# Project Multi Agent-Systems Report

Ramzy Labidi (2668711), Keerti Shetty(2693793), Maliki Fofana (2662677),
Mylène Brown-Coleman(2693610), Amalia Stuger (2696408)

February 7, 2022

***Keywords***— Multi-Agent Systems, GOAL Programming Language, Cooking assistant

**Abstract**

Interactive agent dependent systems have been deployed and integrated in many aspects of our lives throughout the 21st century.Transforming the way we interact with the environment around us and our relationship with technology. One of the most prominent and future-proof mediums of interaction is through verbal communication. Voice assistants, are commonly used as they are by far the most naturalistic way of accessing a web of information. However, in order to reap the advantages of the above-mentioned medium effectively and to mitigate conversational errors as much as possible, developers opted for hard-coding the interaction through employing the Intent-Entity-Context-Response (IECR) approach[5]. This approach specifically follows the conversation through the use of back-and-forth patterns and employs different design concepts and conversational trees to better the user experience. Thus, reaching a natural exchange of information tailored towards a goal reminiscent of a human conversation.

## 1 Introduction

Research in Multi-agent systems(MAS)[2], has recently been the catalyst to the development of practical logic based programming languages and tools. Which are appropriate for the implementation of such systems. Merging new programming paradigms, like Belief-Desire-Intention (BDI), is fast becoming one of the most important topics of research in multi-agent systems, in particular because this is an essential requirement for an eventual technological transfer from touch screen/keyboard dominated interactive user interfaces to conversational interfaces. GOAL[3] is one of these said languages that won the the multi-agent programming contest[1]. Based on the integration of agent tools into existing Intergrated Development Environments (IDEs) rather than starting from scratch, it the chosenlanguage in the project discussed in this paper in order to build an assistant tailored towards Cooking. The process of designing a cooking assistant and testing its functionalities empirically is laid out starting from a basic bot base-line into a fully fledged useful tool set for cooking assistance. The former, officially known as PENETAS, is capable of recognizing intents, entities and contexts of a given point in conversations. It can engage with the users conversely through the deployment of google's Dialogflow toolbox [6]. The bot is also able to respond using set patterns, known better as the happy flow, and is capable of adapting to variants in order to achieve coherence. The interface is rendered through a mixture of HTML commands embedded as prolog predicates and GOAL files specifying the action speculation. In this report, the agent's design choices are discussed in detail following an introduction and explanation of SUPPLE dialog flow's engine and preceding an evaluation of the functionalities of PENETAS.

## 2 Supple dialog flow engine

A SUPPLE DialogFlow agent is a handcrafted paradigm that makes use of conversational UX patterns. Programmed within it is a cognitive agent in order to carry out the reasoning. It is connected to Google DialogFlow and handles the intents and entities in the system. The dialog management system consists of two tasks;dialog state and dialog policy. In dialog state the agent will run the analysis of the different states of the dialog during conversation and for dialog policy, it will predict what to say next based on the conversation history. As the agent uses the handcrafted paradigm it will map all conversational stages to a finite state machine graph. Then, the dialog policy will look through this graph to determine what the user has said in a particular sub-graph and use the dialog on the successive nodes as a response.

As the approach of dialog state tracking is very limited, Google DialogFlow uses a frame-based dialog management that implements a technique called slot filling. In this technique, there are information slots that

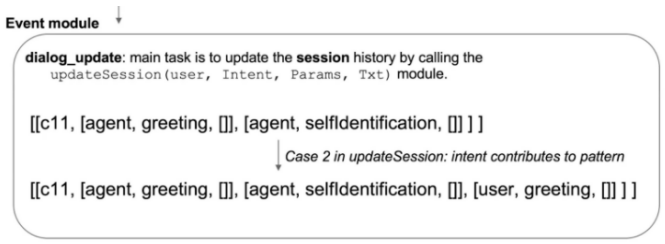
are required to be completed in order to complete a transactional stage with a user. In the scenario that a user does not give the agent not enough information, to fill the slot, the agent will ask the user for more information if it is missing any requirements. As DialogFlow uses conversational UX Design, it is important to understand what that particular method is. Conversational Analysis or Conversational UX Design is a form of pattern proposal in order to obtain a natural-sounding agent to user conversations [4]. It also enhances the usage of adaptability and re-usability of phrases within the conversation. The agent is able to tell stories to a user via the process of Extended storytelling. In addition to this, the agent is able to talk for appropriate periods of time and coordinate consent with the user about continuing the conversation. To achieve this the story is split up into different parts and the agent listens in for specific terms through adjacency pairs and the sequence of moves is mapped towards the conversation design.

Sequence Organisation is where different sequences are combined together in one conversation to interact with the user [7]. As the two actors interact there is a particular structure of moves that come about and once the purpose of the conversation has been filled the conversation closes. What makes this set-up suitable for a cooking assistant implementation? In chat bot design, the conversational UX patterns are modeled using the Intent Entity Context Response (IECR). This prototypical implementation allows for the use of Intents and Entities under the purpose of particular Contexts that the agent understands. This is demonstrated through Google DialogFlow. Furthermore, the SUPPLE Dialogmanager is suitable for a cooking assistant because it uses a knowledge base to combine conversation patterns. The combination of these particular patterns is advantageous to functions like recipe instruction when informing on how to carry out particular recipe steps. The Sequence Update Pattern Processing Logical Expansion (SUPPLE) uses dialog state tracking to update patterns and Dialog Act selection to choose what the agent says from that pattern.


