorFlow and Scikit-learn, Studies have shown that Transformers perform much better across a wide range of natural language tasks. Furthermore, models available from Hugging Face are pre-trained by the big companies and researchers that created them, a process that requires incredibly expensive hardware and trains them to a much better level than I ever could. Therefore, using transformer models from Hugging Face allows me to use models that are a lot more refined and accurate than a more basic model that may make use of a neural network. However, this becomes much more computationally expensive as the input sequences get longer, or more sequences are used, as the RNN tries to compress the sequence into a fixed length vector.

This is where the principle of attention is useful. This means that the model only pays attention to the parts of the input with the most information, therefore allowing the model to maximise the information gained from the input, whilst also being able to handle longer inputs

Transformers, which use the principle of attention, have shown to have much higher accuracy scores whilst having lower training costs than non-attention-based models across a number of language tasks, therefore showing it to be the best type of model for natural language processing tasks.

#### What transformer?

Whilst there are a number of different Transformer models, many of them are created and pre-trained for set purposes. For example, Facebook’s BART model is best used for summarisation, and Google’s T5 is best used for translation.

I have decided to use Meta’s BlenderBot, which is created specifically for Conversational AI, the task I need the model for. All iterations of BlenderBot have been shown to outperform other competing models – like Microsoft’s DialoGPT – in human evaluation tests. And the latest iteration, BlenderBot 3, performs even better than its predecessors. It is factually incorrect 47% less of the time more up to date (on topical questions) 82% of the time. However, as this is a developing field, it is still not perfect – with 1.2% of users reporting responses as incorrect or nonsensical, and 0.12% reporting messages as off-topic. Overall, whilst it is not perfect it is still by far the best transformer model for conversational intelligence. As it is created and trained specifically for conversational AI, it will produce better results in that specific task that I need it for, as opposed to using a more general-purpose model like Google’s BERT. Using a standard, pre-trained version of the model is a lot more practical for my purposes, as Meta states that the model requires a 32GB V100 graphics card to fine tune, which makes it impractical for me.

### Implementing model

Transformer models are publicly available through the Hugging Face library, which allows you to access the models and their tokenizers.

The BlenderBot model has different sizes, with the larger size giving higher quality outputs, but being more computationally expensive.

#### Tokenizer pseudocode

Tokenizers are required to convert the input data, in this case a string, into a type of data understandable by the model, in this case tensors. Hugging Face provides access to the BlenderBot tokenizer, which has been pre-trained by Meta. Meta provides a number of versions of BlenderBot, including a small 90M model, a medium sized 400M model and larger 1B, 3B and 9B models. I am going to use the 400M model as it strikes a balance between good performance and not being too slow. Whilst I could use a larger model, this would make the program run even slower, which would make the programme a lot harder to use for users with lower powered computers. So therefore, as I want to maximise the number of possible users that can use my program, I will use a smaller model.

Text

Description automatically generatedThe name of the model needs to be specified, and the tokenizer should be called, and used to encode the input message.

#### Model pseudocode

Text

Description automatically generatedThe model needs to be called and then is used to generate the ids for the reply tokens

#### Decoding pseudocode

The reply tokens then need to be decoded into a string of text that is readable for humans.



#### Overall pseudocode

This can be combined into one function which can be called whenever a message needs to be generated. As this is done as a reusable component, it means less code needs to be written, allowing for quicker development time and simpler looking code.

Text

Description automatically generated

## A picture containing graphical user interface Description automatically generatedUser interface

Figure

For my user interface, I am going to use multiple web pages, linked to a python backend using the Flask package. Figure 11 shows a mock up what the light mode version of the GUI will look like.

For my design, I took inspiration from popular chat apps like WhatsApp and Discord. This means that users will already be familiar with the user interface, therefore reducing the time it would take for them to get comfortable using the app. I have kept the interface somewhat basic, in order to make it less cluttered and easier to look at. Apps many elements in their UI, and many things happening on the screen at once, can make the user feel overwhelmed, and too distracted to use the app for its original function.

Some of the blank space left is also useful if I were to add more features to the app in later updates. For example, there is a lot of space in the character list for extra characters, so if I decided to update the app to add more characters the user can talk to, I can just add them on the end. If I had left very little space in the UI, then if I come to add more features, I may have to drastically rearrange the UI, making this choice not very practical.

I have used bright high contrast colours to easily differentiate the different elements in the UI, making everything clear and standing out, as well as creating a design that is more aesthetically pleasing than a more basic design. Whilst I could have just used black text and black borders on a plain white background, not only would this not have looked as good, but the more colourful design is also better for accessibility, as it makes everything stand out from the background, and from other parts of the UI. This makes it easier to see and understand for those with difficulties with sight. I tried to avoid using certain colours, like using both red and green together, to prevent issues for colour blind users. Whilst this does only cater for those with the most common types of colour blindness, I think this is the more practical as it caters for most colour-blind people whilst still allowing me to make a colourful design.

### Create HTML web pages

The web pages will be written in HTML. <continue here>

### Use CSS styling for the web pages

Icon

Description automatically generatedCSS will be used to style the web pages: adding colours and other appearance tweaks, as well as creating grids.

Figure

Icon

Description automatically generatedCSS grids are a useful way to arrange and order elements on a web page and can be used to make the design more consistent. I would make the page into a grid, can then create each element to fill a certain number of cells in the grid. An alternative to using CSS grids is to use a flexbox, however flexboxes only work for one dimension, either in rows or columns. As you can see in Figure 13, the flexbox is only ordering items in rows, with the rows wrapping around to the next line if needed. This is alot less organised than the grid shown in Figure 12, which has neatly ordered items by row and column, organising the elements in a much more structured way.

Figure

I will use external CSS, keeping all the CSS in one separate file as I will have a constant style across all pages. If I use internal CSS, I will have to make edits to the CSS in every HTML file, often making the same edits each time, whilst external CSS means I just have to make the edits once. It also prevents repetition, as due to keeping a relatively constant styling, using internal CSS would mean I would be adding the same CSS code to every HTML page. This not only means it is harder to test and refactor, but also increases development time and disk space used (although the reductions in disk space would be minimal due to the small size of CSS files in comparison to other parts of the program).

### Create HTML forms for users to input text

There will be two elements where the user can enter text. The first one is the message box, where users can enter text that the model will read and output a response to. The second one is the search box, where users can search through previous messages. Both of these require the HTML page to take the user input and send it to the python code that can take it as an input, and the HTML page will update to show the output.

### Link to python backend using Flask

Flask is a python package, which functions as a framework to connect html.

Flask organises the main Python script into separate functions, one for each HTML page, that it runs when the page is loaded. It can use the GET and POST methods to take the input from the HTML forms and store it as a variable in the function. This allows it to be run through the rest of the algorithm. The function can then return the next web page to be loaded and transfer variables from the function to the new HTML page. Flask then allows you to use the Jinja template engine (built into Flask) to output the variables onto the web page. Jinja also allows you to use basic programming tools like for loops in the HTML templates, allowing you, for example, to create a div for every variable in a list.

A popular alternative to Flask is the web framework Django, however Flask has a number of advantages over Django. Firstly, Flask is more lightweight, requiring a smaller codebase, speeding up development time and reducing disk usage. Flask also works better with external packages as it functions as a light, modular framework that creates the web framework and leaves everything else open to the developer. Flask also allows much easier access to the request object – used for access data from HTML forms, whereas Django requires the Request object to be explicitly passed around.

