International Institute of Information Technology, Bangalore

Project Report

Spoken-English Evaluator using DevOps pipeline

Under the Guidance of
Prof. B. Thangaraju

Akshay Mahesh Gudiyawar
MT2020137

Vikrant Punia
MT2020049

Source code link: https://github.com/akshaymg99/SPE_Speech_Evaluator.git
Docker-hub link: https://hub.docker.com/repository/docker/akshaymg99/speech-spe

Table of Contents (Index):

Abstract

We have developed a web application which analyzes spoken English for basic interview questions and provides feedback based on syntax analysis (grammatical correctness), vocabulary-strength of the user, which will also indicate the relevance of the answer relative to question asked. In effect it acts like a AI based Mock Interview performance analyzer. Here, the web application displays the question on webpage, and the user has to speak the answer, so that the application records the audio in real-time. Details on mechanism and user interaction is explained in further sections ahead. The web-application takes in the audio input and converts it to text, then executes analyzer functions on the text-corpus. The analyzer functions include vocabulary strength calculation which is done by considering word frequencies in global English dictionary. Syntax analysis is done by using pre-trained BERT model, which is a machine learning model used for NLP (Natural Language Processing) tasks. The application also calculates unique word spoken which can be beneficial in assessing interview performance. So, in effect this web application helps users to check performance of their spoken English for basic Interview questions. This entire development process is done using DevOps CI/CD Pipeline approach. We carry-out version control of source-code, testing, build/dependency installations/environment setup, containerization, deployment, logging and monitoring using various DevOps tools. The tools which we used in this project are Git, Django-test-framework, python-logger, Jenkins, Docker, Rundeck and ELK Stack. The whole process from git-push to deployment is fully automated and hence performs well in industry environment. DevOps helps in collaboration for developers and operations team and to deploy quickly to the customers.

Introduction:

In this section, we provide insight into the problem statement and what effect does the solution have on respective community / industry.

Importance of problem statement: Here the problem statement attempts to solve the problem of providing performance feedback for students who are preparing for interviews related to their spoken English. It works on the audio input of the user's answer to a specific question. It is important for students to prepare for proper communicational skills, spoken English for their interviews. Therefore, this web application will prove to be useful in assessing their spoken English skills, their grammatical correctness and vocabulary strength, meaning how powerful words they use and how unique their words are. To avoid counting common stopwords such as 'the', 'and', we give them low scores. Apart from technical skills, since candidates are checked for proper communication skills, the need to improve their language skills is of vital importance.

This entire development should be carried out using DevOps CI/CD Pipeline. Thus making the process entirely automatic with proper production environment setting.

Solution Approach: We are building the web application in Django platform. The backend processing is done in Django (python framework), where it runs Machine learning programs required for analysis of the speech. The frontend is developed with HTML/CSS and bootstrap templates. JavaScript is used for functionalities such as keeping a timer for answering and changing the display texts of recording button with events. For recording the audio input of the user, we are using "speech_recognizer" module of python, which takes in audio signals and converts them into text corpus. Upon this text, we are performing vocabulary strength analysis, which considers word frequencies from global English dictionary http://norvig.com/mayzner.html. This score is calculated by taking inverse of their frequency, thereby giving reasonably less score for most commonly used words. We have also compiled some words which are relevant to the question asked and gave scores based on how relevant are those to the question asked. The python code calculated vocabulary scores by considering these two aspects. Next for the syntax analysis, the program first breaks the text corpus into individual sentences, and then processes them accordingly. We are using BERT Natural Language Processing Machine learning model which we can fine-tune to perform syntax analysis task. More explanation on this is given in further section. By this way, the application provides feedback report on whether the sentence is grammatically correct or not. Lastly, the application also provides the unique word count, which is done by using a python tokenizer and set-data-structure. This will complete the overall feedback on spoken English of the user's answer. We have used DevOps tools to automate the entire process. It will help in quickly deploying the features through test, build checks and monitoring, it keeps the production cycle in healthy state.

