LOW-LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

Speak Pseudocode2c : A framework to convert customized pseudocode to c code

Submitted in partial fulfilment of the requirements for the award of degree of

Bachelor of Technology
in
Computer Science & Engineering

UE18CS390B – Capstone Project Phase - 2

Submitted by:

Srishti Sachan	**PES1201802126**
Shaashwat Jain	**PES1201802346**
Raghav Aggarwal	**PES120180312**
Rajdeep Sengupta	**PES120180144**

Under the guidance of

Prof. Nitin V. Pujari
Dean IQAC, Computer Science Department
PES University

June - December 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES University
(Established under Karnataka Act No. 16 of 2013)
100-ft Ring Road, Bengaluru-560085, Karnataka, India

Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

This document focuses on the general principles of design and building the framework for Speak Pseudocode2c systems and explains the low-level concepts of different layers in the framework. This will introduce the methodology used for the evaluation and further refinement of concepts, as well as act as a guide for the implementation of the modules in reference.

1.2 Purpose

This document is the low-level design document for the "Speak Pseudocode2c". The purpose of this Low-Level Design (LLD) Document is to add the necessary detail to the current project description to represent a suitable model for coding. This document is also intended to help detect implementation specifics and can be used as a reference manual for how the modules interact at a low level.

1.3 Scope

The LLD documentation presents the structure of the framework, such as the master class diagram and Design Description. The LLD uses technical terms which should be understandable to the administrators of the system. The goal is to display the code on screen via speaking. The abstract is that the users will speak the natural language pseudocode in the microphone and corresponding to that pseudocode its respective c language code will be generated. Mapping pseudocode to source code will be done automatically via the framework.

2 Design Constraints, Assumptions, and Dependencies

2.1 Constraints And Dependencies

1. Currently, we are using the Google Cloud free tier which restricts us from using their service for any commercial applications. We have to abide by their terms of service for the same.

2. Google Cloud free tier has a limitation for Speech-to-Text API which we are using in the development phase of the application, it is free for 60 minutes/month afterward it will

use the credits. Compared to other cloud vendors' Speech-To-Text Google provides very little free availability of this API.

3. The hardware requirements for the system are minimal, although there is a requirement for a microphone and speaker with a working internet connection.

2.2 Assumptions

1. The user is familiar with the pseudo code format.
2. The user has a working internet connection.
3. The host machine has a GCC compiler installed for the execution of the c programs along with python3.
4. and google cloud python library which is necessary for running the framework.
5. There is no problem with the User Microphone and Speakers.
6. The Background noise is minimum where the client is using the application.

3 Design Description

We have decided to follow the function-oriented design approach where the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant tasks in the system. The system is considered as the top view of all functions.

Function-oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.