#### Base Flask pseudocode

Text

Description automatically generatedHere I have specified functions to be run when the user access the home page and the message page. The home page function will just display the home page for the web app. The message page function will take the inputted message from the html page, use a separate function (to be written later) to generate the reply and then renders the message page, sending the generated reply onto the page.

## User interaction

### Text from user should be read by the model

Flask’s request class provides a number of ways to request data for the flask app, for example ‘request.json’ parses JSON data, ‘request.args’ parses data from the URL query and ‘request.files’ parses in uploaded files. As my html files will contain input forms for the user to enter their message into, ‘request.form’ is the best way to do this. It requires you to specify the name of the form you are getting the data from, allowing multiple forms to be used. When using this, the type of requests used need to be put into the methods parameter in ‘@app.route’. ‘GET’ is the default method and is used to request the html web page with the form. When the user submits data into the form, this sends a ‘POST’ request, which the function handles.

The data is requested (using the ‘POST’ request) from the messages HTML form and stored as a variable to be used later in the program.

#### Form request pseudocodeA screenshot of a computer Description automatically generated with medium confidence

### Response from model outputted to user

The page should be updated to show the message the user sent, as well as the message the model outputted in message bubbles, moving the others up.

Flask uses the ‘render\_template’ method to render HTML pages stored in the templates folder, which is usually used as the return of a function for the original page. The ‘render\_template’ method supports inputting other variables alongside. These variables will be used in the template as Jinja variables, that can be used in the HTML code of the page

#### Returning html page pseudocodeText Description automatically generated

### Allow user to enter message using microphone

Users should have the option to submit a message to the user using a microphone. This should be done by having the python program record input from their microphone in the background whilst the user is looking at the html page, and then converting this input into a text string.

### Record Audio

When the user selects the microphone option, the python script should record audio and save it as a temporary file in the program folder. The recorded audio will have a set length of 5 seconds. Whilst in theory I could have the user choose to end the recording themselves, it would be much simpler to code for the recording to automatically stop after a set time, and it is unlikely any audio messages would be longer than this (I will test to make sure this is true).

I will use the Sound Device and Sound File libraries to record and save the audio. First, I declare the duration and sample rate of the recording (the sample rate will be 16kHz to match the training used for the Wav2Vec model which will transcribe the recordings). Then I use Sound Device’s record function to record the audio, specifying the sample rate and the use of two audio channels. Finally, I will use Sound File to save the audio file, using Sound File’s write function.

#### Recording audio pseudocodeGraphical user interface, text Description automatically generated

### Decide on model to convert text to speech

To convert the audio file to a readable string that can be used to generate a reply to, another transformer will be used. In this case, Meta’s Wav2Vec transformer, a state-of-the-art model for automatic speech recognition. The model uses self-supervised training, being trained on unlabelled data to allow it to achieve the best speech representation possible. It then uses supervised fine-tuning, using labelled data to get better at predicting certain words or phenomes. These approaches, particularly the self-supervised training, give it a massive advantage over other suitable models. Allowing the model to learn good speech representation allows it to achieve accurate results on a small amount of labelled data.

Chart, scatter chart

Description automatically generatedWhilst I could use a different model, if that model does not use the same self-supervised training, it would require a much larger amount of training data to reach similar accuracy levels. Whereas Wav2Vec reaches very good accuracy rates with a dataset 10 times smaller than what is usually used. As you can see from the graph above, the latest version of Wav2Vec (2.0) outperforms every competing transformer model, making it the best model for me to use.

Figure

### Fine-tune model

Fine-tuning a model is the process of training the model on a dataset specific to the task it will be used for, making it perform better at that task.

Whilst I could just use the default, pre-trained version of Wav2Vec 2.0 that is available on Hugging Face, it has been shown that fine-tuning a pre-trained Wave2Vec 2.0 model on even a very little amount of data yields a very small word error rate.

#### Dataset pseudocode

Fine-tuning a model requires a dataset to train the model on, and Hugging face provides suitable datasets for this purpose. I will use the TIMIT dataset, a set of recordings of 630 speakers, speaking in 8 different English dialects, all reading the same 10 sentences. This makes the dataset particularly useful for me as training it on a wide variety of English dialects means it should be more likely to understand a wide variety of English speakers. It has a version specifically for ASR (automatic speech recognition – the task the model will be used for) that I will use, that is especially good for this task as all the audio files are in the same 16kHz format, allowing me to use a more appropriate, more customised training algorithm.

As well as a standard train-test split of the dataset (which the dataset comes with already), I will also need to split the chunks of speech in the dataset into letters and use this to create a dictionary of the letters used and how many times they occur – using this as the vocabulary the transformer will use.

Text

Description automatically generated

Before this is done, the dataset will also need to be reformatted to get rid of the special characters found in the text, like punctuation. This is done as these special characters don’t have a specific sound linked to them, so the model would struggle to link them to the audio.



#### Creating data collator pseudocode

The first step to creating the training algorithm is to create a data collator. Data collators are used to combine many samples from a dataset into one batch, which can then be processed by the processor I created earlier and used to train the model. The data collator used to fine-tune Wav2Vec is different from other data collators in that it will need to dynamically pad the samples, ensuring they are padded to match the longest sample in their batch, instead of the overall longest sample. This is particularly necessary due to the differences in length of Wav2Vec’s input samples in comparison to its output samples.

To do this, I need to apply separate padding functions to the labels and the input values, as input speech and output speech are of different modalities.

The following class will be created for the data collator:

Text

Description automatically generated

This diagram shows the data collator class will have a number of attributes. Firstly, padding will specify to the collator whether or not the returned sequences should be padded. Max\_length sets the maximum length of the input values of the returned data, and max\_length\_labels sets the maximum length of the labels of the returned data. Finally, pad\_to\_multiple will pad the sequences to a multiple of the given value.

Figure

Text

Description automatically generatedThe call method will be defined as:

Firstly, the inputs and labels are split as, as stated above, they have to be of different length and will therefore be padded differently. I then use the processor’s pad method to pad first the inputs and then the labels, using their respective maximum length and padding arguments. As I am using PyTorch, I will also specify that the padding process should return PyTorch tensors.

#### Evaluation metrics pseudocode

During training, the model should be evaluated using the word error rate. Loading the word error rate will return a sequence of logit vectors, which will contain the log-odds for each word in the vocabulary dictionary I created earlier. As I want the most likely prediction, I will take the argmax of the logits. I will also decode the prediction ids and label ids. Text

Description automatically generated

#### Loading modelText Description automatically generated

Here I load the pre-trained Wav2Vec model from Hugging Face.

#### Training arguments pseudocode

Here I define all the parameters related to training the model.

Text

Description automatically generatedGroup by length groups samples of similar lengths together into one batch, speeding up training by reducing the number of padding tokens that are passed through the model. Per device train batch size specifies the size of the batch that will be run through each CPU core. Setting the evaluation strategy to ‘steps’ means that the evaluation is done at a set rate (set by the eval\_steps) parameter. Gradient checkpointing is used to save memory, which is useful due to the lower computing power I have access to. However, this does mean the backward pass will be slower. The use of fp16 mixed precision training reduces the also saves memory, with the trade-off of having a lower precision. This is a trade off I feel is necessary to ensure the algorithm can run smoothly on the lower computing power.