The CI/CD pipeline for this project is built using the following tools:

1. Development: PyCharm, Django web framework(python), ML tools: pytorch, tensorflow, speech_recogniser, BERT NLP model.
2. Version control: Git, Git LFS, Github
3. Testing: Django-test framework (similar to Pytest)
4. Integration: Jenkins
5. Delivery: Docker, Docker hub, Jenkins
6. Deployment: Rundeck
7. Monitoring: Elastic Stack with filebeat

Source code link: https://github.com/akshaymg99/SPE_Speech_Evaluator.git
Docker-hub link: https://hub.docker.com/repository/docker/akshaymg99/speech-spe

System Configuration:

For this project the system configuration used for development/deployment are as follows:

System Details:
Model: HP-Omen
CPU: i7 9th (9750) generation CPU @ 2.6Ghz, Turbo 4.5Ghz, 6 Physical, 12 Logical cores
GPU: NVidia GTX 1650 (4GB Dedicated Memory)
RAM: 16GB DDR4, 2667 Mhz
Disk: SSD (30 GB Allocated to Ubuntu on dual boot partition)

OS Details:
Ubuntu 18.04 LTS
Kernel version: 5.4.0.72
Root Disk space: 30GB

Development tools details:
PyCharm IDE for Django-Python web development, version: 2020.3.3
Python version: 3.6.9
Django version: 3.2
ML Python library dependencies:
Pandas =0.24.2
Pydub =0.25.1
Numpy =1.16.2
Keras =2.3.1
SpeechRecognition =3.7.1
Torch =1.1.0
pytorch_pretrained_bert =0.6.2
tensorflow =1.14.0

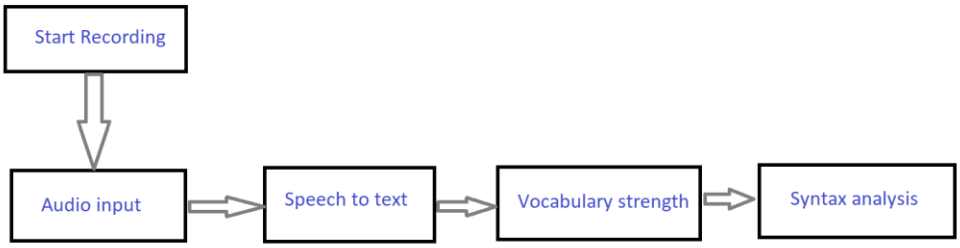
DevOps tools:
Git version: 2.17.1 , git LFS version: 2.13.3
Jenkins version: 2.277.4
Rundeck version: 3.3.10
Docker version: 20.10.6
Elastic search version: 7.11.1
Logstash version: 7.11.1
Kibana version: 7.11.1

Design and Development

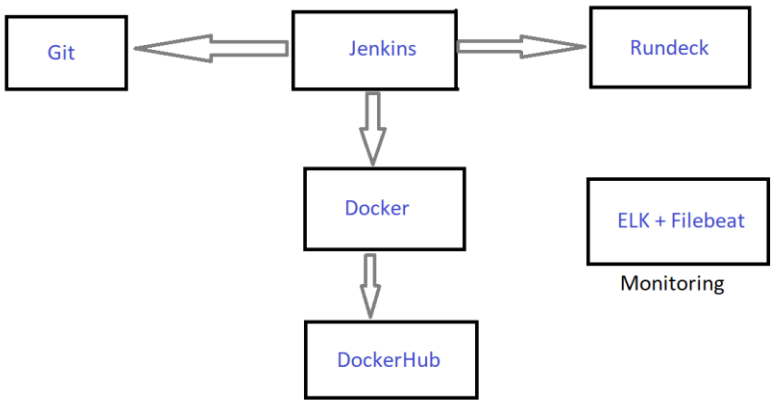
Design Architecture / Diagram

The following Figure shows the architectural diagram of our project, sub-divided into DevOps CI/CD Pipeline and Speech Analysis program.

Design Diagram of Speech Analysis Program:



Design Diagram of Development through DevOps CI/CD Pipeline:



Web application Design

Front-end Design

For designing the front-end webpages for the application, we had used HTML, CSS and JavaScript for setting events such as timer and button text changes on clicking it. Django gives the advantage of setting the base template and extending it to other pages.

First, we had designed a welcome page, where it shows the information on the features it provides (speech to text, vocab strength calculation and syntax checking). It provides a button which on clicking takes the user to Recording page, where the user can speak his answer to the question displayed. The welcome page is as shown below in the Figure [1].

