My Chatbot Application, Reci is an interactive chatbot which assists users in deciding what to cook for their meal.It's intended for users who may be unsure what to eat or have limited ingredients or time. Reci interacts with users using a conversational loop extracting meaningful information from input and mapping it to predefined intents and responses.

The operation of the Reci chatbot can be understood through a series of distinct stages. Initially, Reci functions as a loop, continuously reading user input. It also tracks a memory structure which stores the user's current constraints (time, ingredients, cravings etc). This is crucial as users input different keywords when using the chatbot over time (e.g., keying 15 minutes first, then realizing they only have chicken, then wanting something spicy). After, Reci detects intents like greetings using regex pattern matching over a list of intent patterns which is stored in a dictionary. If an intent is detected, Reci returns an appropriate response immediately to keep the interaction friendly and natural. Reci then proceeds to extract concrete constraints from the user's input. Time constraints are identified using regular expressions that matches expressions like "15 mins" or "30 minutes". Ingredient information is extracted by comparing tokens in the user's input against a controlled ingredient vocabulary derived from the recipe database. This ensures that only valid cooking ingredients are captured, reducing distractions from irrelevant words like "dinner" or "cook". Cravings like spicy, comfort, or healthy are detected through keyword matching. All extracted information are stored in the chatbot's memory structure allowing it to be reused across multiple turns in the conversation.

Once there is sufficient information, Reci would enter the recommendation stage where the chatbot iterates through the recipe database and applies the filtering and scoring rules. Recipes that don't match the user's time constraint are discarded. Recipes that share ingredients with the user's available ingredients are prioritised and additional scoring bonuses are applied if a recipe matches the user's stated craving. Recipes that have already been suggested will be excluded to avoid repetition. The highest scoring recipe will then be selected and presented to the user.

Lastly, Reci would generate a response by formatting the selected recipe into a clear, user friendly reply which includes the recipe name, estimated cooking time, an explanation of why the recipe was chosen, and a numbered list of cooking steps. After presenting a recommendation, Reci prompts the user to decide whether they would like another suggestion. If the user responds "yes", the chatbot recommends a different recipe using the same constraints. If the user responds "no", Reci clears previously stored constraints so that subsequent inputs are treated as a new request, preventing inaccurate recommendations caused by leftover data.

Three test cases clearly illustrate Reci's behaviour. In the first test case, I entered "I have chicken and noodles"; Reci extracted the ingredients and found matching recipes, such as soy garlic chicken noodles, returning a suitable suggestion. In the second test case, I entered "prawns, noodles, and 10 mins," and Reci extracts both the ingredients and time constraints, correctly filtering recipes to suggest only fast prawn-based noodle dishes. In the third test case, the user rejects a suggestion by answering "no" and then enters a new ingredient such as "fish".

Reci then clears the previous memory and correctly provides a fish recipe, demonstrating proper conversation state management.

In order to organise the data and code effectively, recipes, ingredient aliases, and intents are stored in structured dictionaries which makes the system easy to extend by simply adding new recipes without modifying the core logic. The code is modularised into helper functions for intent detection, ingredient extraction, time parsing, and recipe recommendation. This improves readability, maintainability, and debugging.

Advanced techniques applied include Regular expressions to enable flexible natural language input handling, Ingredient normalisation through aliases to improve robustness against varied user phrasing and a controlled ingredient vocabulary that prevents false positives during ingredient extraction. Additionally, a lightweight state management mechanism also allowed Reci to handle multi turn conversations smoothly without becoming confused by previous inputs.

Week by Week development

In the first week of building Reci, my key focus was to establish a simple working chatbot. Thus, the first version of Reci operated as a simple console based loop which accepted user input and responded with messages. The chatbot could only recognise basic keywords such as “beef” or “prawns” and return a fixed recipe suggestion. The feedback received was that the chatbot felt too rigid and unable to handle variations in user input. To fix this, I redesigned Reci to use a recipe database and a rule based recommendation mechanism instead of using fixed responses. This thus allowed Reci to become more flexible and user friendly.

In the second week of developing Reci, I managed to add an ingredient detection function which will allow Reci to suggest recipes based on the ingredients the user has. In the first few rounds of tests, Reci would detect keywords wrongly (eg. detecting dinner as an ingredient). This was fixed by making the ingredient extraction function constrained to a controlled vocabulary which is derived from the recipe database. I also made sure to make ingredient aliases clear in order to normalise different terms which users may input. This improved accuracy by alot and reduced noise during ingredient parsing . A recommendation which I got during the second week was that the chatbot could also make recommendations based on time. This led me to implement a parse_time function using regular expressions to extract numeric time constraints from free text input and then filtering out the recipes based on the time_mins attribute. I next added a craving detection function which made use of keyword matching for predefined tags which has been integrated into the scoring logic which allows the recipes to be ranked instead of being filtered out.

In week 3 , I discovered that the chatbot was unable to handle multi turn conversations. Usually when using a chatbot, users will provide data incrementally but Reci at this stage handles inputs independently one by one. In order to resolve this, I introduced a structured memory object which persists ingredients, time, cravings and recipes which has already been suggested previously. I was also told that persistent memory could cause inaccurate results if users rejected a recommendation. Thus, I fixed this by implementing a reset logic when a user answered “no” when the chatbot asked if they wanted more suggestions. This allows the chatbot to clear all the stored constraints ensuring that previous user inputs would not interfere with the results in the next cycle.

The last week was focused mainly on refinement where I added more recipes to include multiple variants like prawns, fish, rice etc. This was to reduce the chances of empty returns especially when there are strict filtering rules which are applied. On the last week, I also added defensive programming techniques to handle edge cases like ambiguous input or insufficient information or when there are no matching cases. The recommendation function was also refined to avoid duplicate suggestions by tracking previously recommended recipes.