Any arguments not explicitly specified will use their default value, and the values I have used are simply estimates, and will be tweaked based on trial and error when developing the training algorithm

#### Creating trainer pseudocode

Now all of these aspects can be passed into a trainer, which can then be used to train the model. A screenshot of a computer

Description automatically generated with medium confidence

This trainer can then be run using trainer.train() to run the training algorithm on the model

### Implement model

Now that the model has been trained, it should be used to transcribe the audio message that has been stored as a .wav file.

Firstly, the audio file should be loaded, and for this I will use the Sound File library that I used earlier to save the recorded audio files.

Then I will use the fine-tuned processor to process the speech, getting the input values the model requires to transcribe the audio. Then the input values will be run through the model, extracting the logits of the predicted values. As I want the most likely prediction, I take the argmax of the logits. Finally, I use the processor to decode the predicted logits, returning a transcription that should be in readable English and match the audio fed into the model. Text

Description automatically generated

### Allow users to report messages

There will be an option next to every message from the bot to report the message. When clicked, a popup will appear with a HTML form in. There will be a text box for the user to input their reason for reporting the message.

The popup will be created using a Bootstrap modal. This requires a trigger – a button that the user must click to open the modal. The modal part contains a header, main content, and a footer (which will contain a button to close the modal). The main content will contain the HTML input form, which will have a textbox, allowing the user to enter a report reason, and a submit button.

### Reported messages and the report reason should be logged

The program will then take in the reported message, and find the previous message sent from the user. The reported message, the user’s previous message and the report reason will be stored in a CSV file.

This allows for easier error reporting, as I would simply need the CSV file to see how the model made an error and could use that information to change the model’s training if needed.

## Testing

# Development

## Development diagram

Figure

## Development practices

### Python version

This project is explicitly being written in Python version 3.9.9. Whilst when I started development there were more recent versions of Python available (Python 3.10), neither PyTorch nor TensorFlow support for that version yet, and wouldn’t function properly. More updated versions of python may work with it, and I can check once development has finished whether the program will still work on updated versions, but the program is written primarily for Python 3.9.9.

### Git

When developing my project, I am taking advantage of a service called git, a distributed version control system. It stores a history of all changes made to the program in a repository. This is useful as it allows me to go back and revert to a previous version of any of the files in the event of any unchangeable mistakes or corrupted files.

It also works as a backup, as any of these versions are downloadable from a separate server, so if a device used for development gets lost, stolen or broken, I can still access the code for the project by downloading the most recent version from the git repository. Whilst there are other backup options available, such as OneDrive, Google Drive or simply doing a physical backup using an external storage media, these don’t come with the version control feature – saving a new version of some of the files in the project would overwrite the previous versions, therefore not allowing me to go back and fix any mistakes that might be in the most recent version, or seeing other changes I made previously. Also, as git is a very common part of developer’s workflow, it has good support among most IDEs, meaning that it is simple and easy commit changes to a git repository, without having to open another file.

#### GitHub

To easily use git and access my repositories, I use a service called GitHub – an online hosting platform for git repositories. It allows me to view my repositories, as well as a commit history for each file, as shown in Figure 15. Background pattern

Description automatically generated

Figure

It also gives the ability to see exactly what edits each commit made to the file, highlighting added code in green and removed code in green. This allows me to quickly identify what changes may have caused a bug I am experiencing, as well as allow me to quickly revert small parts of the code without having to restore the entire file to a previous commit. An example of this is shown in Figure 16.

### Text Description automatically generatedType checking and Docstrings

Figure

Throughout the code, I use function annotations and docstrings to add clarity to my code and the functions I am writing.

Function annotations work as type hints, showing the data type a variable should be. I use annotations purely for function arguments, as this is where they are the most useful. Whilst I could use annotations for all variables, I don’t find there is much use in using them for mundane things like variable declarations, as it is obvious what data type the variable will be from what the variable is being defined as. On the other hand, with function arguments, it is often not obvious what the variable is meant to mean, and therefore what should be inputted when calling the function.

Another reason why these are useful is that they help prevent errors. In a lot of the functions I will write, inputting an argument with the wrong data type would lead to python throwing up an error, as I would be trying to do an operation designed for a different data type. This gets even worse when it is not an obvious error. For example, a function to add two numbers together would not throw an error in python if both the arguments (that in theory should be the numbers to be added) were strings, as adding two strings together is a valid operation in python (string concatenation). Therefore, the output would be wrong, but I would not find out about the bug until the function’s output was used in a different operation. Using annotations prevents this, as it makes it obvious exactly what type of variable this function is designed for, therefore giving more clarity on potential inputs and the use of the function.

However, these are only type hints are exactly that, only hints. As python is a dynamically typed language, simply using these types on their own doesn’t enforce type checking. Python will still let you use a different type for an argument, even when there is an annotation explicitly stating what type to use. Whilst there are libraries such as Mypy that can use type hints to enforce explicit type checking at compile-time, I haven’t made use of those. I feel the type hints are enough to tell me what to be used and using more libraries in development has downsides: more libraries potentially means more libraries for users to download, even if the type checking wouldn’t be of a massive use to them. Adding libraries means that the download process for my program might be more complex, and will definitely be longer, with the overall program (including any used libraries) being larger, increasing download time and the storage space the program will take up on users’ computers.

I also make use of docstrings. Doc strings are strings literals at the start of the function (straight after the definition) that can be used to create a description of the function, or to give advice on how it is to be used. Not only can this be used to give a deeper explanation, but it could, for example, be used to show an example of the type of output that should be expected from this function.

Below, you can see an example of annotations and docstrings in use, with a function I have written while developing the program. As you can see, there is an annotation to show that the argument ‘message’ is expected to be a string, and that the output of the program is expected Text

Description automatically generatedto be a flask.response object. The docstring provides a brief description of the use of the function, not being particularly long due to the simple nature of the function.

I don’t have to use docstrings or annotations, as comments could be used instead. However, comments usually serve a different use – they are mostly meant for people who read your code with the intent to extend it, whereas docstrings are usually used for people who intend to use your code, without needing to know how it works. As the project is not intended for other contributors, users would not need to know how exactly any specific function works, simply how they should be used.

Furthermore, annotations and docstrings are easy to access for functions without having to find the function’s definition. In python, all functions have a \_\_doc\_\_ method, which shows any possible docstring, and a \_\_annotations\_\_ method which shows any annotations. These mean that the annotations and docstrings can be accessed from other points in the file, other files or even the command line. Therefore, if I used comments, to see the information in the comment explaining how to use the function, I would have to search for the function definition itself, which is particularly tedious on larger projects like this which span multiple files. On the other hand, annotations and docstrings can be accessed from anywhere, therefore reducing development time, as less time needs to be spent searching for previously written explanations.

In fact, my chosen IDE (Visual Studio Code), and most others, shows the annotations and docstring contained in the function whenever any instance of the function is hovered over, as seen in the example below. This would not work if comments were used instead, therefore being another way where the use of annotations and docstrings can help clarity and speed up development time. An example of the annotations and docstrings being displayed for the previously shown ‘message’ function can be seen below.

Graphical user interface, text, application

Description automatically generated

## Flask algorithm

The main algorithm will be written with in python, using the flask library. Flask allows me to combine the html template pages together and use python algorithms and functionality to display data on the html pages. I use Flask’s requests class to request data from specific input boxes, referring to them with their id. I can then render a chosen HTML page, using a number of python variables which will turn them into Jinja variables.