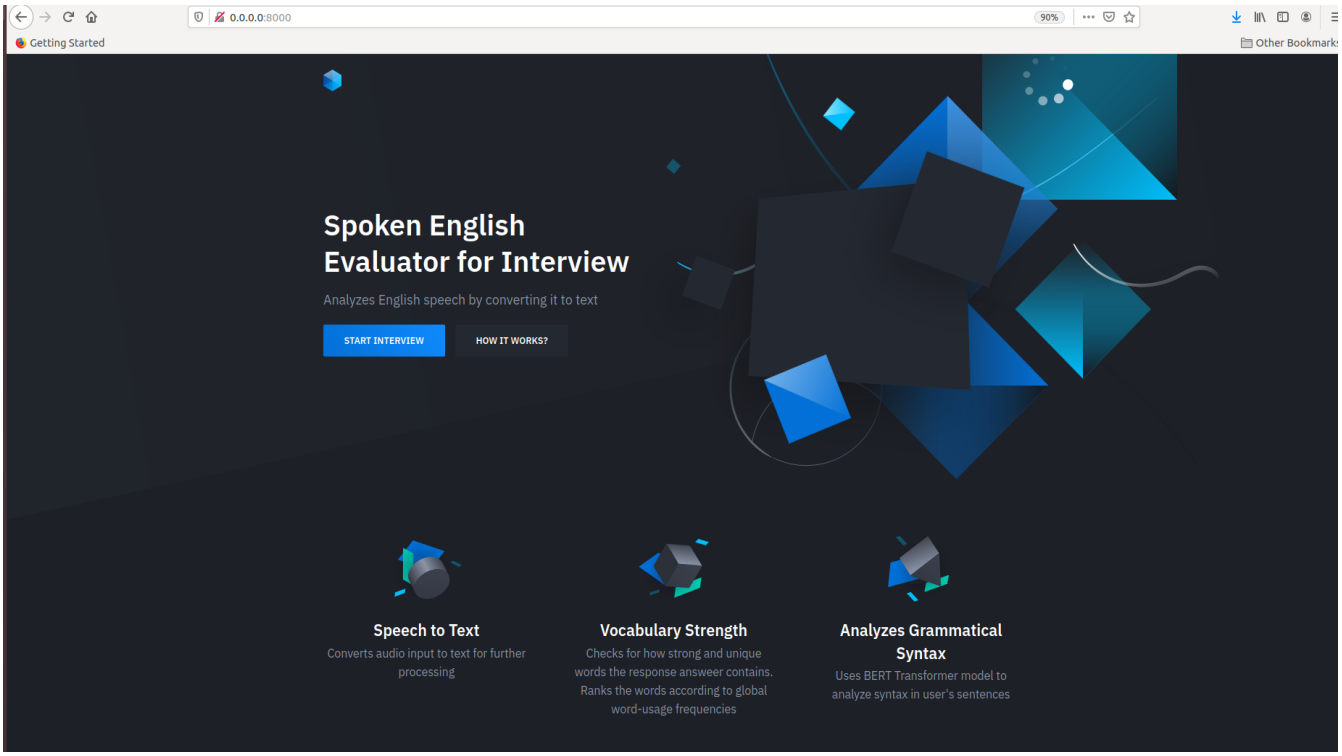


Figure [1] Home page of the web application

After clicking on “Start Interview” button, it takes the user to Record page where to question is displayed, and user has to click on “start recording” button to start the voice recording function. Figure [2] shows the Record web page before clicking the record button.

