Chatbot Project Report – COSC 310 Individual Project

Individual Developer: Gabriel McLachlan
86257383

Original Team: Team 31

Mohammed Al-Surkhi
Jordan Colledge
Gabriel McLachlan
Jordan Ribbink
Nathan Wright

Project Description:

This project consists of a chatbot built with Electron, comes with a generic mobile interface for simplicity of use and understanding. Primarily utilizing Python and Typescript, this chatbot takes on the role of a medical doctor/remote first responder, whereas the user takes the role of a patient.

The user can ask questions, list symptoms, and in response, the bot will describe the likely illness and potential remedies.

Withing this fork of the project, the chatbot utilizes the power of the internet and various APIs to help the user search up terms and concepts where they may be confused, and suggest hospital inquiry.

My GitHub Username: gmclachlan45

Repository URL: https://github.com/GMcLachlan45/chatbot-app

Documentation for API's used

Wikipedia API

The Wikipedia API has been integrated into the chatbot so that if the user wants to know a bit more about what the Doctor is talking about, all they need to put is "Look up _____", and if it's medical related, the Bot will pull up the info from Wikipedia.

The lookup simply uses the function to get the page's information but determining whether something is related to medicine is actually much more subtle. It looks through the categories of the topic and sees if there are specific keywords within the categories.

This helps to fill out where the team identified were some big gaps in conversation from A2 and A3. If the Bot doesn't know something, then it can search it up online. The ability to distinguish between medical topics also adds to the more human side of the Bot.

The Google APIs

All 4 of the next APIs used were integrated all together to give directions to a dynamically chosen location (the closest hospital to the user). If the user is feeling particularly bad, or if the bot detects that something sounds particularly bad, then the user can inquire about the directions to the nearest hospital.

This works to add an extra, more personal and interactive feature to the Bot, one that grounds it more in the real world then a set of relatively disconnected back and forths.

Though each individual api isn't extensively used, they all have their queries, parsing and formatting. More information below.

Google Geolocation API

Geolocation is a tool that uses nearby cell towers and wifi modems to triangulate the users latitude and longitude with a fair bit of accuracy (not 100% in rural places, like UBCO).

Within the getDirections() function to find the general latitude and longitude of the user. This is then converted into a string for further use within the function.

Google Geocoding API

Reverse Geocoding is the process of taking a set of coordinates and approximating a formatted address based on nearby roads and other addresses.

Within the system, we take the latitude and longitude found via geolocation, and convert it into a named location. This location is output along with the rest of the directions, and lets the user know that the Bot knows the approximate location, which can help give a sense of security that a scared and sick user may need.

Google Places API

The Places API is a useful tool to find relevant destinations. In general this could be used to determine points of interest based on what the user wants to go to, but for the Bot he is restricted to giving directions to the nearest hospital.

For this system, we take the position of the user, and use it as an origin location to bias the results to the closest and most relevant result available. Using Places itself helps to make the system more dynamic, as depending on where you are, the nearest hospital will change, making the system much more flexible.

Google Directions API

Directions is self explanatory. Once we have the user's location, and the hospital's address, the Directions API determines the fastest path between the two and the directions that need to be taken to get there.

Once the directions to the nearest hospital have been determined, the system takes all of the relevant strings, brings them together, removes the html parts and concatenates it with the rest of the Bot's response.

Though the wall of text isn't ideal, it makes much more sense that someone would be typing this out beforehand rather than in small segments. Even a user that didn't have GPS could take those instructions print them out, and follow them to where they needed to go. I also personally believe that this makes him a lot more like a first responder, and a lot more human.

API Conclusion

Those were the 5 APIs integrated into my final rendition of Doctor Phil.

The Wikipedia integration helps to extend conversational potential, adds to its humanness through asking about relevance, and help to clarify anything that the Doctor has said.

Meanwhile, the Google Maps integration helps to add a flexible, reliable and useful tool to the Doctor's coversational toolset while making him seem more human and part of this world.

Further inquiry can be had through looking at the implementation in src/agent/agent.py.

Installation and Usage

Requirements

- Node JS - https://nodejs.org/en/
- NVM (optional) - https://github.com/nvm-sh/nvm
- Python 3 - https://www.python.org/downloads/
- Pyenv (optional) - https://github.com/pyenv/pyenv
- Pyenv-virtualenv (optional) - https://github.com/pyenv/pyenv-virtualenv
- Google Maps Services API key (optional)
  - https://developers.google.com/maps/documentation/javascript/get-api-key

Open terminal in the root of the project and run this command:

1. Install NPM dependencies

➢ npm install

2. Install Python dependencies
   - OPTIONAL. Install Pyenv & Pyenv-virtualenv to manage Python environment. Follow instructions provided within documentation

➢ Create a new virutalenv in Python 3.8.10 and activate it
➢ pyenv virtualenv 3.8.10 ${YOUR_VIRTUALENV_NAME}
➢ pyenv activate ${YOUR_VIRTUALENV_NAME}
   - Install requirements via pip by running the following command from the root folder of the project

➢ pip install -r requirements.txt

3. Run this command to train the bot's neural network:
   - npm run train
   - If this gives an error, run this instead:
   - python util/train.py
   - This may take a bit of time, but after it's run once, it doesn't need to be run again.

4. Launch development server using the following bash command in root of project

npm run start

The chatbot should launch.

5. (Optional) Get an API key for Google Maps Services
   - Visit Google API keys documentation
   - Make sure to get the Geocoding API, Places API, Geolocation API, and Directions API.
   - Replace the contents of googleapikey.txt with the API key.

From here down is just documentation from the last two projects, there isn't anything new to the individual assignment

Simplified Project Structure

In the individual project, there were no added files to the bot. All of the work was done within src/agent/agent.py.