Jinja turns HTML pages into templates, where Jinja code can be used to create basic algorithms. This means that the python code can be used to change the output of a html file. If this wasn’t a feature, every time new data was inputted into a form, and processed by a python code which would return an output, a new html file would be needed. This minimises the HTML files that need to be written, saving development time and file space, and also allows the program to be more easily expanded. The drawback is the Jinja is not a full programming language, and as such can only write basic pseudocode-style algorithms. This means that almost all actual programming needs to be done in the python scripts, with the Jinja code only deciding what HTML to output.

Flask also requires a set file structure, where the HTML and CSS files are kept in separate templates and static folders respectively. The static folder also contains any other assets required for the web pages, such as images.

### Base structureA screenshot of a computer Description automatically generated with medium confidence

Here I have used basic, boilerplate Flask code to create a working webapp that simply displays the main page of the app.

First, I import the Flask library, allowing me to use all parts of the library, and then explicitly imported specific methods. This means that I have access to all of the library, whilst still having a shorthand for certain regularly used methods.

The third line creates an instance of a flask app, which will serve as the webapp that the program will run as.

I then create a decorator on line 5, which wraps the function below. @app.route connects the endpoints of URLs to the code contained in the function it is wrapping. In this case, when the URL with the route (‘/’) is searched for, the main() function is called. As ‘/’ is the default endpoint of all URLs (even if you enter a URL without an endpoint, it will simply default to ‘/’), the main function is used to display the landing page – the first page users will see when running the app.

Lines 11 and 12 tell Flask to start the development server when the program is run. This is not intended for users to use, but instead provides a quick and easy way for the developer to see the changes new code has made to the webapp.

I have also added basic structure for the message and search functionalities to be added later on: Text

Description automatically generated

As with the message function, both of these functions use a decorator to specify what URL endpoint will call them. These URL endpoints are triggered by the HTML form actions, so when a user submits an input into the HTML form with the action ‘/message’, it goes to that URL endpoint, therefore calling the message() function.

#### TEST – message input

As shown by the diagram in Figure 14, this is testing a feature which also uses the messages.html and index.html files. The code written in those files, which is used for this feature, is explained below.

As shown in the code below, I have added some basic functionality to the message function, that will be run when the user submits an input into the message input form.

Text

Description automatically generated

This uses Flask’s request object to request the input from the form with the name ‘message-input’. It will then print it to the console, it will then render the message.html page. As the functionality to display messages on that page hasn’t been written yet, this should just display the default page. However, this still allows me to test the rendering functionality, and is necessary anyway as Flask requires these functions to have a return. This allows me to test three features at once: the ability of the HTML input form to take in user input, the ability of the Flask algorithm to request that input and the ability of the Flask algorithm to render a second HTML page.

Input – entering ‘This is a test message’ into the message box in the web app.

Expected output – the python script printing out ‘This is a test message’ into the console, and the message.html page loading in the browser window

A picture containing text, indoor

Description automatically generatedActual output –

Figure

As shown in Figure 17, I have received a 405 error – method not allowed, when I entered text into the message box and clicked enter.

With research, I have found that this was because I did not put the URL methods in the @app.route decorator. A GET request is a common HTML method that I would need to use to retrieve information from the Flask web server, and a POST request is often used to send data to the web server. When a user submits input, the POST method will send the data to the Flask web server, and the GET method will send the information to the python script. As you can see here in the HTML form, I have included the POST method, specifying that as how the data will be sent. 

Changes to make – include the GET and POST methods in the @app.route decorator, as shown hereText

Description automatically generated

This should allow the methods needed for the requested URL, therefore not showing the error, and instead printing the message input.

##### 2nd iteration

After making the changes, ‘This is a test message’ was outputted to the console, showing that the algorithm to request and process data from input forms works.

## HTML pages

### Home page – CSS grid

#### 1st iteration

Here I have created the CSS grid that shows the layout of the home page. This fits the structure of the mock-up I created in the design section.

Text

Description automatically generatedA screenshot of a computer

Description automatically generated with medium confidenceStyles.css: index.html:

I have created a div called grid-container, which contains all the items that will be contained in the grid. I have set the exact size of each row and column inside the grid-container element, using responsive units (vw and vh). This means that the left and right columns will take up 15% of the page, and therefore the middle column take up 70% of the page. The rows work in the same way, except with the top row taking up 9% of the page, the middle 81% and the bottom 10%. I then specify how many columns and rows each grid item should take up. For example, the sidebar panel is in column 1, and rows 1 to 3 and the settings panel is in column 1 and row 3. As I have set the width and height of the grid container to 100vw and 100vh respectively, the grid container should take up the entire page.

Whilst I could have used a different measurement for sizes, such as pixels, instead of vw and vh, the page would not have been responsive. Whilst I could fine-tune the pixel values so that it works on my computer screen, with a 1920x1080 resolution, anyone viewing it with a different resolution monitor, or anyone that may have simply resized their browser window, would see a jumbled up, poorly arranged page. Therefore, I have used vw and vh to ensure the elements in the page scale in accordance with the page’s dimensions.

Instead of using padding, I was originally going to make use of the ‘gap’ property of css grids. ‘gap’ allows me to set a universal spacing between grid elements giving a more visual seperation between parts of the page. Whilst I want this effect, I found that using the ‘gap’ property didn’t work, as it made the grid would be bigger than the page, meaning the user would have to scroll down to see the whole page. This happened because the grid element’s sizes added up to 100% of the page, so adding gaps between them made the grid size more than 100% of the page. I fixed this problem by using padding. Each of the grid elements has 5px of padding, meaning that there is still a visible separator between the grid elements, but the separator is inside the elements, and so doesn’t add to the size of the page.

Output: Graphical user interface

Description automatically generated

Figure

#### Text Description automatically generatedText Description automatically generated2nd iteration

Firstly, I have now added a background to the webpage, making use of the ‘background-image’ property to add a linear gradient the same as that in the mock-up.

I have also changed from using padding as a separator to using a border. Whilst I could have stayed with padding, as it fit the purpose it was needed for before, as shown above, using a border brings a number of advantages. Firstly, the border property gives you more specific control, allowing me to set borders only on certain sides of the page elements. This has allowed me to only give a border to the inside edges of elements, leaving the edges of the page free of borders. It is also easier to give borders a colour independent of the rest of the page, unlike with padding.

Output: A picture containing shape

Description automatically generated

### Home page – adding elementsShape, rectangle Description automatically generated

Figure

I have added a message input bar along the bottom using the following HTML code: A screenshot of a computer screen

Description automatically generated with medium confidence

Here I create a HTML form with the ‘/message’ action, so it will send the input to the /message page, which is read by the Flask code in app.py and is expected to display a page with the /message extension. The input form will send the input using the post method, and the form consists of three elements – a text box, a submit button and a microphone button. The submit button contains an image, and CSS is used to make the button only take up the image. I have also used alternative text for the image and will do this with every image used. This helps with accessibility, as it means that for those with screen readers, their screen readers will be able to describe the image, therefore meaning they won’t miss out on any information they may need from the image. The microphone is also a submit button, but this time it will not send any data to the flask app instead just triggering one of the functions to display a page. The button contains an image, and again CSS has been used to ensure the button appears to only be the image. Again, for accessibility reasons, the image makes use of alternative text.