3.1 Master Class Diagram

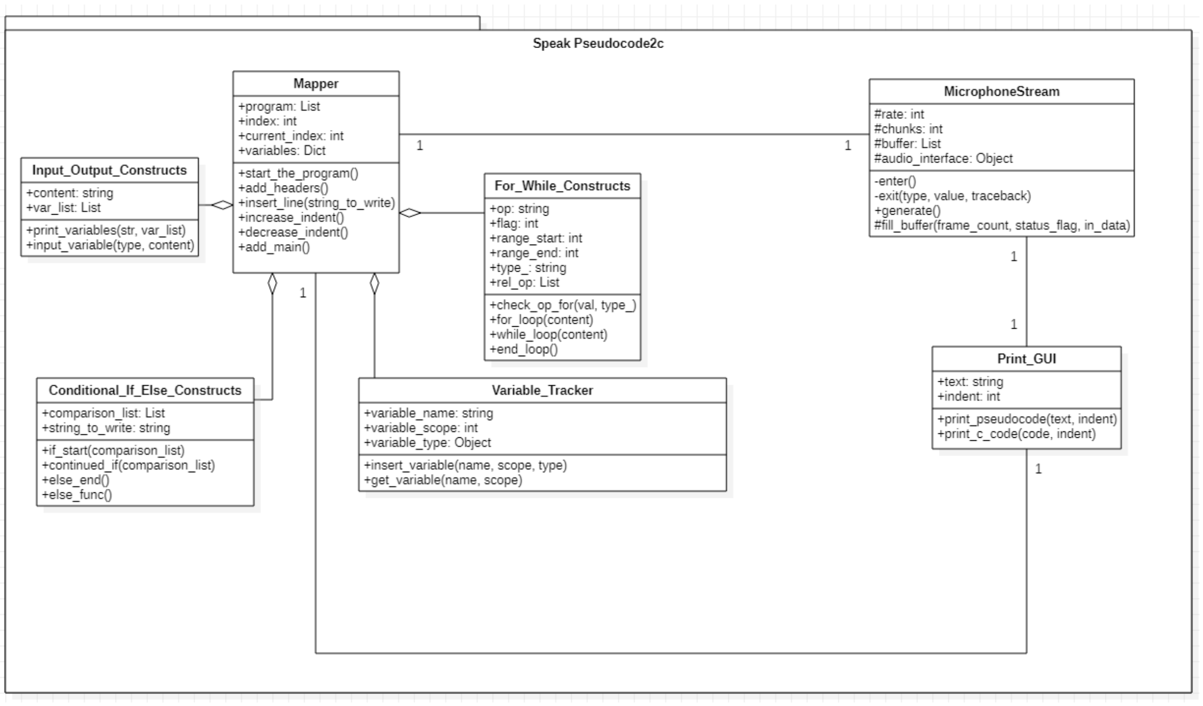

Figure 3.1: Master Class Diagram

3.2 Voice Input to Speech Module

3.2.1 Description

This module takes in the voice input from the user and sends it to the Google Cloud Server with valid credentials. First off, Google Cloud validates those credentials using a JSON Key and then the voice input is converted into a list of valid outputs. Then from the list of potential outputs, we chose the one which has the highest confidence according to the Google training model and then we process that output further.