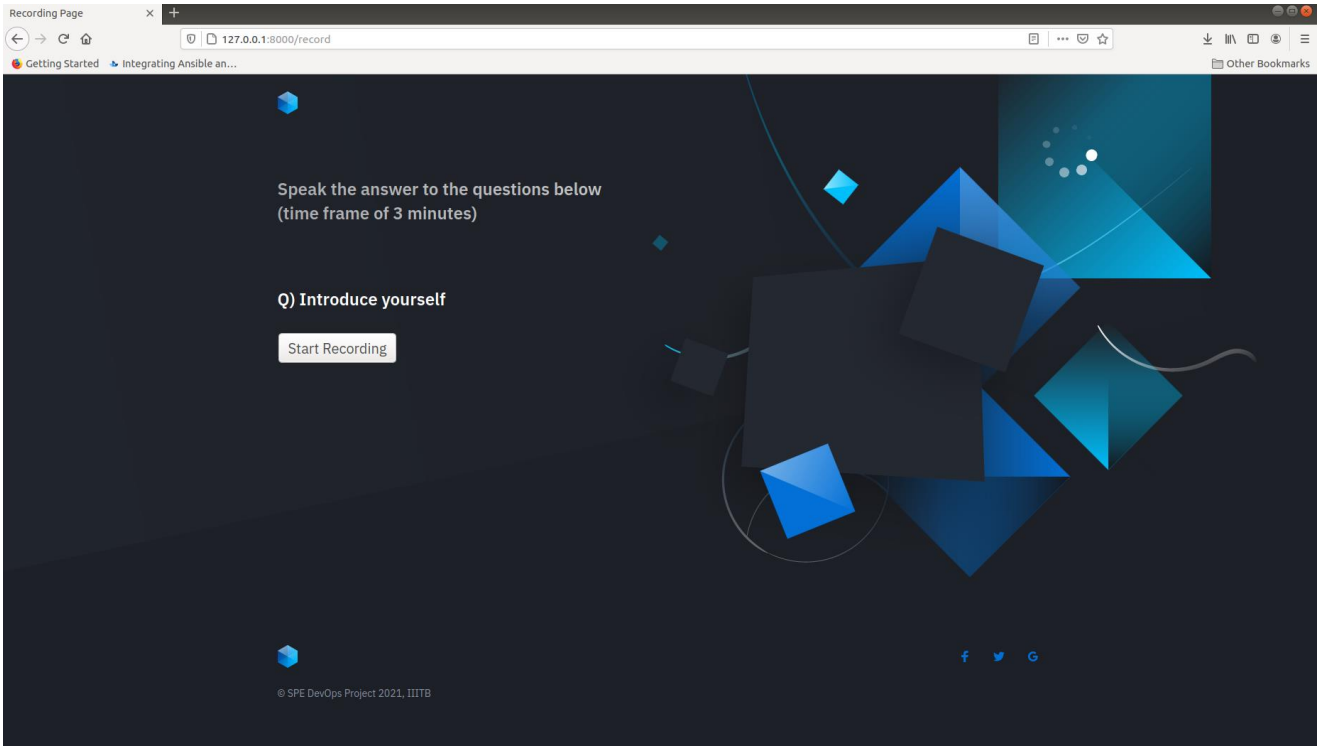


Figure [2] Record page before clicking the “start record” button (timer not started)

Next, when the user clicks the “Start record” button, the application calls the Speech Analyzer function using Django Backend. It also starts a timer which is displayed below the recording button. We have set the timer for question answering as 1 minute. So the application records the audio for the period of a minute. After the timer runs out, it displays “TIME UP” message. This functionality is programmed by using JavaScript. The code for it is show below in Figure [3]. The webpage after recording is started is shown in Figure [4].

```
<script>
    function startTimer() {
        document.getElementById("mybtn").value="Recording..";
        var timeleft = 60000;
        var setTimer = setInterval(function() {

            var minutes = Math.floor((timeleft % (1000 * 60 * 60)) / (1000 * 60));
            var seconds = Math.floor((timeleft % (1000 * 60)) / 1000);

            document.getElementById("mins").innerHTML = minutes + "m "
            document.getElementById("secs").innerHTML = seconds + "s "

            if (timeleft < 0) {
                clearInterval(setTimer);
                document.getElementById("mins").innerHTML = ""
                document.getElementById("secs").innerHTML = ""
                document.getElementById("end").innerHTML = "TIME UP!!";
            }
            timeleft = timeleft - 1000;
        }, 1000);
    }

</script>
```