There are a few new files for user setup (googleapikey.txt) and the Final Project Report though. As such, I've added them here.

```
.
├── ...
├── googleapikey.txt
├── config
│   └── dataset.json   -> Stores our dataset for NLP
├── documentation
│   ├── 30-Turn Convo.pdf   -> Stores images of the thirty-turn conversation as stipulated in requirements.
│   ├── DFD's.pdf   -> Stores the Data Flow Diagrams and descriptions of them.
│   └── Unit Test Descriptions.pdf   -> Stores descriptions of the unit tests used. ├── Project-Report-A2.pdf   -> Our project report document for A2
├── Project-Report-A3.pdf   -> Our project report document for A3
├── Project-Report-FINAL.pdf   -> A copy of my final project report for the individual assignment
├── src
│   ├── agent
│   │   ├── plugins
│   │   ├── tests
│   │   │   └── agent_test.py   -> Unit tests as used by Pytest. │   │   └── agent.py   -> The Python agent. Essentially used to read a query, manipulate it, and return
the results.
│   │   └── chat.py   -> The Python agent. Essentially used to read a query, manipulate it, and return the results.
│   ├── main
│   │   ├── nlp-service.ts    <-  Interfaces with Node NLP module and trains from dataset
│   │   ├── main.ts    <-  Electron entry point, also includes IPC module for communicating frontend
│   ├── renderer
│   │   ├── App.vue    <-  Vue entry point
│   │   ├── components    <-  Component structure following atomic design principles
│   │   │   ├── atoms    <-  Smallest component unit in atomic design
│   │   │   │   ├── ChatMessage.vue    <-  Vue component for chat messages
│   │   │   │   └── TypingMessage.vue    <-  Vue component for "user is typing..."
│   │   │   ├── molecules    <-  Medium sized component unit
│   │   │   │   ├── ChatBar.vue    <-  Vue component for chat bar
│   │   │   │   ├── ChatBox.vue    <-  Vue component for chat box (where messages go)
│   │   │   │   └── ChatHeader.vue    <-  Chat header component (recipient picture+name)
```

```
│   │   │   └── organisms    <-  Largest component unit (full functioning features)
│   │   │       └── ChatContainer.vue    <-  Chat container component (full chat feature)
│   │   ├── index.ejs    <-  Main HTML template where Vue is injected
│   │   ├── index.ts    <-  Webpack entry point
│   │   ├── models    <-  Stores reusable object models (i.e. classes/data structures)
│   │   │   ├── message.ts    <-  Class to identify a sent message
│   │   │   ├── service.ts    <-  Abstract class representing stateful services injected into Vue
│   │   │   └── user.ts   <-Class to identify users in chat
│   │   ├── services
│   │   │   ├── chat-service.ts    <-  Service to manipulate chat state and provide responses from AI
│   │   │   └── services.ts    <-  Services initializer plugin which is installed into Vue (inits all services)
│   │   ├── store    <-  Configuration for Vuex store (state management)
│   │   │   ├── createStore.ts    <-  Initializer for store (object factory)
│   │   │   └── modules    <-  Stores all modules involved in store
│   │   │       └── chat.ts    <-  Chat module for store (manages chat state for chat service)
│   │   └── util    <-  Misc utilities
│   │       ├── createApp.ts    <-  Factory to create Vue app with desired configuration
│   │       └── inject-context.ts    <-  Helper function for injecting key into Vue/components (for DI)
│   └── ...
├── util
│   ├── train.py   -> A script used to train TensorFlow.
│   └── trainer.py   -> Utility for configuration used in train.py.
```

NOTE: Not all files are included. Configuration files and similar files of low relevance (added clutter) are removed.

NOTE: Summary of Python files is very simplified.

Vue Components (pseudo classes)

ChatMessage.vue <ChatMessage>

Props (arguments):

- message -> Message (represents message object containing content/date/sender)

TypingMessage.vue <TypingMessage>

Props (arguments):

- user -> User (represents user typing)

ChatBar.vue <ChatBar>

Props (arguments): N/A

ChatBox.vue <ChatBox>

Props (arguments): N/A

ChatHeader.vue <ChatHeader>

Props (arguments):

- name -> String (represents name of user in header)

ChatContainer.vue <ChatContainer>

Props (arguments): N/A

NOTE: Prop typings are denoted by [propName] -> [type]. They correspond with native TypeScript types OR typings found in our src/models folder. HTML selectors (class names) are indicated by "[file].vue <[selector]>"

NOTE: While vue components are phyiscally represented as classes in code/memory and generally function like so, I don't necessarily know if it is the correct nomenclature. However , based upon the requirements, we can call them this.

Vue technically abides by the MVC (Model-View-Controller structure) where the View is connected two-way data bindding to the Controller (i.e. JS in Vue component). Models are represented as classes in our "src/models" folder

Models

```
├── models    <-  Stores reusable object models (i.e. classes/data structures)
│   ├── message.ts    <-  Class to identify a sent message
│   ├── service.ts    <-  Abstract class representing stateful services injected into Vue
│   └── user.ts   <-Class to identify users in chat
```

Message.ts (Message)

Class members:

- date: Date (message date sent)
- sender: User (message sender object)
- message: string (message body)

Service.ts (*abstract* Service)

Class members:

- protected app: Vue.App (pointer to Vue App instance)

User.ts (User)

Class members:

- name: string (represents user name)
- typing: boolean (represents whether user is typing, default=false)
- photo?: string (represents user photo URL, default=undefined)

Chatbot Project Report – COSC 310 Assignment 3

Team 31
Mohammed Al-Surkhi
Jordan Colledge
Gabriel McLachlan
Jordan Ribbink
Nathan Wright

Project Description:

The project consists of a chatbot built in Electron; the chatbot comes with a generic visual interface to ensure simplicity of use and understanding. (Apparently this is extra credit for the next assignment, which we weren't aware of until it had been implemented.) The chatbot takes on the role of a doctor, who can be asked questions about different symptoms and describe the likely illness and remedy. Thus, the user takes on the role of a patient.

Our GitHub Usernames:

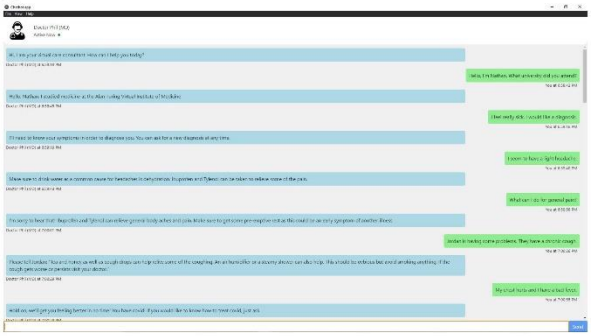
Mohammed – msurkhi-1106
Jordan C. – ItsMyFuneral
Gabriel – GmcLachlan45
Jordan R. – jribbink
Nathan – DapperShark1

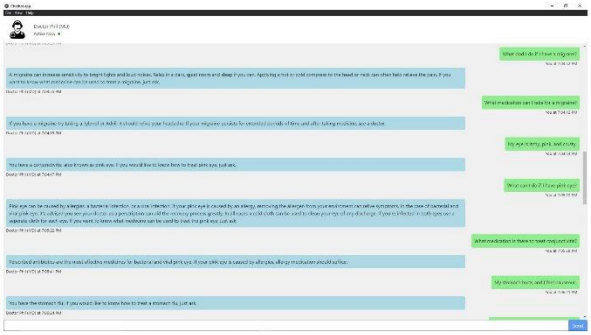
For SDLC information, WBS, and Gantt chart, please see the previous assignment's project report.