3.2.2 Use Case Diagram

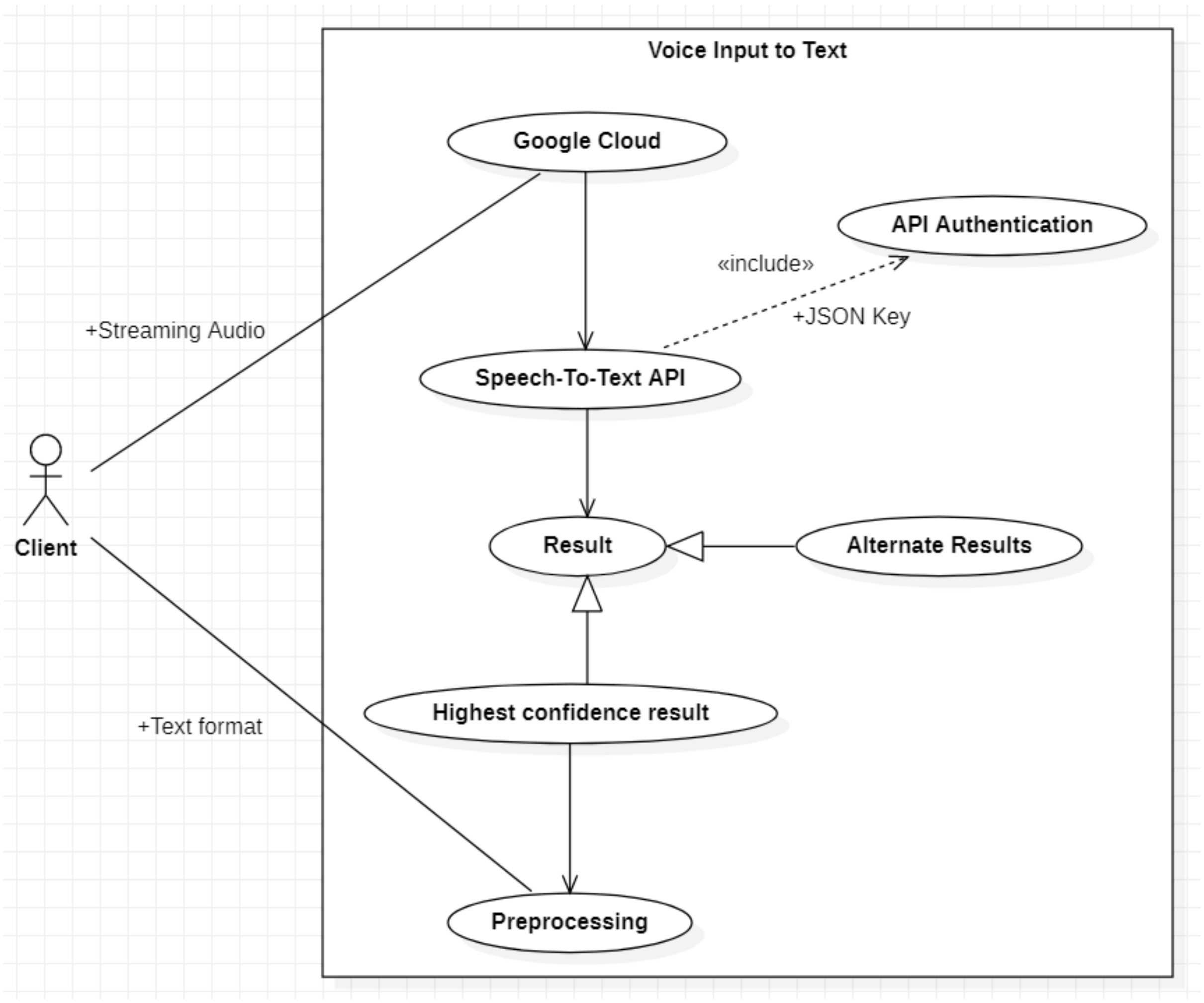

Figure 3.2: Use Case Diagram

3.2.3 Class Description - Mapper

3.2.3.1 Description

This class is responsible for mapping generated text to c code. It consists of methods that have mapping for constructs like input and output, for loop, while loop, and if-else.

3.2.3.2 Data members

Data Name	Data Type	Access Modifiers	Initial Value	Description
program	List	private	NULL	To store the converted source code
index	int	private	0	Used in insert_line
current_indent	int	private	0	To keep track of indent
variable_obj	Variable() class	public	Variable()	To keep track of variable
nlp_obj	NLP() class	public	NLP()	To support NLP

Table 3.2.3: Data Members in Mapper class

3.2.3.3 start_the_program

- **Purpose** - call functions add_headers and add_main.
- **Parameter** - mapper object

3.2.3.4 add_headers

- **Purpose** - insert the initial headers required.
- **Parameter** - mapper object
- **Input** - mapper object
- **Output** - add generated code to mapper object.

3.2.3.5 add_main

- **Purpose** - add the main function for the program.
- **Parameter** - mapper object
- **Input** - mapper object
- **Output** - add generated code to mapper object.

3.2.3.6 declare_variable

- **Purpose** - provides mapping for declaring a variable for pseudocode format - declare <variable name> <variable type> .
- **Parameter** - mapper object, generated text from speech input.

- **Input** - generated text from speech input.
- **Output** - add generated code to mapper object.

3.2.3.7 initialize_variable

- **Purpose** - provides mapping for initializing a variable for pseudocode format - initialize <variable name> = <variable value> .
- **Parameter** - mapper object, generated text from speech input.
- **Input** - generated text from speech input.
- **Output** - add generated code to mapper object.
- **Exception** - ValueError

3.2.3.8 input_variable

- **Purpose** - provides mapping to input a variable for pseudocode formats -
    1. input <variable name> <variable type>
    2. input <variable names> <variable types>
- **Parameter** - mapper object, list of generated text from speech input.
- **Input** - list of generated text from speech input.
- **Output** - add generated code to mapper object.

3.2.3.9 assign_variable

- **Purpose** - provides mapping for assigning a variable for pseudocode format - <variable result> = <variable 1> <operator> <variable 2>
- **Parameter** - mapper object, list of generated text from speech input.
- **Input** - list of generated text from speech input.
- **Output** - add generated code to mapper object.
- **Exception** - VariableNotDeclared

3.2.3.10 print_variables

- **Purpose** - provides mapping for printing a string or a variable . It handles the following formats -
  1. print variable <variable name>
  2. print <string>
- **Parameter** - mapper object, list of generated text from speech input.
- **Input** - list of generated text from speech input.
- **Output** - add generated code to mapper object.

3.2.3.11 continued_if

- **Purpose** - provides mapping for normal and nested if-else statements . It handles the following formats -
  1. if <variable1> <operator> <variable2>
  2. else if <variable1> <operator> <variable2>
  3. else
- **Parameter** - mapper object, list of generated text from speech input.
- **Input** - list of generated text from speech input.
- **Output** - add generated code to mapper object.

3.2.3.12 while_loop

- **Purpose** - provides mapping for while statements . It handles the following formats -
  1. while <variable>
  2. while <variable> <operator> <variable>
- **Parameter** - mapper object, list of generated text from speech input.
- **Input** - list of generated text from speech input.
- **Output** - add generated code to mapper object.