CSS:

Firstly, I added the ‘display: flex’ property to the main div, turning it into a flex container, with the message input form as a flexbox inside. I use the following CSS to style the message form:

Text

Description automatically generatedAs I have specified the flex direction as row, the form flex box will be a row inside the main div. As margin-top is set to auto, the rows will be arranged from bottom up, therefore meaning that the form will be a row on the bottom of the page. I have then set the message-form div to be a grid container. I have used relative units (vw and vh) to create a grid with 3 columns and 1 row, each with set (relative) sizes. I use the ‘grid-auto-flow: column’ property to ensure that the grid only takes up one row. This works because this property makes items flow across a single column, instead of going down in rows. I specify ‘text-align: center’ to ensure that the message input and submit buttons are in the centre of their grid boxes. I also used the gap property to add a small gap (again using relative units) between the grid boxes.

Text

Description automatically generatedThe following CSS is for the text input and submit button: Text

Description automatically generated

The text box is given a border, with the radius creating curved corners, giving a smoother look. It is given a separate background colour, to allow it to be set out from the rest of the page, and easily visible, with good contrast. This allows those with accessability concerns to still be able to easily separate the message box, and other elements on the page, from each other and the rest of the page. I have used relative units to set the size of the font used, meaning that the font will shrink when the page shrinks, therefore meaning that the element should stay (somewhat) consistantly sized in respect to the rest of the page

For the submit button, I have set the background of the button to be transparent, meaning that the only part of the button that can be seen is the message icon. This approach isn’t perfect, as the button itself is still a square, the user could click slightly to the side of the icon and still be pressing the button, however I couldn’t find a better way to do it.

I have also set the height of the icon image itself to 100%, so it should completely fit the dimensions of the button. Without doing that, the image would revert to its original dimensions with no relation to the overall size of the button, which would make it look hilariously oversized, taking up a large portion of the page.

The submit button isn’t strictly necessary, as the user can submit any text entered into the text box simply by pressing the enter key, however the submit icon gives a more explicit way to submit text, and makes it clear the function of the message bar.

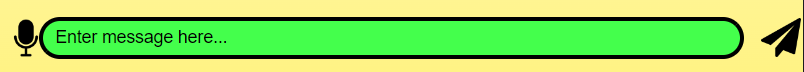
Text

Description automatically generatedI have used the following CSS to style the microphone button:

Similarly to the submit button, I made the background transparent and removed the border so that it appears as if the button is only the microphone icon, and not in fact a box surrounding it. I have also removed any margins and, this is to allow the icon to be centred, taking up the same vertical space as the message input bar and submit button. This hasn’t worked perfectly, as you can see in the image below. When the page is shrunk, the submit button stays centred but the microphone button appears to be a bit too high up. However I couldn’t find a way to fix this, and I feel as though users drastically resizing the height of the page is an rare enough scenario that this shouldn’t be significantly noticed by users.

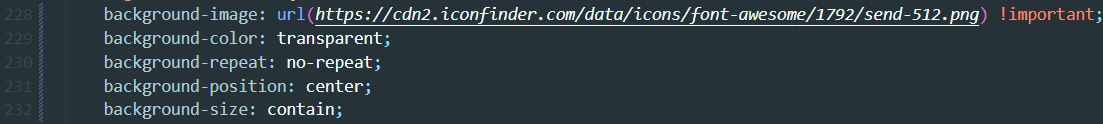


Figure

However, this approach wasn’t working responsively. When I resized the page, shrinking it, the spacing between the form elements became inconsistent, with the microphone icon having basically no seperation with the message bar, whilst the send icon still has a strict seperation, therefore giving it no margin to the edge of the page. 

Figure

To fix this, I changed how I added the microphone and submit icons. Instead of using the <img> tag inside the buttons, I left the content of the buttons empty and added the icons as background images. As you can see from the CSS below, I added the images as background images. I then set the background colour to be transparent, as otherwise a white box the size of the button will appear behind the image. I also ensure that the image is centred inside the button, and strictly contained inside of the dimensions of the button, ensuring it doesn’t expand out.



Text

Description automatically generatedI have also added a message search bar using the following HTML code: .

Text

Description automatically generatedThe form has the ‘/search’ action and the post method, so it will use that method to send the input to the Flask code, which is expected to display a page on the /search extension. The input form contains two elements – a text input and a Datalist. The text input works in the same way to the message input above, except that as there is no submit button. Whilst I could have added a submit button for the same reason that I added a submit button for the message bar (added clarity), I felt like that would have added unnecessary clutter to the header, and I added a (not clickable) search icon inside the search bar which should make the function clear enough. I used the following CSS to create the search bar:

I added a search icon to the left of the search bar by adding it as a background image. I could then position it 10px from the left edge of the search bar, whilst centring it vertically. I then specify the property ‘background-repeat: no-repeat’ as otherwise the background image would repeat as many times as it can fit inside the element.

Similarly, to the message input, I use a different colour to the page background, as well as a rounded border, to separate it from the rest of the page, partially for aesthetic reasons and partially for accessibility reasons.

I added a sidebar to the left-hand-side of the page, which is set out so that it could show a list of other chatbots that the user could communicate with. Whilst now, there will only be one, setting the page out like this means that in the future adding selection boxes for other chatbots.

To do this, I used the following HTML code: Graphical user interface, text

Description automatically generated

This creates a div, which itself contains an image (the chatbot icon) and the chatbot’s name (currently just chatbot).

Text

Description automatically generatedI then styled it with the following CSS code: Text

Description automatically generated

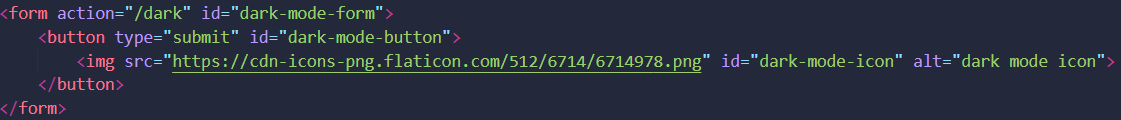
A picture containing logo

Description automatically generatedI added a border to separate that chatbot select section from the rest of the page, as well as any other options that may be added in the future. I set the chatbot-select div to take up the entire width of the sidebar using the property ‘width: 100%’, as otherwise there is a small gap between the chatbot-select div and the border of the sidebar, as seen in Figure 22. I then set the div to be a grid container, with the chatbot icon and name being grid boxes inside. The container consists of two columns and one row, with their dimensions relative to the size of the container. I have also given it a different background colour, allowing it to stand out from the rest of the page and showing that it is the chatbot that has been ‘selected’.

Figure

I set the icon to take up 80% of the width of its grid box, which I felt was a good size that didn’t make the chatbot take up too much space height-wise. This is because, as I have set the height of the image to auto, allowing it to scale proportionally to the width to prevent image distortion, using a larger width would have meant the height would have been bigger as well, taking up too much of the page. I use the ‘margin: auto’ property to centre the icon inside its grid box, however for this to work I also need to include the ‘display: block’ property, which converts the image (which is an inline element by default) to a block-level element. Also, a width needs to have been set so that the left and right margins can take the remaining space and auto-align themselves.

For the chatbot’s name, I use a relative font size, the same as in the message input box. I set the bottom and top margins to auto to centre it vertically, and then align the text so that it starts from the left edge of its grid box.

I have also added a settings panel, which for now just includes a dark mode icon, using the following HTML code: 

The settings div contains a form that uses the ‘/dark’ action, however, it doesn’t use any method to send data as no data is sent. It simply triggers a function in the flask program which should display a different page, without the need to send any data. The form only contains a single element – a submit button that contains an image – the dark mode icon.