Figure 1: Dialog State Tracking

Figure 1 refers to the Dialog State Tracking which will update accordingly what the user says within the pattern given the patterns from the pattern knowledge base. Whenever the user or agent says a phrase it will check that these phrases are in line with the current pattern. This coincides with the concept of contribution that says; “If an active sequence is extended with an act and matches a pattern, then all sub dialogues of that sequence can be interrupted by that pattern and extend that sequence with an act.”. In the case that a new pattern starts whilst another pattern is happening. The concept of sequence expansion is used and this says “If an act initiates a pattern, then an active sequence can also be expanded with that pattern. All active sub-dialogues of that sequence can be interrupted by the pattern, then expanded with the new sequence.”or sequence interruption which says is described as “when an act initiates a pattern, another active sequence can be interrupted by that pattern and that sequence is a sub-sequence of the extended sub sequence”. For Dialog Act Selection a concept called rule gen 2 or pattern based selection which can be described as “if by performing an act, the agent contributes to a sequence in focus, then select that act”. Agenda based selection or gen 3 which says “If all of the sub-sequences of the session are complete and the first pattern on the agenda has not been completed. Then select the first act of the pattern”. The agenda within the agent gets updated to inform the agent what to say when a pattern has been completed. When a user says a phrase, it is fed to the natural


language unit within DialogFlow, then the agent understands what has been said via the natural language understanding and responds with a greeting.Then, the agent will forward this onto the SIC EIS connecter environment. Next, the environment will send this as a precept to the GOAL agent, with no intents, after this it gets forwarded to the dialog update module.The dialog update module keeps track of the conversation. Each time something is said, it will add this to the belief base and update the session history.It tries to make sense of the implications of all the intents. Below shows an example of a contribution written in Prolog code: Next, it will send the information to the dialog generation components which consists of the slot filling, agenda and session checks. It will see that the pattern for c11 is completed and it will not add an intent and go back




to the dialog update module. The dialog update module sees that the c11 pattern is closed so it checks for the completion of the c11 pattern and starts a new empty sequence. All of this information gets sent back to the dialog generation component. The dialog generation module asks itself if the agent should be saying something, to do this it will look to see if there are other patterns left within the agenda. Here it will see the a50recipeName pattern and it will add that to the empty list. It will look into the pattern and see that there is an agent intent recipeInquiry and it adds this to what the agent will say. This will retrieve text and present it within the agents interface.

Figure 2: cooking assistant ideal usage case.

# 3 Design Choice

This section looks at the different design choices made in the implementation of PENETAS and the motivation behind each component.

## 3.1 Greeting

Initially when the agent starts, PENETAS will introduce itself mentioning it's name and the required introduction text that was given to the text() predicate in the text.pl file for the greeting pattern. This is shown in the code snippet below for the greeting pattern and the introduction:

```
pattern([greeting,[agent,greeting],[user,greeting]

agentName(Name), format(string(Txt), "Hi, I'm ~w, the Roman god of delicious food!", [Name]).
```

### 3.1.1 Conversation Starter

Next, PENETAS continues the conversation with an introduction and greeting, that is usually followed by asking politely about the user's well-being. The reference to the user's well being is an added feature introduced during week 4 of the project the allow for a human like conversational flow between the agent and the user. After the question of well being, the response processed by the agent is categorised as either positive or negative. Depending on which response the user has given, the agent replies with the corresponding socially appropriate response. The code snippet below shows the implementation for the positive and negative responses taken into the agent memory. The agent has the choice in these patterns of saying "That's great" for the positiveResponse and "I am sorry to hear that. Good food should make you feel better." for the negativeResponse.

## 3.2 Recipe Selection

In this stage the user will ask the agent about a recipe and the agent will respond with one of the recipes listed within the agent memory.This is done via the recipeChoiceReceipt intent being passed to the text() predicate in the text.pl file.

### 3.2.1 Selection by recipe name

Two versions were implemented with the recipeChoiceReceipt intent to handle the long hand and short hand names of recipes. Firstly, the agent extracts the recipe by using the shortHandName() predicate and concatenates the recipeName to a string for the agent to use within the reply.

Another design choice that was made is to handle the case where there is an unknown recipe that the user asks the agent for. In this case the agent will resort to the DefaultFallback intent which is built into DialogFlow and handles the user input that is not in the knowledge base. It will form a response to respond to the user and ask them about their requests again.

### 3.2.2 Slot Filling

This component allows the agent to fill in missing information when the agent hasn't provided the agent with sufficient information. To implement this the slotFill predicate was used that takes the requestRecipeQuantity intent and the a24detailRequest pattern as parameters. Then, the agent will respond by asking the user for which recipe are they making a request for.

### 3.2.3 User Appreciation

The agent is programmed to respond to the user in an appreciative way to allow for a more human like conversational pattern.This is implemented with the b13appreciation and the responseAppreciation intent.

### 3.2.4 Selection by feature

This was implemented to give the user the choice of selecting recipes from a particular cuisine, diet or geographical location. These are activated after to a50recipeName pattern and are incorporated into one question to allow for a humanistic conversational flow and to allow the user to start cooking faster. Intents were created within the DialogFlow agent to handle this such as:

- recipeCountryDietCalories
- recipeRequestCountry

- recipeRequestDiet
- recipeRequestCalorie

The code snippets below show how the response text to different feature selections were handled by the agent in the text.pl file:

```
currentCalorie(CalorieDown) :- keyValue(recipecalories, Calorie), downcase_atom(Calorie, CalorieDown).
currentDiet(DietDown) :- keyValue(recipediet, Diet), downcase_atom(Diet, DietDown).
```

Here in the figure above the current calorie count and diet of particular can be extracted for the user.The last part of recipe selection by features is when there is no confirmation of intent. Which is when the user is unable to select a particular features and therefore has no feature preference. In this circumstance the a50recipeName will be activated and the agent will ask the user what recipe the would like to make following that pattern.

## 3.3 Ingredient Check

### 3.3.1 Listing recipe ingredients

For the design of this it was decided that number of of servings for the current recipe selected by the user would need to be extracted first. Following for the chosen recipe each ingredient and its serving amounts are extracted, along with the name and measuring units for quantity. These are concatenated into a string and added to a list called ingredients.

Next, the list of ingredients is split in half and returned as intents called ingredientsCheck1 and ingredientsCheck2.The agent waits for confirmation from the user that they have these ingredients. If the agent receives a positive confirmation it will move onto the next pattern. If not it will terminate the sequence and skip the section within the agenda that mention the other ingredients and go to the end of the recipe.