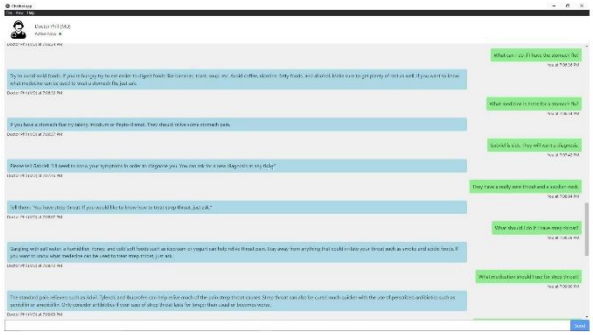
For a list of features included, the data flow diagrams, and the extractable features for an API, please see the GitHub repository and the README. The repository can be found here: https://github.com/cosc310-project/chatbot-app/tree/dev

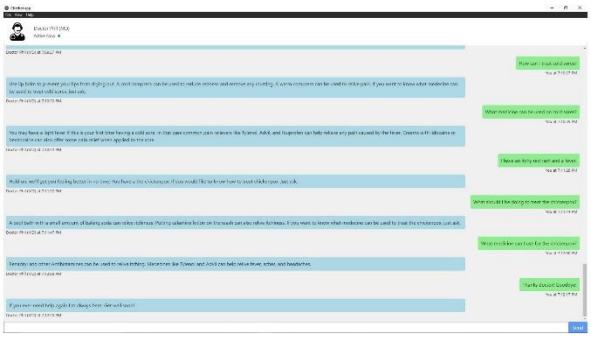
Sample Output: (30-turn dialogue on next page. It's big. Read across first, then down, because Word is finicky.)

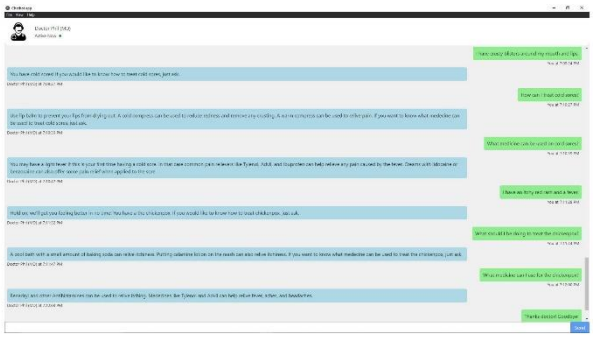
Here's a complete 30+ turn conversation between a user and our chatbot. The bot is capable of diagnosing and offering medical advice for some common illnesses as well as general health issues like a basic headache or muscle pain.





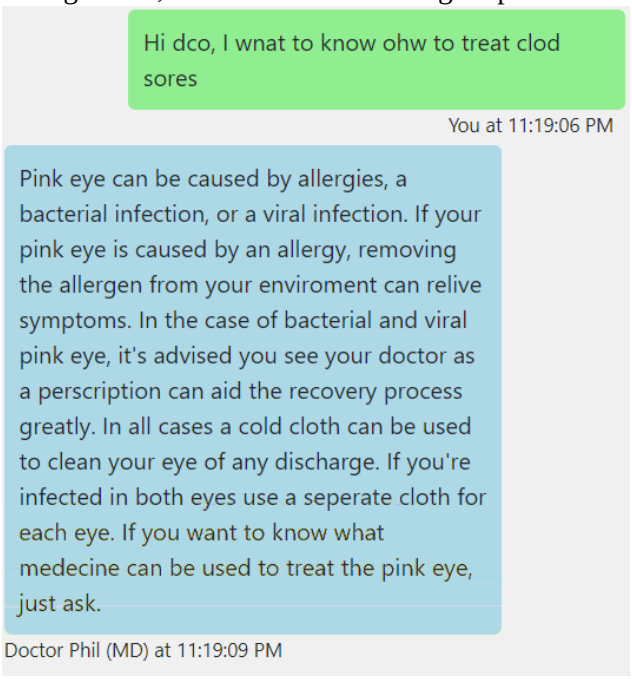






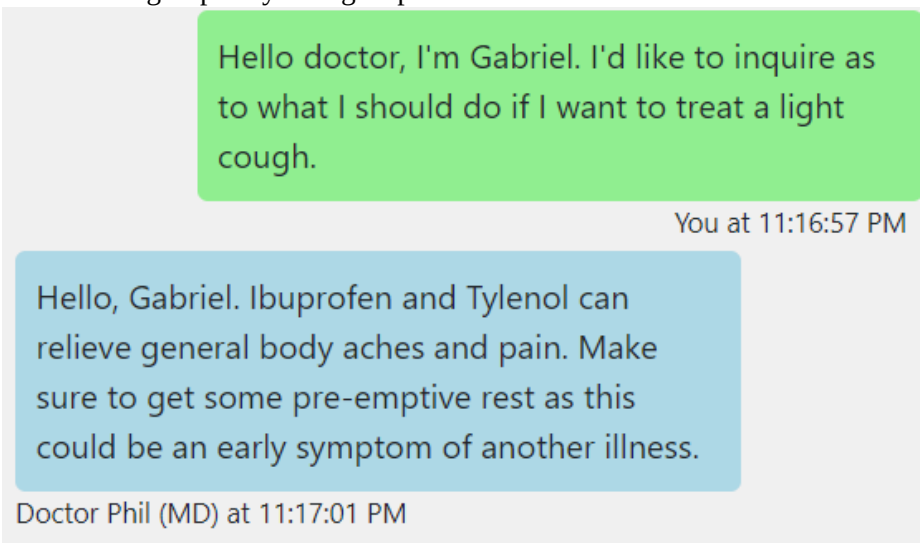
Limitations:

- Bot fails on some messages with short misspelled words. This leads to the spellcheck outputting the wrong words, which causes the wrong output for the bot.



-

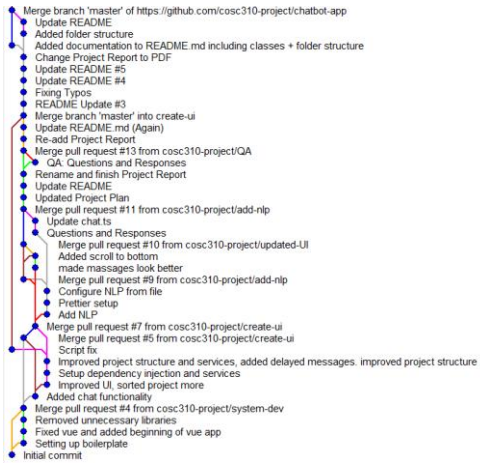
- Bot also fails on relatively long inputs, e.g. inputs longer than the expected 10-15 words. This is because the long input essentially confuses TensorFlow's learning system, and makes it more likely to provide wrong or poorly fitting responses.



You can also find these in "./assets/examples" in the Github repo.