Text

Description automatically generatedI use the following CSS to style the button:

The settings div becomes a flex container, which uses the ‘justify-content: left’ property to ensure that the flex boxes go across the container from left to right, and ‘align-items: center’ to ensure that the items are vertically centred. As there is only one element, it is situated on the left side of the container.

Text

Description automatically generatedAs with the message submit button, I set the background to be transparent and remove the border, so it appears that the button is simply just the image. I then give the image a height relative to the size of the page and set the width to auto to scale the image properly.

### Creating dark mode page

To create the dark mode page, I will use a separate CSS style sheet, that is largely similar to the original style sheet, but with different colours. On the whole, the colours will simply bee the opposite of what they currently are, e.g., switching black text with white text.

There are other possible approaches, such as using different HTML files, in which are different class names that would link to other styles on the original CSS style sheet. Or I could have simply turned the class names into Jinja variables, and sent different variables with flask, depending on if the user was in dark mode or light mode.

However, these approaches have downsides. The first approach would involve making more files, and files which are larger in size, in comparison with a second style sheet. This is due to the simple fact that there are more HTML files, and they are larger in size, with the messages.html file currently at 15KB for example, whilst the CSS stylesheet is only at 8KB. Therefore, adding more HTML files would increase the file size for the user more than adding an extra stylesheet. Based off this, turning class names into Jinja variables would seem like a smarter solution. However, this becomes impractical as the number of classes and ids used increases and would also make the python file larger.

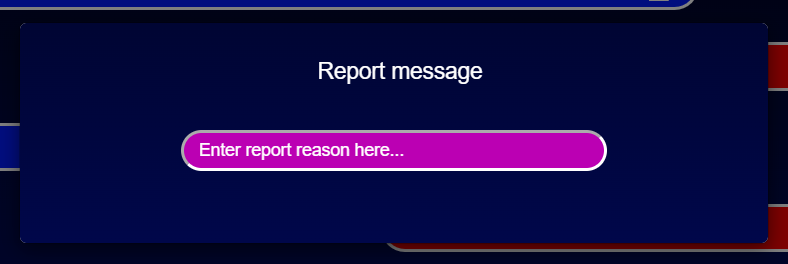
Before creating the dark mode stylesheet, I ensured all images were added with CSS properties, instead of in the HTML files using the <img> tag. This is because the icons will have to be changed due to them being black. I will change them to inverted versions, with the black outlines to be changed to white outlines. Therefore, as the only thing that changed when the user changes from light to dark mode is the stylesheet, all the images will need to be stored in the CSS stylesheet. For example, the colour mode switch icon shown below was stored Shape

Description automatically generatedin the <img> tag inside the button in both the HTML files. This was changed to be a background image in CSS, using the same properties as I have shown before for adding background images, including setting a transparent background colour. I also set the width and height to be 100%, to ensure that the button still fits inside the settings div (the bottom of the sidebar).

Graphical user interface

Description automatically generatedA picture containing rectangle

Description automatically generatedI then created a new CSS file, dark-styles.css, which will contain the stylesheet used for the dark mode version of the page. The stylesheet is the same as the previous one, except the colours are flipped and the images are changed.



I then added the functionality for the user to switch between light and dark mode. To do this, I added a function in the python script which will render the current page with the correct theme, as shown below. Text

Description automatically generated

Separately, I declared variables for the current theme and current page, which will be set to ‘light’ and ‘index.html’ by default respectively. The function above checks what the current theme is when the button is pressed, and switches to the other theme (e.g. if the current theme is ‘dark’, it changes the current theme to ‘light’). It also creates a variable to store what stylesheet to use, storing the file path of either the light or dark CSS stylesheets. This means that it is easy to use them when rendering the page. I then render the current page (as the current page will be set to a file name (e.g. ‘index.html’)), sending the stylesheet variable as a Jinja variable called stylesheet. Below is how it would be used in the HTML file, with the curly brackets around it signifying that it is a Jinja variable and not referencing anything else or to be interpreted as a string for where to find the stylesheet.

The ability to use Jinja for templates is one of the main advantages of Flask, as it allows me to easily edit static HTML files, based of changes in the python script, simply by sending variables across.

As the python function is inside the app.route decorator that listens for the ‘/theme’ URL endpoint, the function will only be called once the HTML form with the action ‘/theme’ submits. As, unlike the message input form, no data is actually sent from the HTML form, I do not use Flask’s requests module to request any data from the form, the only use of the form is to trigger the function.

#### TEST – theme change

As shown by the diagram in Figure 14, this is testing a feature which also uses the messages.html and index.html files. The code written in those files, which is used for this feature, is explained below.

Here I will test if the previously explained code works. Initially I will be on the index page and be using the light theme (as these are the default settings when the app is first opened). I will then click on the button in the bottom left hand corner of the page to change the theme. This should trigger the theme switcher function. This should see that the current theme is the light theme, and change the current theme variable to ‘dark’, and set the default stylesheet to ‘static/dark-styles.css’. Then it should render the index.html page, with the Jinja variable stylesheet also set to ‘static/dark-styles.css’. This means that link tag for the stylesheet would be <link rel=”stylesheet” href=‘static/dark-styles.css’>. HTML should then use this to set the dark-styles CSS file as the stylesheet, therefore meaning that the page will be rendered with dark styles (e.g. with white text and dark backgrounds).

Input – Loading the index page, then clicking the theme switcher button (the moon in the bottom left-hand corner of the page) on the index page.

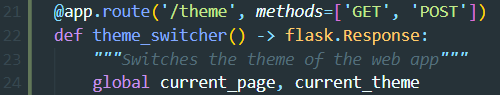
Expected output – the index page but using the dark stylesheet.

Actual output – an Unbound Local Error when attempting to load the input page. Graphical user interface, text, application, Word, website

Description automatically generated

Text

Description automatically generatedJinja returned an Unbound local error when I tried to run the web app, saying that I had referenced the variable ‘current\_theme’ before assignment. This is due to the fact that it was defined outside the function, on its own at the top of the python script, as you can see below. As it was defined outside a function, it is a global variable. This is different to local variables, which can only be accessed and used inside the function they are defined in. Whilst I could have made current\_page and current\_theme local variables, declaring them inside the theme\_switcher function, there are a few advantages to using global variables in this scenario. First of all, these are variables which may be useful for other functions, therefore using global variables means that I do not have to repeat the variable definitions in the other functions, and that the variables will remain consistent throughout the program. The other advantage is that local variable are lost once the function has finished running and returned a value, as opposed to global variables which only reset once the program is stopped. This means that using local variables would make it harder for me to keep track of the current page the user is on, as it would reset to the default option whenever the they click on the theme switcher, even if they are not on the original index page.

To fix the unbound local error, I simply have to import the global variables into the function using the global keyword. This is shown in the code below.

##### 2nd iteration

Input – loading the index page.

Expected output – index page to load in light mode.

Actual output – CSS not loading at all.

Icon

Description automatically generated

The CSS did not load at all when I opened the webapp on the index page. This is because in the HTML files for the index and messages pages, the location of the CSS files is in a Jinja variable called stylesheet. This is sent to the html files using an argument in the render\_template function that is returned at the end of the main, message and theme switcher function. However, this is a local variable that it only created and used inside the theme switcher function and is not used in the main and message functions. Main and message function render the page, they do not include an argument for the stylesheet variable, so when the index page is initially rendered, there is nothing in the stylesheet variable.

Text

Description automatically generatedTo fix this, the variable for the stylesheet needs to be a global variable, defined at the top of the program in order to allow it to be accessed by all functions. I have defined the stylesheet variable outside any function and imported the stylesheet variable into the main and message functions with the global argument.