### 3.3.2 Requesting of ingredient quantity

Whilst requesting ingredient quantity from the current recipe. The following are extracted for the user:

- ingredient name
- default amount
- measuring unit

Using the default amount divided by the default servings, Number, and the resulting value is multiplied by Servings and the resulting value, Quantity, along with the ingredient name and unit is concatenated into a string which is returned as the output.

## 3.4 Capability Check

In this component a design choice was made to inform the user about the different things that the agent is capable of cooking. This is used in the case that the user does not understand the user's intent or the user does not know what recipe they would like to make. It informs the user of all the three recipes that it has within the knowledge base. Which is implemented by creating a chosenRecipes() predicate in cooking.pl which extracts a list of recipes and the countries that they come from. This links back to the country predicate implemented in the recipes.pl file that takes a recipe and a country name as a parameter. For all recipes the dietary category, calorie count and country are stored in recipes.pl but for chosen recipes only then country name and recipe name are extracted.The code snippet below shows some of the patterns and text.pl predicates linked to the agent's response for this rule.

## 3.5 Recipe Instruction

### 3.5.1 Recipe modification

The design choice was made to decide that a recipe can be modified based on the number of servings requested by the user. In turn, this changes the recipe instruction because that ingredient quantities within each recipe step become altered. To implement this a new predicate was added into the recipes.pl file called servings which takes the short hand recipe name and the default number of ingredients. Also, the pattern below was implemented with a rule called servingRequest that triggers the text within the text.pl file, to ask the user about the number of servings that they would like.

Another predicate called currentServing was added to allow for storing the number of servings requested by the user. It takes the servings string parameter and converts it into a number using the atom number() built in predicate within Prolog.Then, it is able to calculate the amount of servings with he default number of servings for that recipe.

### 3.5.2 Clarification/Help

In the design of the recipe steps it was decided that the rule for the pattern a25recipeStep was created and the intent provideClarification. This allows the agent to extract the values from the step predicate within the recipes.pl file and asks the user to check that they have the correct ingredients.

## 3.6 Recipe Switching

In the design of the this component it was decided to include a new variant of the pattern called a30recipeName was used. In this implementation after the agent reads out a step in the recipe, the user can still continue with the current recipe or ask the agent for a different recipe.

## 3.7 Displaying relevant information

### 3.7.1 Visual Support

When PENETAS starts, it will load a page with the image of a real life android robot. Then, whilst it introduces itself verbally according to the greeting intent, the text string from the text.pl file that matches the greeting intent will appear on the right hand side of the screen, followed by the talk button on the bottom of the screen. After the users presses the talk button and gives their response. The askUserWellbeing intent will be received causing the text on the right hand side to match that stage within the text.pl file. Depending on if the user responds with a positive or negative response the agent will showcase the positive response text or negative response text on the right hand side of the screen. Next, the agent will start the recipeCountryDietCalorie intent text and display this on the right of the screen. Depending on which feature the user chooses when speaking into the talk button.

The agent proceeds to ask the user which recipe they would like to make according to the text that relates to the recipeInquiry intent. Which in turn causes the image of the selected recipe to be rendered in the middle of the page, along with the text corresponding to the askServings intent from the text.pl file. Contingent on which serving the user has selected, the agent will then proceed to make calculations about the quantity of each ingredient needed to cook the recipe. Thus leading to the agent starting the introIngredientsCheck intent text to be displayed on the screen.Following on from this, the agent will list all of the ingredients needed for the recipe and the corresponding quantities in text directly under the image on the page. If the user confirms that they have all of the needed ingredients the agent will proceed to the utensils check and a list similar to that of the ingredients check list will be displayed under the recipe image.

If the user confirms that they have all of the utensils the agent will proceed to the recipe step and display the text that relates to the recipeStep intent under the selected recipe image on the screen. As the users goes through all of the steps of the recipe, the text changes depending on the particular step number.Once the user has reached the last step of the recipe, the agent will reach the finalStep intent and display the matching message to the user. After this the lastTopicCheck intent will start and the text asking the user if they need to be assisted with anything else will show on the screen.

### 3.7.2 Handling of negative responses

In the event that the user doesn't have or is missing certain ingredients, the agent will display the text linked the negativeFarewellReciept intent at the right hand side of the screen and then follow the same pattern for the farewell intent text.

### 3.7.3 Page rendering logic

- myThreeImagesPage is a predicate that is used with three different variations.In each variation the currentRecipe predicate is used to determine the type of recipe that is shown when the page is rendered.
- onePotPastamg is a predicate that holds the image card for the one pot dairy free pasta image url. It is called upon by the myThreeImagesPage predicate.
- stirFryImg is a predicate that holds the image card for the spicy shrimp and green bean stir fry image url. It is called upon by the myThreeImagesPage predicate.
- chineseRiceImage is a predicate that holds the image card for the one pot fried rice recipe image url. It is called upon by the myThreeImagesPage predicate.
- The initialPage predicate is used to render the images of all three recipes when the agent initially starts.

For all page based predicates the button() predicate is used to render a talk button, which allows for the user to be able to speak to PENETAS. In the dialog generation GOAL module, the myThreePages predicate is used to render images and the talk button every time an intent is expected by the agent. This is captured by

the expectedIntent predicate and then the session for the agent is updated as new dialog is generated for the agent to speak whilst rendering the related image.

The dialog update GOAL module also uses the myThreePages predicate to render a new page every time the agent can believe that an event has happened via the user pressing the talk button.The expected intent predicate is used in this module to compare the incoming intents with the intent according to the agent's response dialog in the text.pl file. The code snippets below showcase the alterations to the html.pl file that were mentioned previously in the report:

```
if not(bel(waitingForEvent(_))),
bel(expectedIntent(user, _), lastAddedIntentTriple([_, Intent, _]), text(Intent, Txt),
myThreeImagesPage(Txt, 'Talk', Html))
then renderPage(Html).
```

## 3.8 Utensil check

When a utensil is required there are two different intents created to handle these scenarios. Firstly there is grantUtensilClarification and secondly there is grantClarification. The first intent is used to retrieve the utensil from the agent memory and the second intent is used to

### 3.8.1 Listing utensils

For the design of listing utensils that user would need for a particular recipe, it was decided to use the currentRecipe predicate to draw the current recipe out from memory.This was designed with the findall built in predicate within Prolog to to create complete list of utensils called UtensilsList. Each utensil was extracted using the Utensil predicate from the recipes.pl file, if the utensil is required for the recipe. For the 2nd variant the only difference is:

```
\+(utensil(Recipe, Utensil)).
```

which returns true if the utensil is not in the recipe.