Github Graph:

This is the entire Github tree, including all branches that were used for different features:



This was the branch structure of Assignment 2. As you can see we did much better in using branches in this assignment, something we wanted to improve on in the first section

Chatbot Project Report – COSC 310 Assignment 2

Team 31
Mohammed Al-Surkhi
Jordan Colledge
Gabriel McLachlan
Jordan Ribbink
Nathan Wright

Project Description:

The project consists of a chatbot built in Electron; the chatbot comes with a generic visual interface to ensure simplicity of use and understanding. (Apparently this is extra credit for the next assignment, which we weren't aware of until it had been implemented.) The chatbot takes on the role of a doctor, who can be asked questions about different symptoms and describe the likely illness and remedy. Thus, the user takes on the role of a patient.

GitHub repository can be found at: https://github.com/cosc310-project/chatbot-app

Our Github Usernames:

Mohammed – msurkhi-1106
Jordan C. – ItsMyFuneral
Gabriel – GMcLachlan45
Jordan R. – jribbink
Nathan – DapperShark1

Software Development Lifecycle (SDLC): Waterfall

Rationale:

With a smaller project such as this one, we found it reasonable that we would be able to use the waterfall model. It makes sense, as each assignment (assignment 2, assignment 3…) is separated, but complete, paralleling Waterfall's separation into versions.

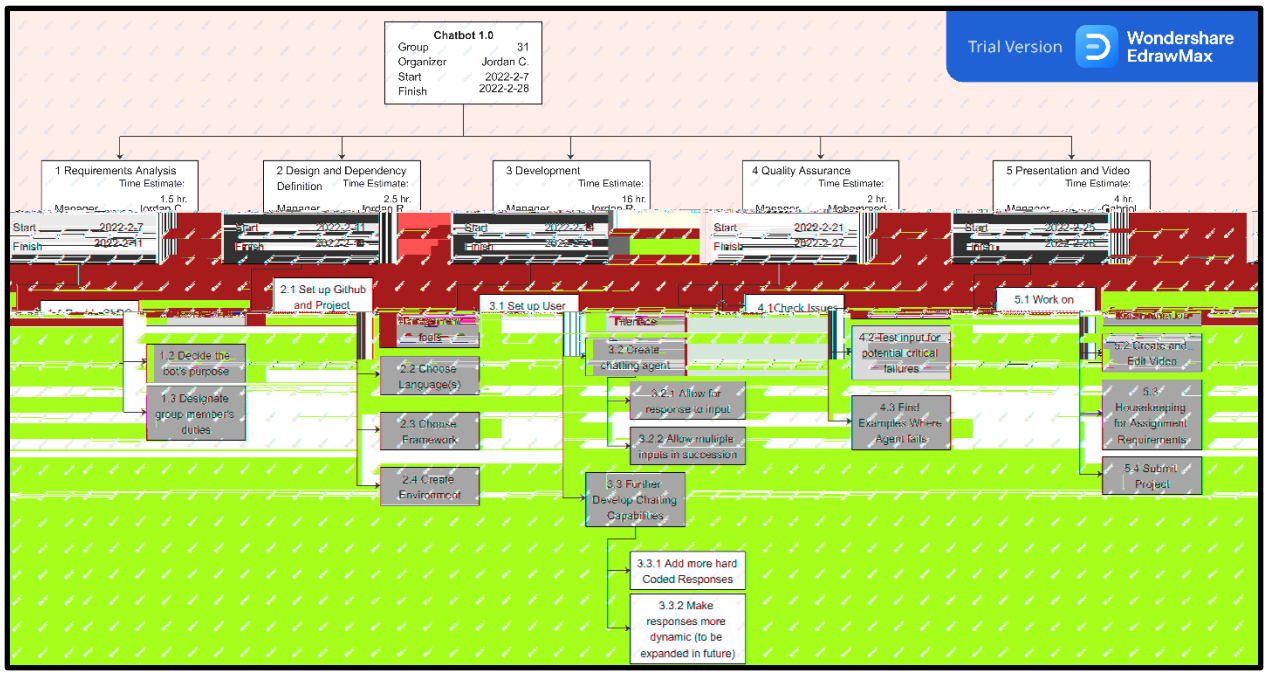
The main downside is that it's hard to accommodate change once a phase is finished in waterfall, but as the project is on a relatively small scale, we agreed that we would be able to finish the phases without much difficulty. Plus, since this project is relatively low stakes, if we run into any cataclysmic issues that *must* be addressed, we can allow ourselves to go back. Any minor issues will be saved for version 2.0 though.

SDLC Phases:

- Establish requirements
    - Determine programming language and IDE
    - Determine roles of user and agent
    - Determine topic of discussion
    - Establish and set up GitHub repository
        - Add members to GitHub repository
        - Create GitHub project
        - Add tasks to GitHub project

- System and software design
    - Study requirement specifications (from prior phase)
    - Specify system requirements
    - Define system architecture and tools
- Implementation and Unit Testing
    - Begin development in units
    - Test units
    - Fix units
    - Repeat 1-3 until units are finished
- Integration and Testing
    - Integrate units into main system
        - Push units to GitHub
        - Ensure no conflicts in integration
    - Ensure team members all up to date
    - Test system for faults/failures

Work Breakdown Structure (WBS):



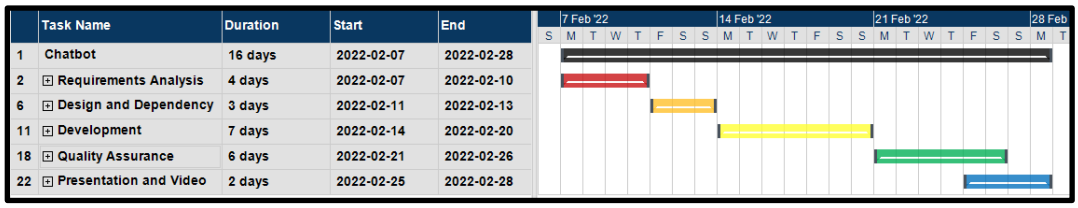
This WBS was created during the Requirements Analysis phase after we looked at the strengths from the first assignment. The main project organizer has been Jordan C., he keeps us working and was a great mediator for figuring out the rest of the tasks. Every other section was managed by someone who did well in assignment 1 in that area, so as organizer Jordan C. also facilitated the requirements, Jorden R. was the primary software architect, Mohammed focused on finding the issues, and Gabriel worked on the presentation.

You probably notice that Nathan wasn't a section organizer, but that's because he worked hard in every stage of development. We all contributed to each part a little bit but mostly focused on our portion, while he took it upon himself to go above and beyond in every part.