Also, another possible future issue is that the current page variable currently does not change. Therefore, when the theme switcher function is run due to the theme switcher button being clicked, the index page will always load (hopefully in dark mode), even if the user was on the message page before. To fix this, the current\_page variable will also be imported into the message function, where it will be changed to ‘message.html’ this means that if the theme switcher button is clicked whilst on the message page, the user won’t be sent back to the home page.

#### 3rd iteration

Test 1:

Input – Loading the index page, then clicking the theme switcher button (the moon in the bottom left-hand corner of the page) on the index page.

Expected output – the index page should load in light mode, and then when the theme switcher button is clicked, the page should change to dark mode.

Actual output – the page now initially loads in light mode, just like it did before. Then, once the theme switcher button is clicked, the page changes to dark mode.

This test shows that the theme switcher works on the index page, changing the styling when used. This shows that the theme switcher button and corresponding function works, and that the use of the Jinja variable to change the stylesheet for the index HTML page works.

Test 2:

Input – entering a test message to access the message page, and then clicking the theme switcher button

Expected output – the message page should load in light mode, and then when the theme switcher button is clicked, the page should change to dark mode.

Actual output – the message page initially loads in light mode, the same theme as the index page. Then when the theme switcher button is clicked, the message page loads with dark stylings.

This test shows that the use of the Jinja variable to change the stylesheet for the message HTML page works.

Test 3:

Input – opening the index page, turning it to dark mode by clicking on the theme switcher button and then entering a test message.

Expected output – the index page should initially load in light mode and then change to dark mode when theme switcher button is clicked (this has already been tested). Then when the message page is switched to, it should also be in dark mode as the styling should be consistent.

Actual output – the index page loads in light mode, when the theme switcher button is clicked the theme changes to dark mode. Then when a test message is entered into the message box, the message page is opened with the dark mode stylings.

This test shows that the global variables for the stylesheet, current theme and current page work correctly, updating when meant to in the theme switcher and message function, and then applying the new values when the functions are run again.

For this to work, the current page variable is initially set to ‘index.html’, the current theme variable is initially set to ‘light’ and the stylesheet variable is initially set to ‘static/light-styles.css’. These are global variables that are defined at the top of the program, before any function definition. The index page is then loaded by default when the page is initially opened using the main function. This function imports the stylesheet variable, and then includes it as an argument in the render\_template function it returns. This sends it to the index HTML page, storing the string inside a Jinja variable called stylesheet. This then sets the reference of the stylesheet (the href part of the stylesheet tag) to the contents of the stylesheet variable - ‘static/light-styles.css’, therefore the index page will load with the light CSS stylesheet.

Then, when the theme switcher function is run, the function first checks what the current theme is. As the theme has not been changed yet, it is still at the default value of light. Therefore it will be changed to ‘dark’ and the stylesheet will be set to ‘static/dark-styles.css’. Then the theme switcher function ends with the render\_template function. The current page is still the default value of ‘index.html’, and so the index page is rendered, with the newly edited stylesheet variable inputted as an argument. This is sent across to the index HTML page, which then renders the page at the address in the stylesheet variable.

As stylesheet is a globale variable, editing it in one function preserves the edits for use in all other functions. So when the message function is used to render the message page, the new stylesheet is sent across as the reference to be used for the CSS stylesheet.

### Creating the messages page

The messages page will be the same as the index page, except containing the messages inside the main div. It will use a container div, which will contain individual divs for each message.

Text

Description automatically generated

Here I have created a div inside the main div called ‘messages-container’. Inside it is currently a div for a message from the bot, and a div for a message from the user. These will be user generated in the future, but to test the HTML and CSS, I am currently using hardcoded versions. This currently just creates an empty, bordered box above the message form, as shown in Figure 23.

Shape, rectangle

Description automatically generated

Figure

#### Messages boxes

##### 1st iteration

Each message written by either the user or the AI should be shown on the screen in the form of message bubbles, as shown in the design mock-up in Figure 9. This mock-up shows rows of message bubbles, with the most recent at the bottom and the least recent at the top. There will be a maximum number of text boxes shown, which will be until the screen is filled – 4 user messages and 5 chatbot messages. Whilst it may be a useful feature to allow the user to infinitely scroll up through past messages, I found that this didn’t work with method I was using to display the messages. I am using a CSS grid, which doesn’t allow for a grid of infinite size, where the size of the list of items is unknown. Furthermore, the need for an infinite scroll through messages is negated by the search feature, which allows users to search through their past messages. If anything, this is a better way of doing it, as users can quickly find a specific message, rather than having to scroll for a while to find the message they are looking for.

For the HTML side, I have written 9 divs like below for the message boxes, 4 with the class user-message-box and 5 with the class ai-message-box. Text

Description automatically generated

All of these boxes are in a div called messages-container, which will act as a grid container. Using the CSS below, I have turned messages-container into a grid with 1 column and 9 rows, each with a height of Text

Description automatically generated11%. Unlike in a lot of the rest of the CSS used throughout my program, I used percentage units instead of vh and vw units. This is because vh and vw are relative to the entire size of the viewport (the area of the page viewable to the user), whereas percentage units are relative to the size of the div. Using vh would mean that the boxes would extend below the input form, not just staying in the container div. Also, as the size of the container div is already relative to the size of the viewport, the design will still be responsive.

Table

Description automatically generated with medium confidenceThis HTML and CSS produces the page shown below. In Figure 24, you can see that the message boxes are equally divided up in terms of space inside the container div.

Figure

##### Graphical user interface Description automatically generated2nd iteration

Figure

Text

Description automatically generatedHere I have turned the grid boxes into actual message bubbles. The following is the HTML for one of the AI’s message bubbles:

Text

Description automatically generatedText

Description automatically generatedThe AI’s message bubbles are contained inside a container div, which contains both the message bubbles and the chatbot icon, to show that the message is from the chatbot. To create this, I made the ai-message-container div a grid item, and then gave it a width and height of 60%, to ensure the message bubbles are not too big, and to give appropriate spacing between each one. The CSS below moves the container to the bottom left corner of the grid space, ensuring it is separated from the user’s messages (as it is aligned to the other side of the page), and again giving it an appropriate gap to the message above. I then turn the container into a grid container with one row and two columns, sized as 7% of the container and 93% respectively. As before, whilst these are not vh or vw units, they will still be responsive to the overall page size due to the responsive nature of the container. In grid column 1 is the chatbot icon. Whilst originally, I tried to use a div which would contain and <img> tag for that chatbot icon, I couldn’t get this version to work. The image kept being ludicrously big, far outsizing the grid square supposedly available for it. Instead, I simply created a div for the image, and set the background of that div to the chatbot icon, as follows:

Using line 77, I specify that the image should be contained inside the div, not overflowing outside of it, and centre it using line 76.

A screenshot of a computer

Description automatically generated with medium confidenceThe following is the HTML for one of the user message bubbles:

The user’s message boxes are bare grid items, with being inside a container, as there is only one element to the box – the message bubble itself, rather than a secondary element like the chatbot’s icon, that would have to be put alongside it using a grid. As with the AI message box, I set the height and width to 60% for the same reason and used the below CSSText

Description automatically generated to align it to the bottom right corner of the grid space, further spacing it from the AI message boxes. Whilst this extra degree of seperation isn’t strictly necessary, it is another way to visually set the user’s and AI’s message boxes apart. It is particularly useful for users with smaller screens, as, due to the responsive design practices prevalent through the design of the web page, shrinking the window still has the different boxes physically separated from each other. Another way I separated the different message bubbles is by using different colours. Using vibrant, visually distinctive colours easily allows users to separate the two types of message bubbles from each other and is particularly useful for users with poorer vision.