In the case that the user does not have all of the required utensils, the agent terminates the sequence and responds with a default message.

The following code snippets show the response within the text.pl file that the agent would give when a user gives a positive for negative response:

```
text(grantUtensilClarification, "No") :-
currentRecipe(Recipe), currentUtensil(Utensil), \+(utensil(Recipe, Utensil)).
currentUtensil(Utensil) :- keyValue(utensil, Utensil).
```

## 3.9 Restart

### 3.9.1 Conversation repair by using fallback intents

When the user asks the agent questions that it does not know the answer to, the agent is designed to handle these questions with a type of intent called default fallback. These are built into dialogflow and connected to the b12 patterns.pl which says:

```
pattern([b12, [user, defaultFallback], [agent, paraphraseRequest]]).
text(paraphraseRequest, "I did not quite get that").
```

Also, there are maybe certain responses from the user that may cause the agent to mistmatch intents to a incorrect context. For this a contextMisamtch intent was created with the b13 pattern to inform the user before a mistake is made:

```
pattern([b13, [user, _], [agent, contextMismatch]]).
text(contextMistmatch, "I am not sure what that means in this context.").
```

## 3.10 Terminate

### 3.10.1 End of recipe

When the agent reaches the end of the recipe, a new text predicate was added in text.pl was added to handle informing the user about the final step.

### 3.10.2 Conversation closing

In the closing the agent is designed to ask the user if there is anything else that they need help with. Three different intents were created to handle the user's potential response which are the following:c40lastTopicCheck farewell, c40lastTopicCheck, wellwish,c40lastTopicCheck restart.

The first variant is the circumstance where the agent and they user both say farewell. The second circumstance is where the user says that they do not need anything from the agent, and the agent wishes them a farewell. Lastly, the agent would acknowledge the user's response with a question and then restart to a new session.

# 4 Evaluation

In order to understand the set-up and the results of the tests conducted on PENETAS, the happy flow and optimal functionalities of the agent would be explained in detail and put through multiple tests. Each part of the agent functionality was tested thoroughly and in an empiric fashion to mitigate false conclusions.

## 4.1 Opening and recipe selection

The agent usually starts out the conversation with self-introduction and a greeting, which is expected to be met with the user greeting back the agent.

```
pattern([greeting,[agent,greeting],[user,greeting]]).

pattern([conversation, [agent, askUserWellbeing], [user, positiveResponse], [agent, reply]]).

pattern([conversation, [agent, askUserWellbeing], [user, negativeResponse], [agent, replyPositive]]).
```

Thereafter the agent starts the conversation pattern, where it begins by politely asking about the user's wellbeing. At this point, there are two different possible deviations in the conversation depending on the user's reply, a positive response would lead to a gratifying 'That's great' while a negative one would lead to a spirit-lifting phrase being uttered by the agent. Then we reach the point where the user is asked about their preferences in terms of what they fancy cooking. These preferences are mainly filtered using diet, calories or particular cuisine values that the user expresses and result in 4 deviating conversations patterns.

```
pattern([a5recipeName,
[agentrecipeCountryDietCalorie],
[user,  recipeRequestCountry],
[agent, recipeCountryChoice]]).
pattern([a5recipeName,
[agent, recipeCountryDietCalorie], [user,  recipeRequestDiet], [agent, recipeDietChoice]]).
pattern([a5recipeName,
[agent, recipeCountryDietCalorie], [user,  recipeRequestCalorie],
[agent, recipeCalorieChoice]]).
pattern([a5recipeName,
[agent, recipeCountryDietCalorie], [user,  disconfirmation]]).
```

This is then followed by a another pattern where the agent asks the user about his preferences before they make their choice of a particular recipe.

```
pattern([a50recipeName, [agent, recipeInquiry],
[user, recipeRequest],
[agent, recipeChoiceReceipt]]).
pattern([a50recipeName, [agent, recipeInquiry],
[user, defaultFallback],
[agent, limitedExpertise],
[agent, describeCapability],
[agent, new(a50recipeName)]]).
```

This entices the agent to look in the knowledge base for the credentials of the requested recipe and in case it was found, it will move on to the next pattern otherwise the agent will exclaim his inability to guide the

user through the recipe steps of that particular recipe and follows that by citing the recipes it can guide the user through. After choosing the recipe, the agent requests the number of servings the user wishes to cook, here the options range from 1 to 6.

```
pattern([servingRequest, [agent, askServings], [user, sayServings], [agent, confirmServings]]).
```

## 4.2 Recipe instruction and closing

The agent can cope with many deviations from the recipe instruction, and apply slot-filling where needed. Recipe Instruction starts off with ingredients check, where all the ingredients and their amounts are mentioned after choosing the amount of servings.

```
% Positive result
pattern([a50ingredientsCheck,
[agent, introIngredientsCheck],
[agent, ingredientsCheck1],
[agent, ingredientsCheck2], [user, confirmation],
[agent, positiveReceipt]]).
%Negative result
pattern([a50ingredientsCheck,
[agent,introIngredientsCheck],
[agent, ingredientsCheck],
[user, disconfirmation],
[agent, negativeWelfareReceipt], [agent, terminate]]).
```

Thereafter, the user would confirm his readiness in terms of ingredients availability, leading to an either positive message uttered by the agent and proceeding to utensils check or a negative one that the agent terminates thereafter stating so. That is then followed by the utensils check. If the user does not have the utensils, it would have responded with a disconfirmation and thus terminate the agent. If the agent confirms to have the utensils the agent proceeds to the actual cooking of the recipe.

```
% Positive result
pattern([a50utensilsCheck, [agent, introUtensilsCheck],
[agent, utensilsCheck],
[user, confirmation],
[agent, positiveReceipt]]).
%Negative result
pattern([a50utensilsCheck,
[agent, introUtensilsCheck],
[agent, utensilsCheck],
[user, disconfirmation], [agent, negativeWelfareReceipt], [agent, terminate]]).
```