As this WBS was made at the start of the project, it doesn't contain the actual time it took for each section. So, here's a breakdown of the Estimated Time vs. Actual Time

Stage	Estimate (hours)	Actual (hours)	Reason
Requirements	1.5	1 - 2(including graphs)	Planning things out was easy and seeing the strengths of each member during assignment 1, designating each role was simple. The charts took longer than the deliberation.
Design	2.5	~4	Github and choosing the to use Electron didn't take too long, but setting up the environment took some trial and error
Development	16	5.5	This section was really overestimated, mostly because the UI and bot responses was actually very simple to implement. Stringing together inputs was much easier than first expected as well, which significantly reduced the development time. We decided to focus on a more hardcoded approach and would've used more time had we focused on an API in this section.
QA	2	3	No huge issues needing attention were found, but small typos and such were found throughout the week of QA, which added a bit of extra time.
Project Roundup (vid)	4	~4.5	This one was one of the most accurate. The video is simple enough, and it's just making sure we have everything in this document.

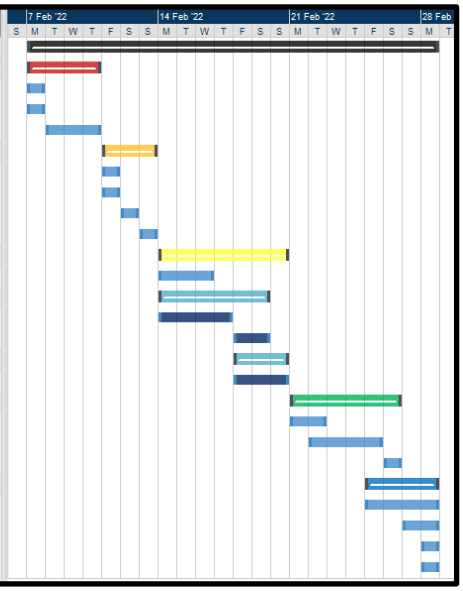
Gantt chart:
(Condensed)



This is the condensed Gantt chart for our project. It's rather simple to follow, each color is a different section of the project, and it parallels the requirements of the waterfall methodology, where all the software development parts come one after another, with no overlap. The end dates of each task are different than the WBS, but that was merely a limitation of the software.

(Full)

	Task Name	Duration	Start	End
1	Chatbot	16 days	2022-02-07	2022-02-28
2	⊟ Requirements Analysis	4 days	2022-02-07	2022-02-10
3	Choose SDLC	1 day?	2022-02-07	2022-02-07
4	Designate the Bot	1 day?	2022-02-07	2022-02-07
5	Group Member Roll Call	3 days	2022-02-08	2022-02-10
6	⊟ Design and Dependency	3 days	2022-02-11	2022-02-13
7	Set up Project Management	1 day	2022-02-11	2022-02-11
8	Choose Languages	1 day	2022-02-11	2022-02-11
9	Choose Frameworks	1 day	2022-02-12	2022-02-12
10	Create environment	1 day	2022-02-13	2022-02-13
11	⊟ Development	7 days	2022-02-14	2022-02-20
12	Set up UI	3 days	2022-02-14	2022-02-16
13	⊟ Develop Agent	6 days	2022-02-14	2022-02-19
14	Input/Output	4 days	2022-02-14	2022-02-17
15	Allow for >1 input	2 days	2022-02-18	2022-02-19
16	⊟ Flesh out Responses	3 days	2022-02-18	2022-02-20
17	Add Hard Coded Responses	3 days	2022-02-18	2022-02-20
18	⊟ Quality Assurance	6 days	2022-02-21	2022-02-26
19	Check Git Issues	2 days	2022-02-21	2022-02-22
20	Check Critical Failures	4 days	2022-02-22	2022-02-25
21	Get Examples where agent fails	1 day?	2022-02-26	2022-02-26
22	⊟ Presentation and Video	2 days	2022-02-25	2022-02-28
23	Documentation	2 days	2022-02-25	2022-02-28
24	Create and Edit video	1 day?	2022-02-27	2022-02-28
25	Housekeeping	1 day?	2022-02-28	2022-02-28
26	Submit	1 day?	2022-02-28	2022-02-28



This is the full Gantt chart. Each of the supertasks retain their colors from the condensed version, and the subtasks are the lighter blue sections beneath.

The chart is ordered by dependency, so any tasks that fully come to the right of any other task require the other tasks to be finished (i.e., Roll Call needed the SDLC and bot designation to be finished before it could be worked on).

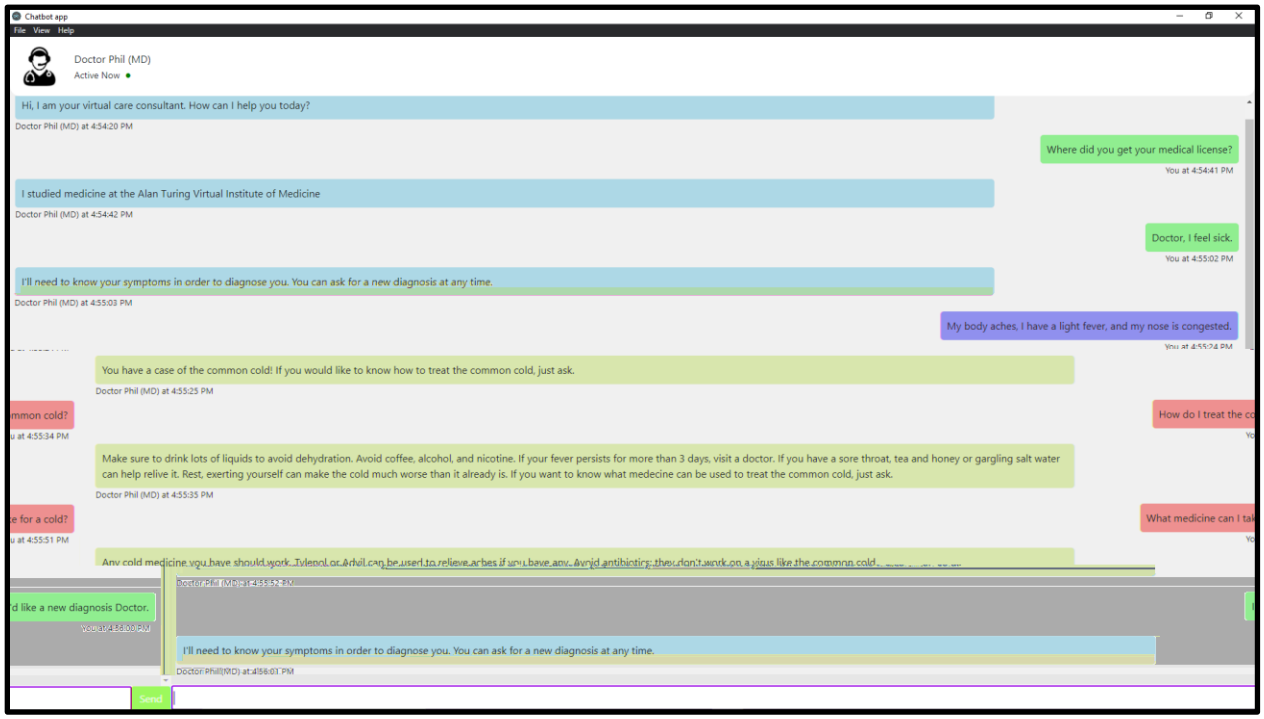
Any tasks that overlap the same time could have been taken care of in parallel (like UI and I/O), though in practice, they were still primarily worked on sequentially in order of end date.