- **Exceptions** - VariableNotDeclared

3.2.3.13 for_loop

- **Purpose** - provides mapping of all for statements . It handles the following formats -
  1. for iterator [anything] (optional start_point) till end_point(char or int) (optional increment/decrement by int).
  2. for iterator in range from alphanumeric till alphanumeric increment by integer.
  3. for iterator in range alphanumeric till alphanumeric.
  4. for iterator in range till alphanumeric.
- **Parameter** - mapper object, list of generated text from speech input.
- **Input** - list of generated text from speech input.
- **Output** - add generated code to mapper object.
- **Exceptions** - VariableNotDeclared

3.2.3.14 end_func

- **Purpose** - To handle ending of constructs like while, for and if.
- **Parameter** - mapper object
- **Input** - mapper object
- **Output** - add closing braces to mapper object.

3.2.3.15 get_program_list

- **Purpose** - Return the contents of the program in mapper object.
- **Parameter** - mapper object
- **Input** - mapper object
- **Output** - return program

3.2.3.16 comment

- **Purpose** - Enable user narration by commenting lines which are not intended to be part of the code.
- **Parameter** - mapper object, list of generated text from speech input.
- **Input** - list of generated text from speech input.
- **Output** - add generated code to mapper object.

3.2.3.17 break_stmt

- **Purpose** - Insert break statement wherever required.
- **Parameter** - mapper object.
- **Input** - mapper object.
- **Output** - add break statement to mapper object.

3.2.3.18 continue_stmt

- **Purpose** - Insert continue statement wherever required.
- **Parameter** - mapper object.
- **Input** - mapper object.
- **Output** - add continue statement to mapper object.

3.2.3.19 process_input

- **Purpose** - process the speech input and send it to the appropriate function for further conversion.
- **Parameter** - mapper object, speech input in string format.
- **Input** - mapper object, speech input in string format.
- **Output** - return program from mapper object.

3.2.4 Class Description - Pseudocode2c

3.2.4.1 Description

This class is used to implement the graphical user interface (gui) for the framework. The class contains two vertical split text boxes which run parallely with the help of threads. The left text box is used to interact with Google Speech to text, it takes speech input customized pseudocode and converts it into text. The right text box displays the c language source code. It takes the pseudocode in text format and passes it to the Mapper class.

3.2.4.2 Data members

Data Name	Data Type	Access Modifiers	Initial Value	Description
path	Path() class	public	os.getcwd()	Get current working dir
alive	bool	public	True	Check if GUI is running
is_save	int	public	0	To check if .c is saved
font14	Tuple	private	(Times New Roman)	Set the font in GUI

Table 3.2.4: Data members in Pseudocode2c class

3.2.4.3 callback

- **Purpose** - This function contains the piece of code which will run after the thread terminates, usually, garbage cleaning.
- **Input** - class object.
- **Output** - Closes the Tkinter.

3.2.4.4 run

- **Purpose** - Thread starts running from this point. The code written under this will be executed first.
- **Input** - class object
- **Output** - Build various widgets (text box, frame, and button) of the framework .

3.2.4.5 save_code

- **Purpose** - When the save button is clicked then it executes the code under this function. It stores the text content in the right text box and writes it into the .c file.
- **Input** - class object.
- **Output** - Output - .c source program file.

3.2.4.6 compile_program

- **Purpose** - When the compile button is clicked then it executes the code under present in the right text box, but first the text needs to be stored in the .c file.
- **Input** - class object.
- **Output** - Run .c program and display result on the terminal.

3.2.4.7 remove_junk

- **Purpose** - When the undo button is clicked then it deletes the last number of lines written in the right text box i.e the conversion of pseudocode to source code for a particular line.
- **Input** - class object and count of lines written.
- **Output** - deletes the previously written lines.

3.2.4.8 exit_code

- **Purpose** - When the exit button is clicked then it executes the code under this function. It destroys all the widget created by the run function.
- **Input** - class object.
- **Output** - destroy all the widgets.

3.2.4.9 insert_lhs

- **Purpose** - It listens to the output of Google speech-To-Text and writes the text format pseudocode in the text box.
- **Input** - class object and text to be written in the left text box.

- **Output** - writes the pseudocode in left text box.

3.2.4.10 insert_rhs

- **Purpose** - It passes the output of Google speech-To-Text to the Mapper class which returns the c language source code and writes the source code in the right text box.
- **Input** - class object and text to be written in the right text box.
- **Output** - writes the c source code in right text box.