Text

Description automatically generatedI use the below CSS to position the message text in the vertical centre of the boxes, accompanied by the ‘position: relative’ attribute, used by all message bubbles. Explicitly using the ‘-ms-transform’ property ensures that the styling will work the same on all browsers a user may access the web page from, including internet explorer. Ensuring compatibility for all browsers like this lowers the barrier to entry for using my web app, therefore ensuring that the greatest number of users possible will be able to use it and all see a consistent styling.

Finally, I use the following CSS for all message boxes, whereas the number of the box increases, the grid row decreases. This means that the boxes will be ordered from bottom up.

#### Bootstrap modal

I now want to add in the report icon for the AI messages. This will be done using a bootstrap modal, which is a popup window displayed on top of the page. Bootstrap is a third-party framework that includes template for HTML and CSS design. There are other options to create pop-up boxes, like jQuery dialogs, but I found that using Bootstrap modals is a much simpler way of doing it and requires much less code to be written as you can just make use of the pre-written CSS Bootstrap provides.

Html: Text

Description automatically generated

This modal is not created using my own CSS and is instead created using Bootstrap. Therefore, to allow me to make use of Bootstrap elements, I added the Bootstrap stylesheet, adding it above my own stylesheet, as shown below. Adding it above my own stylesheet means that my stylesheet has priority, therefore meaning any CSS I add to style Bootstrap elements will overwright Bootstrap’s own styling. Bootstrap’s CSS is used by referencing Bootrap’s classes. An example of this is shown in line 42 when the Bootstrap class ‘modal fade’ is referenced.

First, a button to activate the modal is created, with the class referencing a class from the bootstrap style sheet, which is required for this to function.

Then the report modal is created. The div with the modal fade class uses bootstrap CSS to add a fade effect to the rest of the page whilst the modal has popped up. Whilst this isn’t necessary for the modal’s functionality, and I could just not include it, it adds a bit of extra emphasis to show the user exactly what part of the page they should focus on and interact with. It also adds a visual hint as to where you can click to close the modal.

The modal content div includes inside it all the content that will be contained inside the pop-up box: a heading and an input form. The header contains a title for the modal – ‘Report message’, and a close button. The close button is a button that makes use of Bootstrap’s ‘btn’, ‘btn-info’ and ‘btn-lg’ classes, as well as the ‘data-toggle’ and ‘data-target’ attributes to control the modal. It is an empty button, as it will be given a background image with CSS.

The input form will call the ‘/report’ action, using the POST method, in order to send the input data to the function in the python script with the ‘/report’ decorator. The form simply contains a text input box for users to enter their report reason, and the form will submit when the user presses enter, whilst typing into the text box. Whilst I could add a separate submit button, I don’t think this is necessary. The form will submit when the enter key is pressed, whether there is also a submit button or not, making the submit button functionally useless. Also, unlike with the message input form, I don’t think it looks stylistically better to have a separate submit button.

However, there was an issue where using the modal was, for some reason, changing the sizing of the search box, making it drastically smaller when I added the bootstrap styling. I found that this was because it was, for some reason, making use of the box-sizing property that it had gotten from bootstrap. To fix this, I added a ‘box-sizing: content-box’ to my own CSS for the search box, which took precedence over bootstrap’s property. Content-box just sets the box sizing to the default, which is different to the border-box setting it was getting from bootstrap. I am still unsure as to why it was using bootstrap styling for the search box, because, as far as I know, I was not calling any bootstrap templates.

Text

Description automatically generatedI then added my own CSS to style the modal, first styling the activator button. First, I turned the AI message bubbles into CSS grid containers using the below CSS code. This splits the message bubble into a grid with two columns – one having a width of 93% of the bubble’s width, and the other having 3% of the width.

Text

Description automatically generatedI then specify that the text in the AI message bubbles should be in column one and use the below CSS to style the report button, including specifying that it should be in column two.

I set the background of the button to a report icon image, and centred it using line 114, as well as ensuring it only takes up the space inside the button using line 115. I used the ‘!important’ argument when setting the background image and removing the border, to ensure that it overwrites any Bootstrap styling, as I found that without that, there was still a border and a white background as this was part of Bootstrap’s default styling modal buttons. Leaving the border there would, however, have uses – the clickable area of the button is a rectangle and not just the area of the PNG background image, so users could click slightly off the image, and it would still work as a button. Therefore, the border has a use as it shows users explicitly where to click. However, I feel like the button works out better without the border, as it looks better stylistically and it makes it harder for users to miss-click, by making the button still work even when they didn’t click exactly on the image (chances are if they are clicking right next to the image, they intended to click the button).

Text

Description automatically generatedI then repeated the below CSS code for all different states of the button (active, hover, focus and visited) as this was necessary to overwrite all of the Bootstrap styling and ensure a consistent design even when clicked on.

Next, I created styling for the actual modal and its content. Firstly, I vertically centred the modal, meaning that when it pops up after the button is clicked, it appears in the middle of the page. To do this, I wrapped the modal content in a container with the class ‘vertical-alignment-helper’, and added the ‘vertical-align-center’ class to the next div, as shown by the HTML code on the page above. I then used the CSS below to make these classes render the modal box in the middle of the page.

I turned the ‘vertical-alignment-helper’ class into a table, with the div inside (‘vertical-align-center’) being a cell in the table. I then use line 151 to vertically centre the div. Then, I inherit the height and width of the modal-content class from Bootstrap’s ‘modal-dialog’ class. I also inherit the maximum width from the same class to avoid an issue where the modal window stretched to the full width of Text

Description automatically generatedthe page for some reason.

I use CSS to turn the modal header class into a grid container, with one row and two columns – one taking up 90% of the width and the other using the other 10%. The title is in grid column one, and uses a padding (measured in percentage of the overall width of the modal) to place the title relatively in the centre of the modal. The close button is in column 2, and similarly to the activator button, uses a background image, as well as specifying that the background colour would be transparent, the image will not repeat and will be centred and contained in the button, as well as having no border. This gives helps give a smooth, modern look to the modal. As with the activator button, I repeat these arguments for the focus, active and hover states of the button, to ensure a consistent styling at all times. When doing this, I also make use of the ‘!important’ attribute again, ensuring that in the case of any conflicts with Bootstrap’s default styling, my CSS takes priority. This is important as I found Bootstrap likes to add white backgrounds, as well as borders, outlines and box shadows to buttons. Therefore, the ‘!important’ arguments is useful as it ensures the button is always styled the way I intended, not accidently using any of Bootstrap’s styling.

The report reason input form uses a similar styling to the other text input forms on the web page (the search bar and message input bar), with a green background, black border with rounded corners, and padding to give a comfortable amount of spacing between the text and the edges of the text box. This means that the text box won’t feel too crowded or packed together.

The result of this is AI message bubbles that look like the following, and a modal that is centred in the middle of the page with the appropriate styling, as shown below in Figure 26 .

Figure

Graphical user interface, application

Description automatically generatedThe results of the modal styling is shown in Figure 27, with the modal being centred, both vertically and horizontally, in the middle of the page. The dimmed area outside the modal area is clickable, and will close the modal.

Figure