To start off the cooking procedure, the agent confirms the recipe request and then initiates a new pattern responsible for the utterance of the recipe instructions, this part's pace is partially controlled by the user as they can signify to the agent when they are done with a step, if they need further clarification to complete a particular one or in case they are wondering about the quantity of an ingredient or the need for a specific piece of cutlery. Finally, the agent would make a last topic check if it's done with the conversation by asking the question: "Can I assist you in anything else?". If the user responds with farewell or disconfirms the question the conversation is done. If the user confirms this question however, the agent restarts the agenda to provide the user with what it has wants, to create a natural flowing interaction.

the agent sends their farewell and wishes the user a good day.

```
pattern([c43farewell, [agent, farewell], [user, farewell]]).
```

```
% Positive result
pattern([a50utensilsCheck,
[agent, introUtensilsCheck],
[agent, utensilsCheck],
[user, confirmation], [agent, positiveReceipt]]).
%Negative result
pattern([a50utensilsCheck,
```

```
[agent, introUtensilsCheck], [agent, utensilsCheck],
[user, disconfirmation],
[agent, negativeWelfareReceipt], [agent, terminate]]).

pattern([a25recipeQuantity,
[user, requestRecipeQuantity],
[agent, grantRecipeQuantity]]).

pattern([a25checkUtensil,
[user, utensilClarification],
[agent, grantUtensilClarification]]).

pattern([a25recipeStep,
[user, requestClarification],
[agent, provideClarification]]).

text(provideClarification, Info) :-
currentRecipe(Recipe), stepCounter(Cnt),
elicit(Recipe, Cnt, Info).
```

## 4.3 Repair

In order to mitigate endless repair cycles and avoid taking the user out of the experience, repair patterns were set to repair the conversation when needed and return into the happy flow. first repair pattern to mention would be the default fallback pattern as it is the most basic and pivotal pattern in our agent's pattern base, this pattern exclaims an inability to comprehend an utterance to the user and requests them to repeat it. Another pivotal repair pattern would the context mismatch one as it is activated when the user's input does not match the pattern currently in the agenda or in other words irrelevant. Thirdly, the appreciation pattern that is activated when the user shows the agent appreciation, our agent reciprocates it to have a more natural and happy conversation flow. The final repair pattern to mention is the detail request pattern used for filling missing slots for the intent requestRecipeQuantity. For when the agent does not know what recipe the user refers to when asking a question. Therefore the agent asks the question "for which recipe?". To retrieve some more information to let the conversation go more smoothly.

```
pattern([a24detailRequest,
[agent, requestRecipe],
[user, recipeRequest]]).
pattern([slotFill(X), [agent, repeat(X)]]).
slotFill(requestRecipeQuantity, a24detailRequest).

b13 contextMismatch & b13appreciation
pattern([b13, [user, _], [agent, contextMismatch]]).

pattern([b13appreciation,
[user, appreciation],
[agent, responseAppreciation]]).
```

In addition to initiating and responding to repair, where possible the agent can assist the user with information (in speech and visual presentation) on what part of its utterance needs to be addressed in order to repair.

## 4.4 Extensions

Possible extra improvements: If the user asks the agent for something else, like quantity of ingredient or what can you do while making the recipe, it displays the content of the current agent intent, maybe we could modify it to display the part of the recipe it was at before the user request.

We tried the implementation of an extra pattern named: b100featureRequest. If the user asks which options it has, after it gets asked what recipe it would like to cook, the agent responds with the a5 pattern followed by the a50 again. We were not able to implement it on time sadly, because we could not make a working action that updated the session in the right manner and also reset the agenda. The action to reset the agenda gave us constant errors.

```
pattern([b100featureRequest,
[user, featureRequest], [agent, new(a5recipeName)],
[agent, new(a50recipeName)]]).
pattern([b100featureRequest,
[user, featureRequest],
[agent, limitedExpertise],
[agent, describeCapability]]).
```

Since this is going to be used while the user is working in the kitchen, after the user selects the recipe, the use of buttons could be eliminated by making this handsfree and penetas could always just listen and then identify the intents from the user's speech

## 4.5 Orientation

The Visuals shown as our agent boots up present the user namely with the possible choices and following their decision as they go through the recipe instruction walk-through, the visuals would change accordingly to the selected recipe and user's requests. These capabilities were tested by presenting the bot to people that do not know about its functionality and asking them to dictate what the visuals communicate to them at any given moment. Every single one of them had no issues when it comes to understanding the flow of the conversation and the directions they could take from there.

## 4.6 Overview

Overall, the visuals provide an overview that along with the voice module of the agent, presents the right balanced amount of information and an overview of its context and meaning. This statement's truth was tested along the orientation section, as it is a pivotal part of visual navigation, by asking users to give an overview of the procedure and was met mostly with success.

## 4.7 Testing

In order to test the capabilities of the agent, various tests were conducted:

### 4.7.1 Navigation

the agent can accommodate navigation at all relevant points in the conversation.