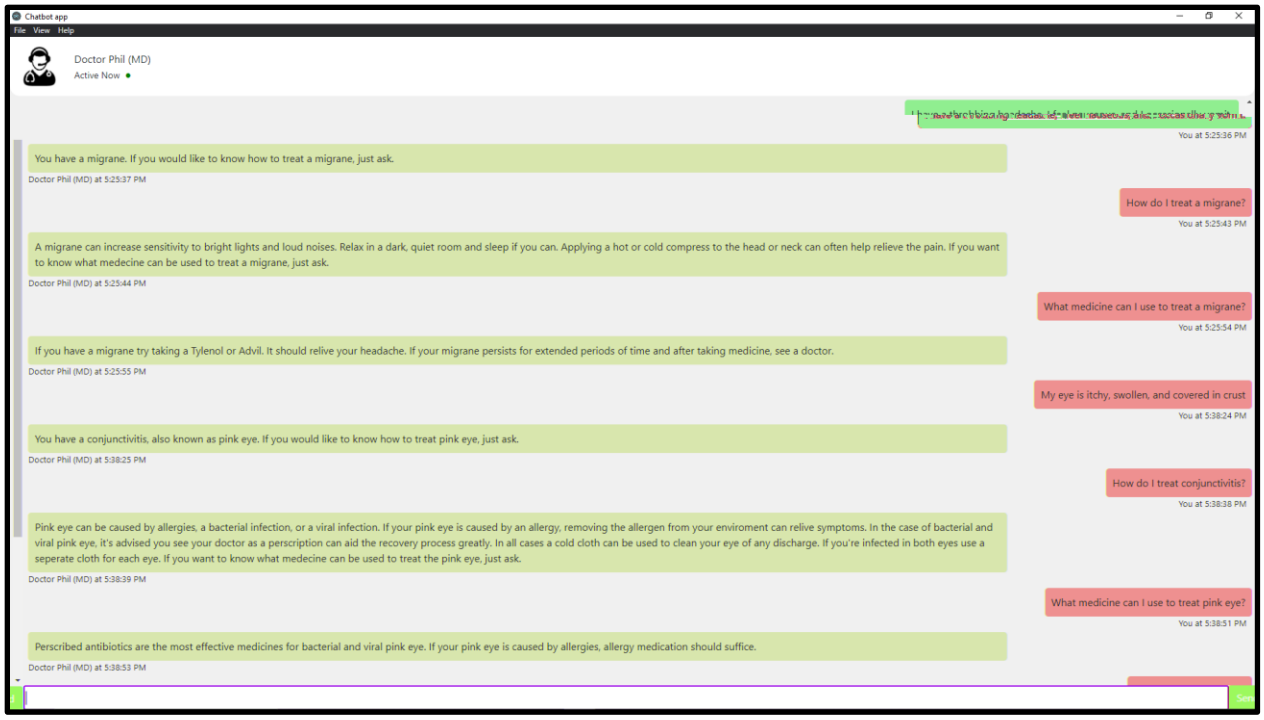
Program Limitations:

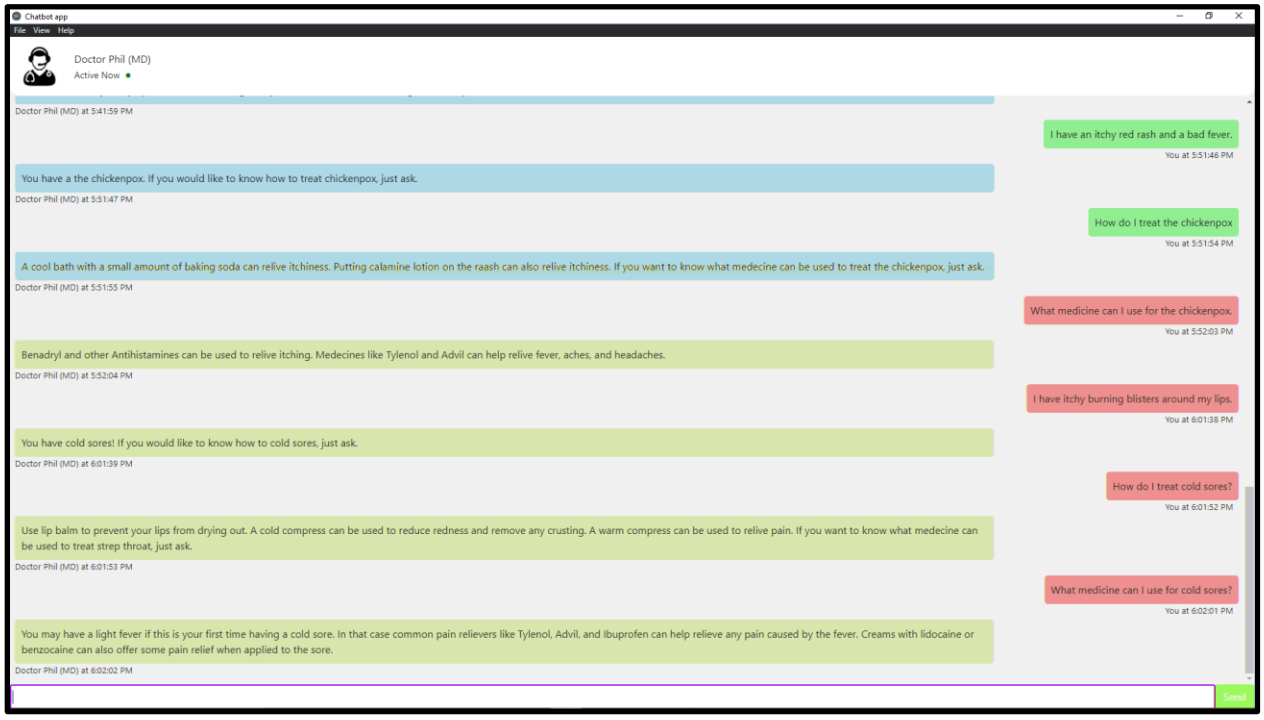
- Cannot understand typos or different terms than what's given
- Responds to strings of gibberish that include a single
- Can’t handle inputs with more than one symptom or question.
- Is restricted to its relatively small ailment dictionary (i.e., doesn’t cover every illness)