Figure [3] Timer function code in JavaScript

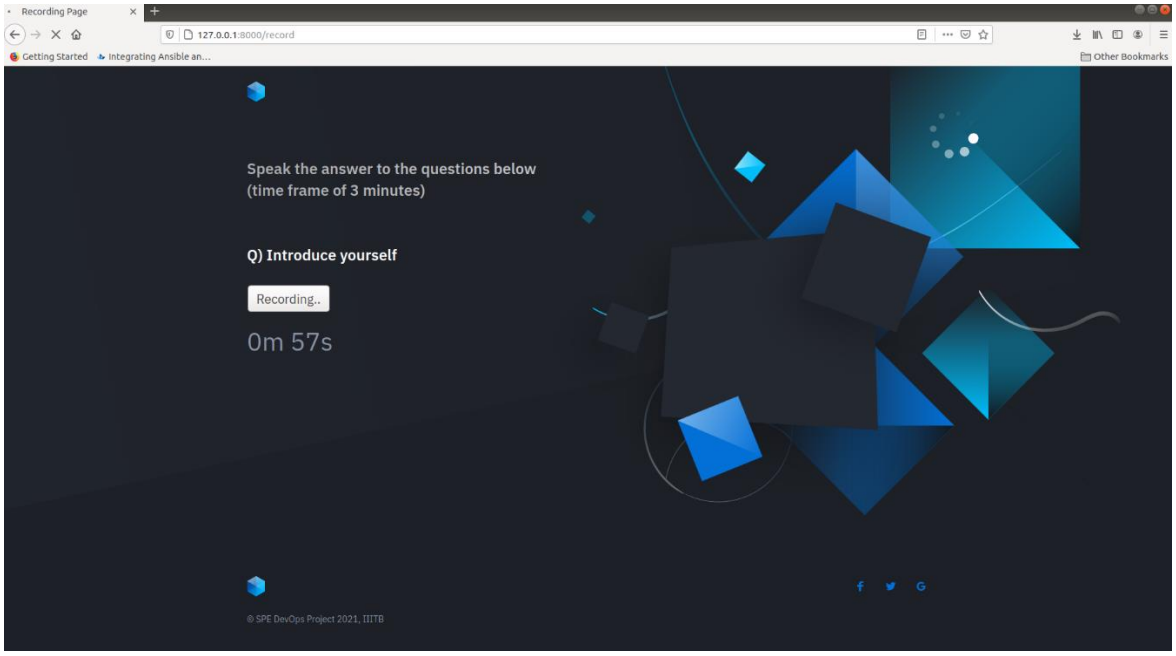


Figure [4] Recording webpage after the recording is started (Timer started)

We have also used JavaScript to change the text on recording button from “Start Recording” to “Recording..” to indicate to the user that recording is being done.

After the Analysis is complete by speech Analyzer function, it returns the output in the form of a list [sentence analysis, vocab-strength, unique words] to the Django view function in backend. Django takes this output and renders it in Result webpage in the form of html tables, which is dynamically filled and size adjusted as per the size of the output. Figure [5] shows an example output displayed after the analysis is carried out.