- Tested all 3 features with all the values for each feature
- Asked the agent for a recipe not in the database and tested how it handles this
- Asked the agent for a recipe it knows to see how it reacts
- Checked if the ingredient quantities changed based on the servings requested
- Tested the agent response to the user not having all the ingredients
- Tested the agent response to the user not having all the utensils
- Once recipe steps started, we tested step clarification by saying i don't understand this
- Once recipe steps started, we asked for ingredient amount clarification
- Once recipe steps started, we asked if a utensil was required
- Once recipe steps started, we tested switching recipes by asking for another recipe by saying i want to make stir fry

### 4.7.2 Usability

- We also tested the default fallback intents by saying irrelevant things and registered the agent's response.
- We tested capability check by asking the agent at random times what can it do
- We tested the last topic check by saying bye
- We tested the last topic check by saying no thank you
- We tested the last topic check by saying yes when the agent asked us if it could help us with something else Appreciation part was checked by saying thank you

#### 4.7.3 User Testing

- We asked for a recipe ingredient quantity without an active recipe instructions to test the slotflling part
- Tested the agent to see if it recognised the user greeting it
- Tested the agents response to the user saying i am doing good
- Tested the agents response to the user saying i am not doing so well

The bot is able to accommodate some deviations from the happy flow and provide users with a cooking assistant experience mimicking a true conversational flow. PENETAS, our agent, would need to be programmed with more recipes, more conversation scenarios and incorporated in a different more compact platform to be a truly capable assistant, however, it is still usable and capable enough to provide an overall good user experience.

# 5 Conclusion

The purpose of this project was to construct a useful voice assistant that is capable of engaging in complex verbal interchanges revolving around cooking. Based on the code provided and the tests conducted, it can be concluded that PENETAS is able to perform its function fully while adjusting to some deviations. Future work on PENETAS could yield a fully fledged assistant with rounded capabilities such as multi-threaded conversation handling and vocabulary enhancement modules. Overall, the project has been an exhaustive experience as most of the framework did not function as intended and many bugs were found in the initial set up that were not addressed. In terms of team coordination, there were no issues with the majority of group members, however one was quite distracted from the project and did not put any effort whatsoever towards finishing it. The main issue we encountered was when we hit a road block and could not finish week 1's deliverables due to a confusion with the specified English language setting in the dialog flow agent and the version of English specified in GOAL. As well as that, we had an issue with the selection of the images accompanying the recipe selection. Our initial solution of altering the predicates linked to the recipe selection intents in dialog generation and dialog update files failed. After acquiring the help of other group members that were initially assigned to other tasks, we came to realize that the logic behind selecting a particular page URL was coded in the html prolog file instead of the GOAL modules. Consequently, the image selection was successful and we were able to complete the week 3's requirements. As for the 4th week Nonetheless, this project gave us a neat insight into the world of voice recognition based assistants and introduced us to google's dialogflow platform. The part that was deemed most interesting was about understanding the logic behind the IECR [5]approach to building conversational agents, in the manner shown in the figure below, capable of properly synthesizing speech.

# Appendices

## Appendix A Week 1 report

Recipe Selection and Recipe Instruction For the recipe selection we chose three one pan pasta recipes with quite similar features. We did this altogether as a group and after we divided the tasks. This week we created a flowchart to get a visual idea of how to get the conversation, and after we did that we used dialog flow and GOAL to create the first part of the recipe instruction. We first enabled the instructions of the recipe by creating patterns in the patterns.pl file. After that we combined the end of the recipe together with closing the overall conversation. Issues and solutions: We had loads of issues starting up and testing the progress we created. We combined our knowledge and progress together so that we were able to test our progress along the way and we started working as a group of four to combine our knowledge and to create a smooth transition from the recipe selection to the recipe instruction. Action Steps for Next Week: We plan to enable conversation repair and capability check for the recipe instruction. For the recipe selection we will enable utensils check, ingredients check and also the selection of a recipe by one feature.

Visual support: To start working on a first implementation, we picked the name Penatas for our agent and we displayed the name and an image related to the chosen three recipes, using HTML. This was done by making the PastaAglioPage rule in html.pl more generic as in displaying the recipe of choice. We Found images for the recipes that we opted to further work out and defined image cards as was done for the example recipe. Thereafter defined a rule by which the page is rendered based on the currently chosen recipe and made sure to update the statements in $dialog_generation.mod2g, dialog_init.mod2g and dialog_update.mod2g as well.$

Issues and solutions: we had some issues running the software on one of our machines as it creates a rare and strange visual glitch on windows OSs. The solution to this was coordinating our work together more since the code couldn't be tested on that machine. Action Steps for Next Week: We plan on enabling buttons and

giving the user the choice of recipes by pressing one of the designated buttons. Otherwise, we will work on that week's deliverables.

# Appendix B Week 2 report

Recipe Selection and Recipe Instruction: For the recipe selection and recipe instruction we combined all our manpower to make progress this week. Issues and solutions: We had loads of issues starting up and testing the progress we created. We made some progress, however, we stayed stuck on the recipe steps. We consulted our TA and professor Florian about it, they suggested multiple solutions, we implemented all of them however it did not yield any results. We worked as a group of six to solve all the issues, however we found a lot of ways that it didn't work. We're staying hopeful and know that when we make it past this speedbump we are going to continue to work. Action Steps for Next Week: We plan to keep working to finally move on to the next step, and to implement the dialog flow agent further and go through. For the recipe selection and instruction, we plan to catch up on the work after we've enabled our agent.

Visual support: Since there was no progress with week 1 recipe instructions and the midilling with html files and css was prohibited after we implemented it, we decided to wait for our recipe instruction to initiate the instructions section and work on the design to implement this upcoming week Issues and solutions: the instructions section refused to initiate although the agent was programmed to roll it in many ways that we tried out after the advice of our professor. The solution would be to remake the agent as it's the final piece that's suspect. Action Steps for Next Week: Work on debugging the bug we encountered and focus on designing the page while doing so and implementing them.

# Appendix C Week 3 report

Recipe Selection and Recipe Instruction: We added a few features for recipe selection and recipe instruction, and we constructed predicates to define each new feature. The following features have been added:

1. Cuisine-specific recipe suggestions 2. Diet-related recipe suggestions (VEG, NON VEG etc) 3. Calorie-based recipe selection.....extended concept of estimating the number of servings and ingredients required.

We have also added the fallback response for the agent if it doesn't register the input from the user or cannot relate the input to any of the intents already mentioned in dialogflow.

Issues and solutions: The main issue was to implement each new feature in eclipse and create the respective rules while working on entities and intents in dialogflow.

Action Steps for Next Week: Selection: We will need to filter according to portion size. Different features will be added at the same time 2-3 features. extension of greetings response by adding more phrases. Instruction: We have to work together to implement 2 new features and make sure that the utensils clarification works properly.