3.2.4.11 show_alert

- **Purpose** - On occurrence of any error or exception, it prompts an alert box showing the type of exception occurred and alerts users about the mistake committed.
- **Input** - class object and text to be written in the alert box.
- **Output** - Prompt the alert box if any error or exception occurs in the c program.

3.2.5 Class Description - MicrophoneStream

3.2.5.1 Description

This class opens a recording stream as a generator yielding the audio chunks. It receives the audio data, encodes it and sends it to Google cloud for processing. After processing, it receives the text output from Google Speech-To-Text API and outputs it to the graphical user input (gui).

3.2.5.2 Data members

Data Name	Data Type	Access Modifiers	Initial Value	Description
rate	int	private	16000	Rate at which audio is sent
chunk	int	private	1600	Creating small packets of audio
buff	queue.Queue()	private	Queue()	Store the encoded audio
closed	Boolean	public	False	Check if audio stream is open

Table 3.2.5: Table to show data members in Mapper class.

3.2.5.3 __enter__

- **Purpose** - It creates a thread-safe buffer of audio data and runs the audio stream asynchronously to fill the buffer object. This is necessary so that the input device's buffer doesn't overflow while the calling thread makes network requests, etc.

- **Input** - class object
- **Output** - class object and start listening to the audio and also, starts writing data to the buffer.

3.2.5.4 __exit__

- **Purpose** - Signal the generator to terminate so that the client's streaming_recognize method will not block the process termination.
- **Input** - class object, value and traceback
- **Output** - Closes the audio listener and clear the buffer.

3.2.5.5 _fill_buffer

- **Purpose** - Continuously collect data from the audio stream, into the buffer.
- **Input** - class object, in_data, frame_count, time_info, and status_flag.
- **Output** - Writes data to the buffer.

3.2.5.6 generator

- **Purpose** - Use a blocking get() to ensure there's at least one chunk of data, and stop iteration if the chunk is None, indicating the end of the audio stream.
- **Input** - class object
- **Output** - Yields the data in binary raw format.

3.2.5.7 listen_print_loop

- **Purpose** - Iterates through server responses and prints them. The response passed is a generator that will block until a response is provided by the server. Each response may contain multiple results, and each result may contain multiple alternatives. Here we print only the transcription for the top alternative of the top result.
- **Input** - responses from Google Speech-To-Text
- **Output** - Prints the customized pseudocode on the gui left text box.