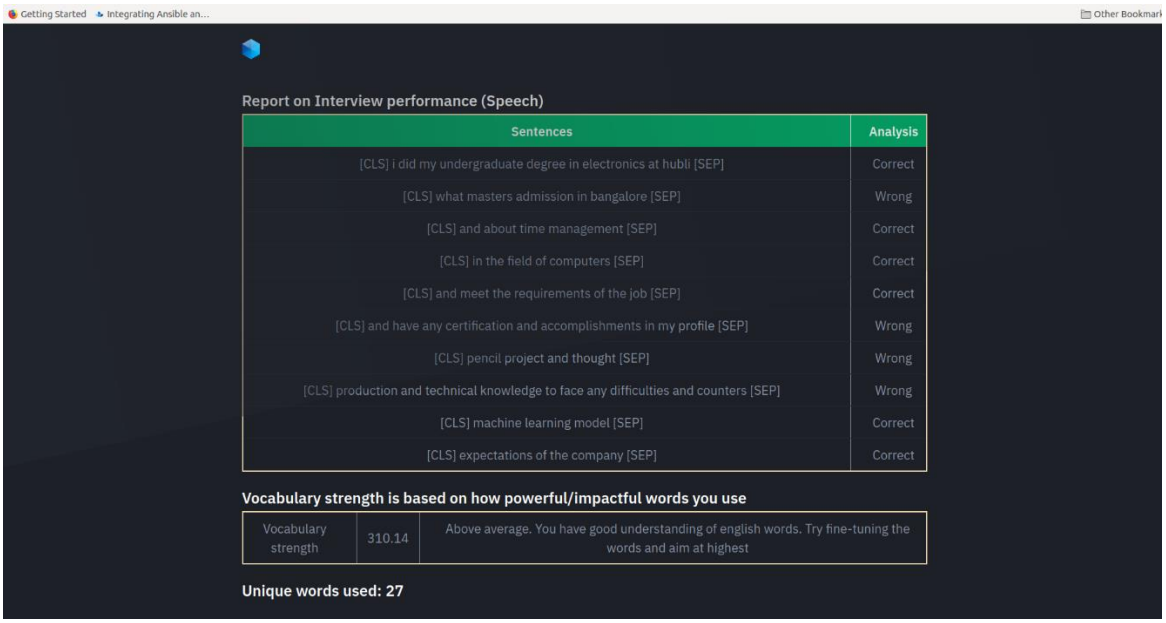


Figure [5] Result webpage after analysis is complete

Back-end Design

Django Framework renders webpages via view functions, and using “render” function which takes input parameters as request and the resultant webpage. In this web-application, the welcome page is rendered first whenever the application starts. Then using “URL” functionality, we are directing it to record webpage as shown in below Figure [6].

```
    <h1 class="hero-title mt-0">Spoken English Evaluator for Interview</h1>
    <p class="hero-paragraph">Analyzes English speech by converting it to text</p>
    <div class="hero-cta">
    <a class="button button-primary" onclick="location.href='{% url 'record' %}'">Start Interview</a>
    <a class="button" href="#">How it Works?</a>
</div>
```

Figure [6] Using URL method to trigger page rendering with onclick JS function

Next, to trigger execution of “Analyzer.py” which does all the speech analysis logic, we are using Django’s view functions. When the “Start Recording” button is clicked, we are using POST method of form to call the Analyzer.py program as shown below in Figure [6] and [7].

```
<h5 class="hero-cta">Q) Introduce yourself</h5><br>

<form method="post">
    {% csrf_token %}
    <input onclick="startTimer()" type="submit" value="Start Recording" name="run_analyzer" id="mybtn" />
    <!--
    <button id="mybtn" value="Start Recording" type="submit" name="run_analyzer" onclick="startTimer()"></button>
    -->
</form>
```

Figure [6] Form POST method to trigger Analyzer program

```
    if request.method == 'POST' and 'run_analyzer' in request.POST:
        res = analyze()

        if res is None:
            logger.error("Didnt receive result analysis")
        else:
            logger.error("Analysis successful")

        table_1_answers = []
        table_1_sentences = []
        for each in res[0]:
            table_1_answers.append(res[0][each])
            table_1_sentences.append(each)

        table_1 = []
        for i in range(len(table_1_sentences)):
            temp = {}
            temp['Sentence'] = table_1_sentences[i]
            temp['Analysis'] = table_1_answers[i]
            table_1.append(temp)

        if res[1] >= 500:
            vocab_analysis = "Excellent! Your knowledge of english vocabulary is impressive"
        elif res[1] < 500 and res[1] >= 300:
            vocab_analysis = "Above average. You have good understanding of english words. Try fine-tuning the words and aim at highest"
        elif res[1] < 300 and res[1] >= 150:
            vocab_analysis = "Average. Your knowledge of english vocabulary words is average. There is room for improvement"
        elif res[1] < 150 and res[1] >= 75:
            vocab_analysis = "Below average. Your vocabulary is sufficient for casual conversation, but not good for professional use"
        elif res[1] < 75 and res[1] >= 25:
            vocab_analysis = "Poor. You have rudimentary proficiency in english, and need to work hard"
        elif res[1] < 25:
            vocab_analysis = "You hardly know english vocabulary. Learn it before proceeding further"
        else:
            vocab_analysis = "Data not sufficient"


        data = { 'table_1':table_1, 'vocab_strength':res[1], 'vocab_analysis':vocab_analysis, 'unique_words':res[2] }

        return render(request, 'result.html', data )
```

Figure [7] View function to execute “Analyzer.py” program when POST method is received

The Django framework structure has files such as settings.py, urls.py, views.py which are vital on constructing the backend logic of a web-application. We shall explain them in brief. Settings.py contains important configuration instructions for the application such as directory path information, allowed hosts info, installed apps, template and static files paths, database information (if any), language settings, logger setups etc. “urls.py” contain webpage urls to which the Django can redirect when called. Its shown in Figure [8] below.

```python
from django.urls import path
from . import views


urlpatterns = [
    path('', views.index, name='index'),
    path('record', views.record, name='record'),
    path('result', views.result, name='result')
]
```