Visual support: We're working on a background image for the visual support, which will show a certain dish if one of the recipes is selected. We're also striving to improve the website's overall layout to make it more appealing to users and much more user-friendly.

Issues and solutions: the main issue was how to render a new page over the main page for a specific dish or specific feature through eclipse.

Action Steps for Next Week: Improvement in overall layout of the webpage and make it more attractive while fixing the background picture according to the selected recipe.

# Appendix D agent flow and testing

```
[ greeting, conversation, a5recipeName, a50recipeName, servingRequest, a50ingredientsCheck,
a50utensilsCheck, a30recipe ]
```

Greeting

```
pattern([greeting,[agent,greeting],[user,greeting]]).
```

Our agent starts with greeting the user. The user responds by greeting the agent back. This pattern will lead to the conversation pattern. Conversation

```
pattern([conversation, [agent, askUserWellbeing], [user, positiveResponse], [agent, reply]]).
pattern([conversation, [agent, askUserWellbeing], [user, negativeResponse], [agent, replyPositive]]).
```

After greeting the user our agent starts the conversation pattern, where it starts by asking the users' wellbeing. The users' response can lead to two different replies. If the user responds positively the agent replies with the phrase: "That's great.". If the user responds negatively the agent tries to lift the user's spirit with a positive reply: "I am sorry to hear that. Good food should make you feel better". a5recipeName

```
pattern([a5recipeName,
[agent, recipeCountryDietCalorie],
[user,  recipeRequestCountry],
[agent, recipeCountryChoice]]).
pattern([a5recipeName, [agent, recipeCountryDietCalorie],
[user,  recipeRequestDiet], [agent, recipeDietChoice]]).
pattern([a5recipeName,
[agent, recipeCountryDietCalorie],
[user,  recipeRequestCalorie],
[agent, recipeCalorieChoice]]).
pattern([a5recipeName,
[agent, recipeCountryDietCalorie], [user,  disconfirmation]]).
```

After the greeting and conversation the agent starts the a5recipeName, "Would you like to cook food from a particular cuisine, diet or height of calories?". This gives the user the option to express it's preferences if the user has them it could go four ways. This is followed by a50recipeName. a50recipeName

```
pattern([a50recipeName, [agent, recipeInquiry], [user, recipeRequest], [agent, recipeChoiceReceipt]]).
% If the user asks for an unknown recipe the agent responds by telling the user what it can make
% And asks again to the user what it wants to make
pattern([a50recipeName, [agent, recipeInquiry], [user, defaultFallback], [agent, limitedExpertise],
[agent, describeCapability], [agent, new(a50recipeName)]]).
```

In the a50 pattern the agent first asks the user what it would like to cook. The user then has a request of a particular recipe. If this recipe is in the knowledge base the agent moves on to the next pattern. If the user asks for a recipe which is not in the knowledgebase the agent responds with the fact that it has a limited expertise and then describes which recipes it can make. And last but not least it then asks the user which recipe it would like to pick. servingRequest

```
pattern([servingRequest, [agent, askServings], [user, sayServings], [agent, confirmServings]]).
```

After choosing the recipe, the agent asks the user how many servings the user wants to make. The user has the option between 1 and 6 servings and the agent will confirm the amount of servings. a50ingredientsCheck

```
% Positive result
pattern([a50ingredientsCheck,
[agent, introIngredientsCheck],
[agent, ingredientsCheck1],
[agent, ingredientsCheck2],
[user, confirmation],
[agent, positiveReceipt]]).
%Negative result
pattern([a50ingredientsCheck,
[agent,introIngredientsCheck],
[agent, ingredientsCheck], [user, disconfirmation],
[agent, negativeWelfareReceipt], [agent, terminate]]).
```

After choosing the amount of servings the user has, the agent starts by introducing the ingredients check. If the user confirms it, the agent will proceed with a positive message: "That's amazing to hear", and proceed to the utensils check. If the user said that it doesn't have the ingredients, the agent will reply: "I'm sorry to hear that". And terminate the agenda. a50utensilsCheck

```
% Positive result
pattern([a50utensilsCheck,
[agent, introUtensilsCheck],
```

```
[agent, utensilsCheck], [user, confirmation],
[agent, positiveReceipt]]).
%Negative result
pattern([a50utensilsCheck,
[agent, introUtensilsCheck], [agent, utensilsCheck],
[user, disconfirmation], [agent, negativeWelfareReceipt],
[agent, terminate]]).
```

After the user has confirmed the ingredients, the agent proceeds to the utensils check. If the user does not have the utensils, it would have responded with a disconfirmation and thus terminate the agent. If the agent confirms to have the utensils the agent proceeds to the actual cooking of the recipe. a30recipe a30recipeStep

```
pattern([a30recipe, [agent, recipeConfirm], [agent, repeat(a30recipeStep)], [agent, finalStep]]).
pattern([a30recipeStep, [agent, recipeStep], [user, recipeContinuer]]).
%Restarts the agenda if the user requests a different recipe
pattern([a30recipeStep, [agent, recipeStep], [user, recipeRequest], [agent, restart]]).
```

After the agent has gone through all the previous steps, it moves on to a30recipe, where it confirms the recipe and then goes into the a loop of a30recipeStep where it has a back and forth with the user to go through every step.