3.3 Sequence Diagram

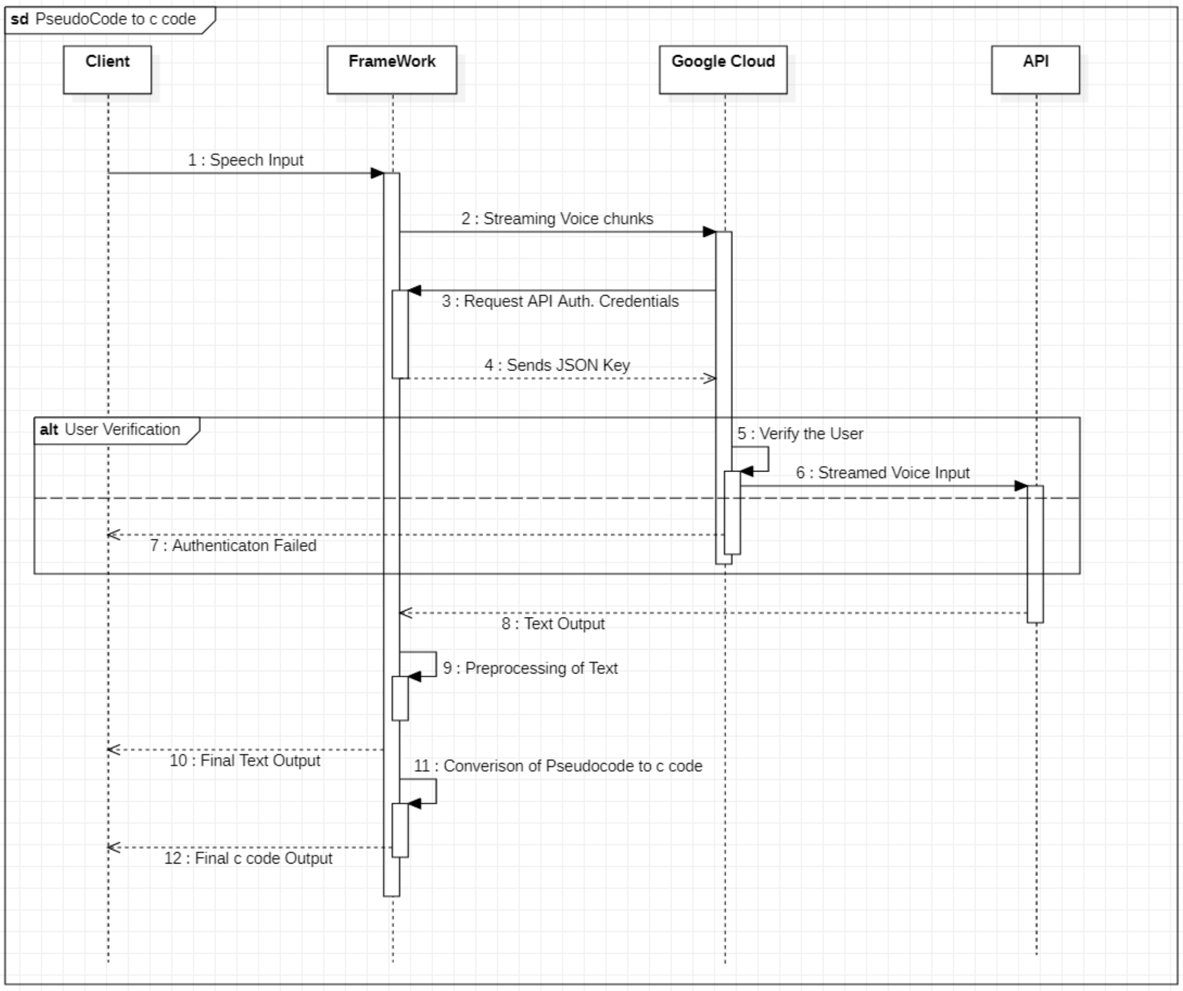

Figure 3.3: Sequence Diagram

3.4 Packaging and Deployment Diagrams

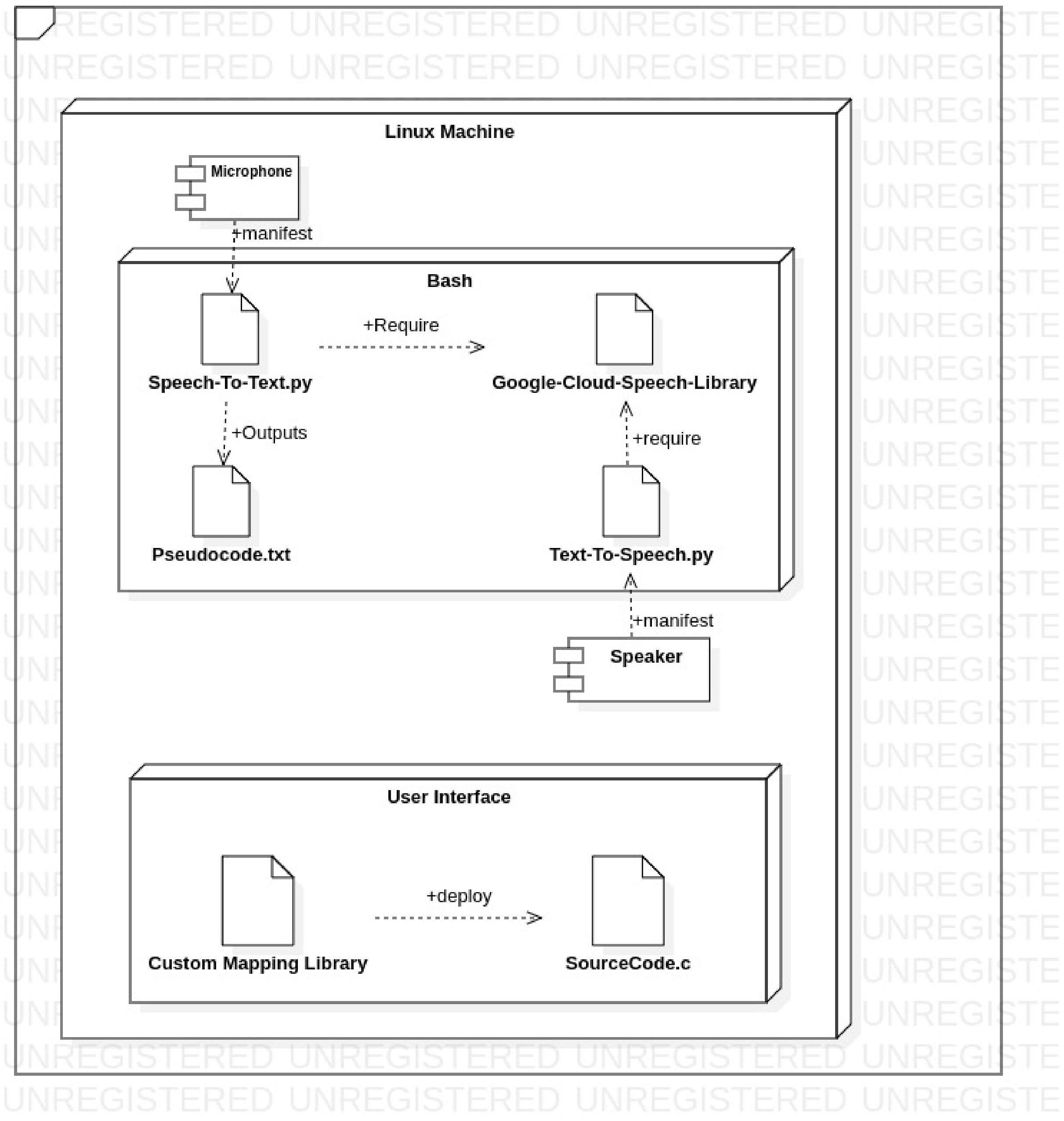

Figure 3.4: Deployment Diagram

4 Implementation and Pseudocode

4.1 Converting Voice Input to Text

4.1.1 Importing the libraries

We first import the google cloud speech to a text library which is used to send the audio data in the form of chunks. The authentication of the user is done automatically when the library is imported as it looks for a JSON file which is generated for a service account and the path to that file needs to be in the environment variable.

```
from google.cloud import speech_v1p1beta1 as speech
```

4.1.2 Configuring the parameters

Since we are sending data in the form of chunks, therefore we need to set the frequency and language of the chunks. en-IN is used to recognize English-India which improves the accuracy of the output. The audio file is encoded with LINEAR16 (16-bit linear pulse-code modulation (PCM) encoding) codec. An audio encoding refers to the manner in which audio data is stored and transmitted. The output needs to be stored somewhere in order to do preprocessing on it. To do that a filename transcript.txt is used and the output is flushed from terminal to that file for further preprocessing.

```
RATE_Of_Frequency = 44000
CHUNK_Date = int(RATE_of_Frequency / 10) #440
language_used = "en-IN" # language code
file2 = open("transcript.txt", "w")
file2.close()
config = speech.RecognitionConfig(encoding=speech.RecognitionConfig.AudioEncoding.
    ↪ LINEAR16, sample_rate_hertz=RATE, language_code=language_code, speech_contexts
    ↪ =[{"phrases": corr_list, "boost": 20.0}])
```

4.1.3 Listening to the Microphone