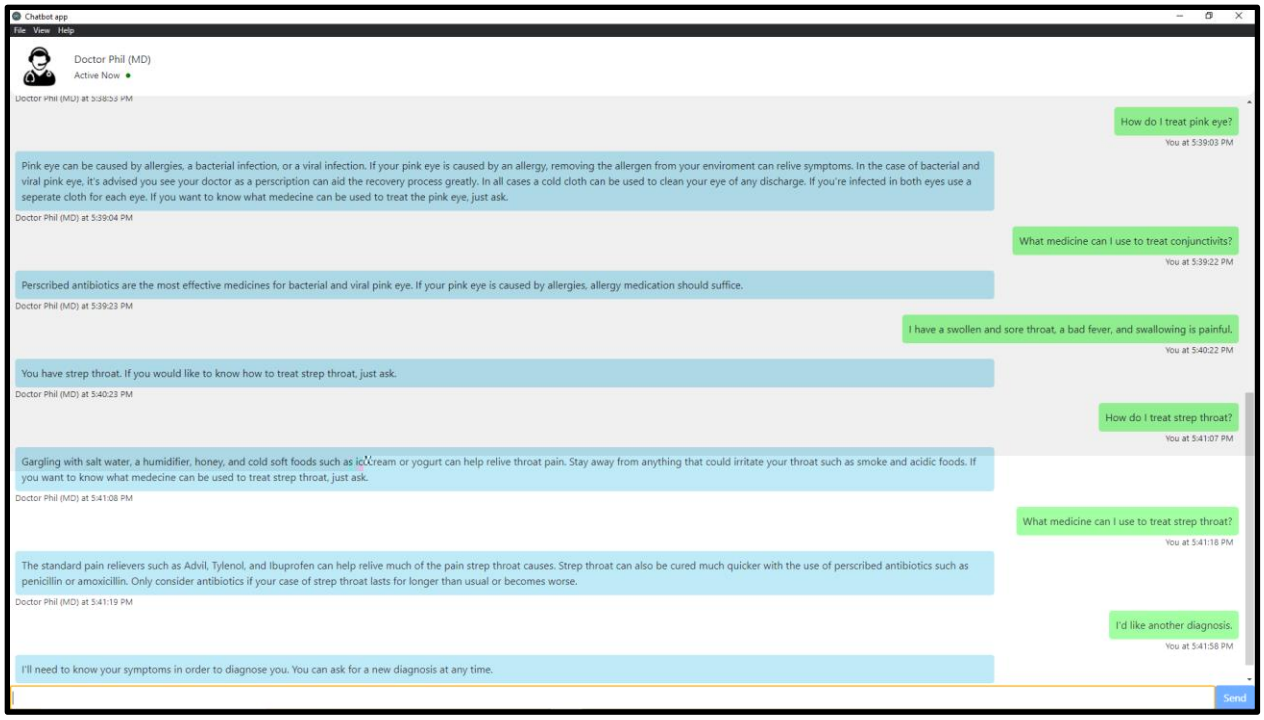
Sample Output:

Here's an example of a 30-turn conversation with our chat bot.