Figure [8] urls.py file which contains webpage url info

“Views.py” contains the backend logic which we can implement on each web-page elements. Here we have used this to execute our Speech Analyzer function when recording button is clicked as shown in Figure [7]. In Result page, we are constructing a table dynamically according to the size of output sentences. This is done with for loop in Django as shown in Figure [9].

```
        <div class="container">
            <h4> Report on Interview performance (Speech) </h4>
            <table border="2" cellpadding="5" cellspacing="5">
                <th>Sentences</th>
                <th>Analysis</th>
                {% for i in table_1 %}
                    <tr>
                        <td>{{i.Sentence}}</td>
                        <td>{{i.Analysis}}</td>
                    </tr>
                {% endfor %}
            </table>

            <h5>Vocabulary strength is based on how powerful/impactful words you use</h5>
            <table border="2" cellpadding="5" cellspacing="5">
                <td>Vocabulary strength</td>
                <td>{{ vocab_strength}}</td>
                <td>{{ vocab_analysis }}</td>
            </table>
            <h5>Unique words used: {{ unique_words}}</h5>
        </div>
    </section>
</main>
```

Figure [9] Populating Dynamic table with output

Speech Analyzer Program Design

In this section, we discuss about how speech analyzer works with screenshots of relevant code. We made use of ML models to perform these analysis task. This concept comes under the field of Natural Language Processing (NLP) of ML. Here we make use of a NLP model called 'BERT' which is a transformer model developed on enormous corpus of text. We take this model and fine-tune it our syntax analysis task.

Speech Record and text Conversion

We are doing the audio recording using "speech_recognition" module, which will use the system's microphone to listen to audio signals, then for converting to that speech to text we are using "recognize_google" function provided by speech_recognition module. We have set the timer to one minute for it to record. The code for this is shown in Figure [10]

```
r = sr.Recognizer()
sentences = []
inp_corpus = ""
print("Starting voice recording")

while time.time() < t_end:
    try:
        with sr.Microphone() as source2:
            r.adjust_for_ambient_noise(source2, duration=1)
            audio2 = r.listen(source2)
            MyText = r.recognize_google(audio2)
            MyText = MyText.lower()
            inp_corpus += MyText
            sentences.append(MyText)

    except sr.RequestError as e:
        print("Could not request results; {0}".format(e))

    except sr.UnknownValueError:
        print("unknown error occured")
```

Figure [10] Audio recording and text conversion code

Vocabulary Strength Calculation

Upon the text which speech-to-text converter generated, and after performing some filtering operations such as bad characters removal, we are performing vocabulary strength analysis, which considers word frequencies from global English dictionary http://norvig.com/mayzner.html. This score is calculated by taking inverse of their frequency, thereby giving reasonably less score for most commonly used words. We have also compiled some words which are relevant to the question asked and gave scores based on how relevant are

those to the question asked. The python code which makes use of dict datastructure, calculates vocabulary scores by considering these two aspects is shown in Figure [11].

```
    ## Pre-processing
    bad_chars = [';', ':', '!', "*", ",", "."]

    inp_corpus = ''.join(i for i in inp_corpus if not i in bad_chars)
    inp_corpus = inp_corpus.replace('&', 'and')
    inp_corpus = inp_corpus.replace('@', 'at')
    inp_corpus = inp_corpus.lower()
    corpus_list = list(inp_corpus.split(' '))

    ## setting weightage parameters for scores
    alpha = 0.4
    beta = 0.6

    ## textfile contatining words and their usage frequency
    # script_dir = Path(__file__).parent
    f = open(DEPENDENCIES_DICT_DIR, "r")
    dictonary = dict()
    list_lines = f.readlines()

    ## creating word-frequency dictonary out of words-text file
    rank = 1
    for line in list_lines:
        li = list(line.split(' '))
        word, frequency = li[0], li[2]
        dictonary[word] = rank
        rank += 1

    for each in dictonary.keys():
        temp = (dictonary[each])
        scaled = (temp - 1) * 99 / (len(dictonary) - 1)
        scaled += 1
        dictonary[each] = scaled

    visited = []
    score_1 = 0
    unique = 0
    for word in corpus_list:
        if word in dictonary.keys():
            if word not in visited:
                score_1 = score_1 + dictonary[word]
                visited.append(word)
                unique += 1

    ## calculating score_2 with tag ranks
    score_2 = 0
    db = pd.read_csv(Q1_CSV)
    db_dict = dict(db.values)
    for word in corpus_list:
        if word in db_dict.keys():
            if word not in visited:
                score_2 = score_2 + db_dict[word]
                visited.append(word)
                unique += 1

    strength = (alpha * score_1) + (beta * score_2)
    strength = float("{:.2f}".format(strength))
    f.close()
```

Figure [11] Vocabulary strength analysis

Here in this code, “Dependencies_dict_dir” is the path for global word count file. And “Q1.csv” is an excel file in which we had compiled words which are relevant to the question asked. We have set alpha and beta as 0.4 and 0.6, which are weightages given to the score calculated from above two files. The final score is the weighted mean of both the scores, which are calculated from above said files.