As there is no limit on the duration for the user to speak, therefore the microphone needs to keep on listening. In order to do that we keep the mic open and when it receives the word exit or quit in the content stream, it stops the program automatically. *MicrophoneStream* is a class which contains modules/functions that acts as a generator for audio chunks by opening a recording stream. The API in this program supports only 1-channel (mono) audio. To fill the buffer object, the audio stream runs asynchronously. This is needed to prevent the input device's buffer from overflowing as the calling thread makes network requests. This is the structure of the *MicrophoneStream* class:

```
class MicrophoneStream(object):
def __init__(self, rate, chunk):
def __enter__(self):
```

```
def __exit__(self, type, value, traceback):
def _fill_buffer(self, in_data, frame_count, time_info, status_flags):
def generator(self):
```

4.1.4 Printing the data received from server

To print the data which is received from the server we implemented the function called listen_print_loop which takes the responses as a function argument. The function Iterates through server responses and prints them. The responses passed is a generator that will block until a response is provided by the server. Each response may contain multiple results, and each result may contain multiple alternatives. We print only the transcription for the top alternative of the top result. The pseudo-code of listen_print_loop is as follows: procedure listen_print_loop(responses):

```
set variable num_chars_printed to 0
start for loop on responses using iterator as response
if response.results is None //iterate over other responses
     continue
set variable transcript to result.alternatives[0].transcript
    if result is not final
            write the data + "\r" on console
            flush the buffer
    else
            write data on console
            if exit or quit in transcript
                    stop the loop
            set variable num_chars_printed to 0
```