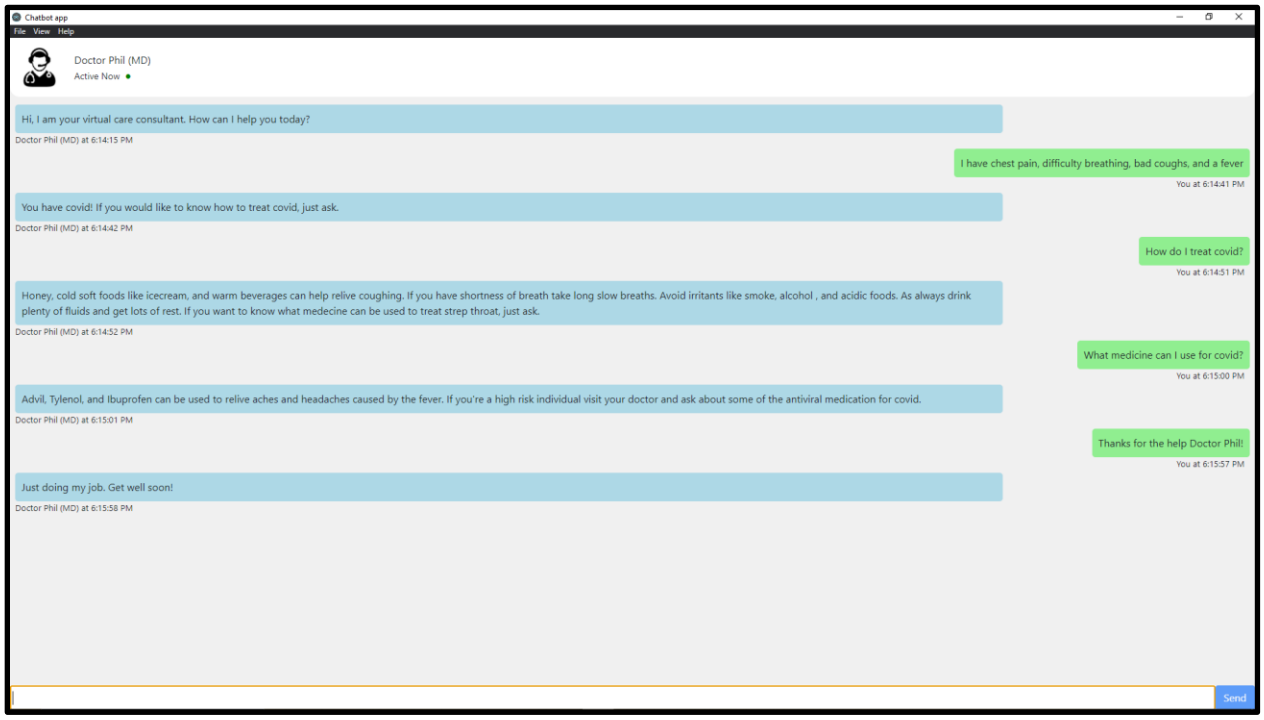



Chatbot app
File View Help
Doctor Phil (MD)
Active Now
You at 5:25:36 PM
You have a migrane. If you would like to know how to treat a migrane, just ask.
Doctor Phil (MD) at 5:25:37 PM
How do I treat a migrane?
You at 5:25:43 PM
A migrane can increase sensitivity to bright lights and loud noises. Relax in a dark, quiet room and sleep if you can. Applying a hot or cold compress to the head or neck can often help relieve the pain. If you want to know what medecine can be used to treat a migrane, just ask.
Doctor Phil (MD) at 5:25:44 PM
What medicine can I use to treat a migrane?
You at 5:25:54 PM
If you have a migrane try taking a Tylenol or Advil. It should relive your headache. If your migrane persists for extended periods of time and after taking medicine, see a doctor.
Doctor Phil (MD) at 5:25:55 PM
My eye is itchy, swollen, and covered in crust
You at 5:38:24 PM
You have a conjunctivitis, also known as pink eye. If you would like to know how to treat pink eye, just ask.
Doctor Phil (MD) at 5:38:25 PM
How do I treat conjunctivitis?
You at 5:38:38 PM
Pink eye can be caused by allergies, a bacterial infection, or a viral infection. If your pink eye is caused by an allergy, removing the allergen from your enviroment can relive symptoms. In the case of bacterial and viral pink eye, it's advised you see your doctor as a perscription can aid the recovery process greatly. In all cases a cold cloth can be used to clean your eye of any discharge. If you're infected in both eyes use a seperate cloth for each eye. If you want to know what medecine can be used to treat the pink eye, just ask.
Doctor Phil (MD) at 5:38:39 PM
What medicine can I use to treat pink eye?
You at 5:38:51 PM
Perscribed antibiotics are the most effective medicines for bacterial and viral pink eye. If your pink eye is caused by allergies, allergy medication should suffice.
Doctor Phil (MD) at 5:38:53 PM


Chatbot app
File View Help
Doctor Phil (MD)
Active Now
Doctor Phil (MD) at 5:41:59 PM
I have an itchy red rash and a bad fever.
You at 5:51:46 PM
You have a the chickenpox. If you would like to know how to treat chickenpox, just ask.
Doctor Phil (MD) at 5:51:47 PM
How do I treat the chickenpox
You at 5:51:54 PM
A cool bath with a small amount of baking soda can relive itchiness. Putting calamine lotion on the raash can also relive itchiness. If you want to know what medecine can be used to treat the chickenpox, just ask.
Doctor Phil (MD) at 5:51:55 PM
What medicine can I use for the chickenpox.
You at 5:52:03 PM
Benadryl and other Antihistamines can be used to relive itching. Medecines like Tylenol and Advil can help relive fever, aches, and headaches.
Doctor Phil (MD) at 5:52:04 PM
I have itchy burning blisters around my lips.
You at 6:01:38 PM
You have cold sores! If you would like to know how to cold sores, just ask.
Doctor Phil (MD) at 6:01:39 PM
How do I treat cold sores?
You at 6:01:52 PM
Use lip balm to prevent your lips from drying out. A cold compress can be used to reduce redness and remove any crusting. A warm compress can be used to relive pain. If you want to know what medecine can be used to treat strep throat, just ask.
Doctor Phil (MD) at 6:01:53 PM
What medicine can I use for cold sores?
You at 6:02:01 PM
You may have a light fever if this is your first time having a cold sore. In that case common pain relievers like Tylenol, Advil, and Ibuprofen can help relieve any pain caused by the fever. Creams with lidocaine or benzocaine can also offer some pain relief when applied to the sore.
Doctor Phil (MD) at 6:02:02 PM
Send

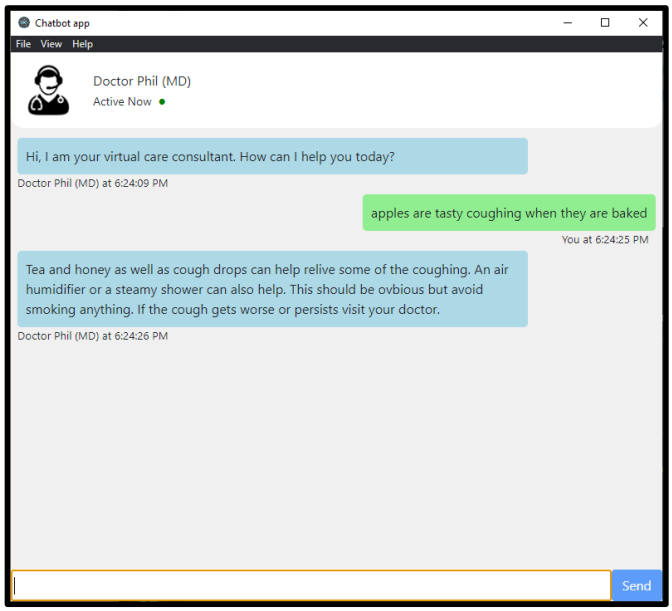


And a bonus example where we thank Doctor Phil:



Improper Handling:

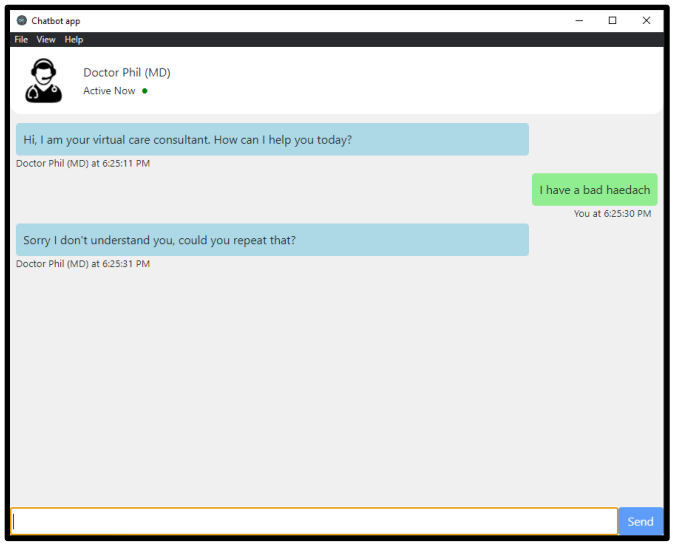
Here are a couple of examples where the bot fails to handle a conversation properly:



In this example, a weird, nonsensical input triggers a real output.

The error message should have been displayed, but instead we get a treatment to help with coughing, just because the word coughing was entered.

While it's not necessarily the worst output, it could throw doubt on the robustness of the bot and may need to be addressed.



In this example, a simple typo for an obvious word triggered the default error message.

The bot should have inferred that "headache" was intended, but instead gave us the error message.

This could be a problem if someone just didn't know how to spell their ailment, and possibly would need to be addressed.

CASE Tools Used:

Github and Github Projects: The team's decided codebase and project management system. Used to keep both the code, and the issues clear.

Edraw Max: We used the trial, web version of Edraw Max to develop the Work Breakdown Structure, with the various tasks and subtasks

MatchWare by MindView: Used to create the Gantt chart to outline the plan from the WBS. Allows for a neat and tidy view of the course of action for the project.

VS Code: The IDE of choice for the development of our project.