Patterns Outside the Agenda

```
a25recipeStep, a25recipeQuantity & a25checkUtensil
pattern([a25recipeStep, [user, requestClarification], [agent, provideClarification]]).

text(provideClarification, Info) :- currentRecipe(Recipe), stepCounter(Cnt), elicit(Recipe, Cnt, Info).
```

Once the user encounters something which it does not understand while it is going through the recipe steps, it can ask a question for instance:"What do you mean?". And after this question the agent explains the step more thoroughly. And then just continues going through the recipe.

```
pattern([a25recipeQuantity, [user, requestRecipeQuantity], [agent, grantRecipeQuantity]]).
```

While cooking the user might ask a question like: "How much money do I need?". Succeeding the question of how much is needed the agent grants the recipe quantity by answering with a sentence that provides the answer.

```
pattern([a25checkUtensil, [user, utensilClarification], [agent, grantUtensilClarification]]).
```

The a25checkUtensil part is a pattern we made for when the user has a question like: "Does this recipe require a scale?". And the agent will just confirm or deny. b12 defaultFallback

```
pattern([b12, [user, defaultFallback], [agent, paraphraseRequest]]).
```

The b12 pattern is for when the agent does not recognize the input the user provides. The agent will answer: "I did not quite get that.", which implies that the agent did not understand the users' request. b13 contextMismatch b13appreciation

```
pattern([b13, [user, _], [agent, contextMismatch]]).
```

When the recognized intent that the user provided does not suit in the given pattern the agent will respond with: "I'm not sure what this means in this context", this implies that the users' question is not relevant to the agents knowledge

```
pattern([b13appreciation, [user, appreciation], [agent, responseAppreciation]]).
```

If the user shows the agent appreciation, our agent reciprocates it to have a more natural and happy conversation flow

```
c40lastTopicCheck
pattern([c40lastTopicCheck,
[agent, lastTopicCheck],
[user, farewell],
[agent, farewell]]).
pattern([c40lastTopicCheck,
[agent, lastTopicCheck],
[user, confirmation], [agent, restart]]).
pattern([c40lastTopicCheck,
[agent, lastTopicCheck],
[user, disconfirmation],
[agent, wellWish], [user, farewell],
[agent, farewell]]).
```

The last topic check is a way the agent checks if it's done with the conversation by asking the question: "Can I assist you in anything else?". If the user responds with farewell or disconfirms this question the conversation is done. If the user confirms this question however, the agent restarts the agenda to provide the user with what it has wants, to create a natural flowing interaction.

c43farewell

```
pattern([c43farewell, [agent, farewell], [user, farewell]]).
```

The farewell pattern is the end of the conversation and starts by the agent saying its goodbye to the user and the user can also respond with a farewell intent to close this pattern.

```
a24detailRequest
pattern([a24detailRequest, [agent, requestRecipe], [user, recipeRequest]]).
pattern([slotFill(X), [agent, repeat(X)]]).
slotFill(requestRecipeQuantity, a24detailRequest).
```

The a24detailRequest is used for filling missing slots for the intent requestRecipeQuantity. For when the agent does not know what recipe the user refers to when asking a question. Therefore the agent asks the question "for which recipe?". To retrieve some more information to let the conversation go more smoothly.

Testing: tested the agent to see if it recognised the user greeting it tested the agents response to the user saying i am doing good tested the agents response to the user saying i am not doing so well tested all 3 features with all the values for each feature asked the agent for a recipe not in the database and tested how it handles this asked the agent for a recipe it knows to see how it reacts checked if the ingredient quantities changed based on the servings requested tested the agent response to the user not having all the ingredients tested the agent response to the user not having all the utensils once recipe steps started, we tested step clarification by saying i dont understand this once recipe steps started, we asked for ingredient amount clarification once recipe steps started, we asked if a utensil was required once recipe steps started, we tested switching recipes by asking for another recipe by saying i want to make stir fry We also tested the default fallback intents by saying irrelevant things and registered the agent's response. we asked for a recipe ingredient quantity without an active recipe instructions to test the slotfilling part we tested capability check by asking the agent at random times what can it do we tested the last topic check by saying bye we tested the last topic check by saying no thankyou we tested the last topic check by saying yes when the agent asked us if it could help us with something else Appreciation part was checked by saying thank you

Possible extra improvements: If the user asks the agent for something else, like quantity of ingredient or what can you do while making the recipe, it displays the content of the current agent intent, maybe we could modify it to display the part of the recipe it was at before the user request.

We tried the implementation of an extra pattern named: b100featureRequest. If the user asks which options it has, after it gets asked what recipe it would like to cook, the agent responds with the a5 pattern followed by the a50 again. We were not able to implement it on time sadly, because we could not make a working action that updated the session in the right manner and also reset the agenda. The action to reset the agenda gave us constant errors. pattern([b100featureRequest, [user, featureRequest], [agent, new(a5recipeName)], [agent, new(a50recipeName)]]). pattern([b100featureRequest, [user, featureRequest], [agent, limitedExpertise], [agent, describeCapability]]). Since this is going to be used while the user is working in the kitchen, after the user selects the recipe, the use of buttons could be eliminated by making this handsfree and penetas could always just listen and then identify the intents from the user's speech

NOTE: WE ALREADY MADE 2 IMPROVEMENTS TO OUR AGENTS WHICH WAS A REQUIREMENTT, HOWEVER THESE ARE DIFFERENT SCENARIOS, ONE'S WE FAILED TO IMPLEMENT OR WOULD LIKE TO IMPLEMENT IN THE FUTURE

# References


[1] Tristan Behrens et al. "The multi-agent programming contest from 2005–2010". In: *Annals of Mathematics and Artificial Intelligence* 59.3 (2010), pp. 277–311.

[2] Rafael H Bordini et al. "A survey of programming languages and platforms for multi-agent systems". In: *Informatica* 30.1 (2006).

[3] Koen V Hindriks and Jügen Dix. "GOAL: a multi-agent programming language applied to an exploration game". In: *Agent-oriented software engineering*. Springer. 2014, pp. 235–258.

[4] Robert J. Moore and Raphael Arar. *Conversational UX Design: A Practitioner's Guide to the Natural Conversation Framework*. New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 9781450363013.

[5] Kadek Ratih Dwi Oktarini. "Are You Flirting, Objectifying or What? a Conversation Analysis of "you're very sexy" Conversational Turn". In: *Soshum: Jurnal Sosial dan Humaniora* 10.3 (2020), pp. 294–308.

[6] Navin Sabharwal and Amit Agrawal. "Introduction to Google dialogflow". In: (2020), pp. 13–54.

[7] Emanuel A. Schegloff. *Sequence Organization in Interaction: A Primer in Conversation Analysis*. Vol. 1. Cambridge University Press, 2007. DOI: 10.1017/CBO9780511791208.