Print extra space Display interim results, but with a carriage return at the end of the line, so subsequent lines will overwrite them. If the previous result was longer than this one, we need to print some extra spaces to overwrite the previous result.
The implementation of listen_print_loop is as follows:

```
def listen_print_loop(responses):
        num_chars_printed = 0
        for response in responses:
        if not response.results:
                continue
```

```
# The `results` list is consecutive. For streaming, we only care about the first
    ↪ result being considered, since once it's `is_final`, it moves on to considering
    ↪  the next utterance.
        result = response.results[0]
        if not result.alternatives:
                continue
# Display the transcription of the top alternative
        transcript = result.alternatives[0].transcript
# Display interim results, but with a carriage return at the end of the line, so
    ↪ subsequent lines will overwrite them. If
the previous result was longer than this one, we need to print some extra spaces to
    ↪ overwrite the previous result
    overwrite_chars= " " * (num_chars_printed-len(transcript))
        if not result.is_final:
                sys.stdout.write(transcript + overwrite_chars + "\r")
                sys.stdout.flush()
                num_chars_printed = len(transcript)
        else:
                print(transcript + overwrite_chars)
                if re.search(r"\b(exit|quit)\b", transcript, re.I):
                        print("Exiting..")
                        break
                num_chars_printed = 0
```

4.2 Converting numeric words to digit

We wrote our custom defined function to convert the numeric words to digit. For example:

```
    one:1, two:2, five thousand six hundred seventy:5670
```

By doing this it will increase the readability of the pseudocode and it will help us in the later stage of the implementation where we have to map our custom pseudocode to the source code in c language. We have taken this a step further and also converted the basic mathematical symbols in words to actual symbols. For example:

```
    Plus : +     Minus : -        Modulo : %        is equal to : =
```

For this task we could have used the python libraries like *wordtodigits* or words2number but there was a problem with this. When speaking a word which can not be converted to a digit (normal english words) then it immediately stops listening to the server and returns an error which we don't want. Further these libraries do not map the mathematical symbols from words to actual symbols and in order to do that along with these libraries, it was increasing the latency which is again a downside. Therefore, using a custom defined function we can fill two needs with one deed.

4.3 Mapping Pseudocode to Source code

4.3.1 Variable Tracker Module

It keeps track of variable's type and scope throughout the program for easy access to other modules and exceptions like Variable Already Declared and Variable Not Declared scenarios. It is a Singleton Class.

4.3.2 Input/Output Module

The module receives distinct types of inputs from the user i.e. declare, initialize, input (scanf) and output (printf). Uses an escape keyword for variables in output called "variable" followed by variable name.

Pseudocode:-

```
declare number integer
assign number = 1
print value of number is variable number
```

Output:-

```
int number;
number = 1;
printf("value of number is %d", number);
```

4.3.3 If... else conditional module

Maps all the if, else, and else if statements by the user to the conditional statement structure in c. Has support for giving multiple relational operators in the condition.

Pseudocode:-

```
if number less than ten
print number less than ten
else
print number not less than ten
end if
```

Output:-

```
if(number<10)
{
        printf("number less than 10");
}
else
{
        printf("number not less than 10");
}
```

4.3.4 For Loop module

Takes in multiple inputs from the user for all different types of implementations possible. Can have empty declarations and can handle increment and decrement of iterators easily. It handles declaration of the iterator if needed, or uses an iterator declared in the previous scope. For example:-

Pseudocode:-

```
for i in range from 1 till number increment by 1.
end for
```

Output:-

```
for(int i = 1; i <= number; i++)
{
}
```

4.3.5 While Loop module

Maps all the while statements spoken by the user and handles any variable not declared errors by some default declaration of the variables used in the while loop. Also has support for multiple relational operators. Can use the break and continue statements to exit from the loop for some condition. For example:-

Pseudocode:-

```
while i < 10
end while
```

Output:-

```
int i = 0;
while(i < 10)
{
}
```

Appendix A: Definitions, Acronyms and Abbreviations

API:- Application Programming Interface

gcc:- GNU Compiler Collection

GCP:- Google Cloud Platform

IDE:- Integrated Development Environment

LLD:- Low Level Design

Appendix B: Record of Change History

#	Date	Document Version No.	Change Description	Reason for Change
1	24/10/2021	1	First Copy	None

Table 4.3: Record of Change History