Syntax Analysis of Sentences

The text corpus output from speech recorder is broken into individual sentences for syntax analysis on them. To perform syntax analysis, we are using NLP BERT model, which is trained on enormous amount of text data beforehand. We fine-tune it to analyze syntax of each sentence and provide output on whether they are grammatically correct or not. Figure [12] shows the code

```
print("Starting syntax analysis")

device = "cpu"
if torch.cuda.is_available():
    torch.cuda.empty_cache()
gc.collect()
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
model.load_state_dict(torch.load(BERT_MODEL, map_location='cpu'))
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)

model.eval()
sentences = ["[CLS] " + sentence + " [SEP]" for sentence in sentences]
labels = [0]
tokenized_texts = [tokenizer.tokenize(sent) for sent in sentences]
MAX_LEN = 128
predictions = []
true_labels = []
input_ids = pad_sequences([tokenizer.convert_tokens_to_ids(txt) for txt in tokenized_texts], maxlen=MAX_LEN,
                          dtype="long", truncating="post", padding="post")
input_ids = [tokenizer.convert_tokens_to_ids(x) for x in tokenized_texts]
input_ids = pad_sequences(input_ids, maxlen=MAX_LEN, dtype="long", truncating="post", padding="post")

attention_masks = []
for seq in input_ids:
    seq_mask = [float(i > 0) for i in seq]
    attention_masks.append(seq_mask)

prediction_inputs = torch.tensor(input_ids)
prediction_masks = torch.tensor(attention_masks)
prediction_labels = torch.tensor(labels)
model.eval()

with torch.no_grad():
    logits = model(prediction_inputs.to(device).long(), token_type_ids=None,
                   attention_mask=prediction_masks.to(device).long())
logits = logits.detach().cpu().numpy()
predictions.append(logits)

flat_predictions = [item for sublist in predictions for item in sublist]
flat_predictions = np.argmax(flat_predictions, axis=1).flatten()
flat_true_labels = [item for sublist in true_labels for item in sublist]

result = {}
for i in range(len(flat_predictions)):
    if flat_predictions[i] == 1:
        result[sentences[i]] = 'Correct'
    elif flat_predictions[i] == 0:
        result[sentences[i]] = 'Wrong'
```

Figure [12] Syntax analysis code used for sentences

Here, we will explain the code in brief as follows: For training, sentences are taken from cola public dataset and are inserted with tokens CLS and SEP at beginning and end to make it work for BERT. These will function as sentence separators. We tokenize our sentences by making use of bert tokenizer using a base pre-trained model. To make each sentence length equal, we have set the length of sentence as 128 and padded remaining places with null strings. The main

concept of BERT is attention mask, which will tell on which values you have to focus and which not. The data is split into train and test datasets, and then tensors are constructed from it, which can be used for training with pytorch. We also define our data-loader which helps in loading data in batches, so that computational resource requirement is less. We get weights for various layers when we train this model. Once that is done, we separate weight parameters from bias, gamma, and beta parameters. We filter one group without these values and another with them. We had trained the above model with 4 epochs and used gradient forward pass to calculate the loss function. After this, we save the trained model and use it for syntax prediction on sentence inputs. Here in this project, we had trained the model separately and saved it in a directory to use it when necessary. The model size after training was about 420 MB, so we had to use github LFS storage to put it in out Github repo. Training the model while deploying it is impractical, so we had to train it beforehand, and then only use the saved model in the web-application.

DevOps CI/CD Pipeline

This section explains about the Software development method we had followed while developing this web-application.

Source Code Management

In CI/CD pipeline it is important to maintain versions of our source code, it will help us to keep track and revert back to any previous version when required. We are using Git /GitHub version control system for this project. Since one of the file (BERT model) was > 400MB size, we had to use GitHub LFS (Large File storage) which can track large files and store them in server.
The development of this application has been done incrementally. The commits done can be seen at https://github.com/akshaymg99/SPE_Speech_Evaluator/commits/master .

After developing code in an IDE, to post it to SCM (Git), we have to execute the following commands in our project working directory:

`git remote add origin <remote repo url>` // create connection to remote repo

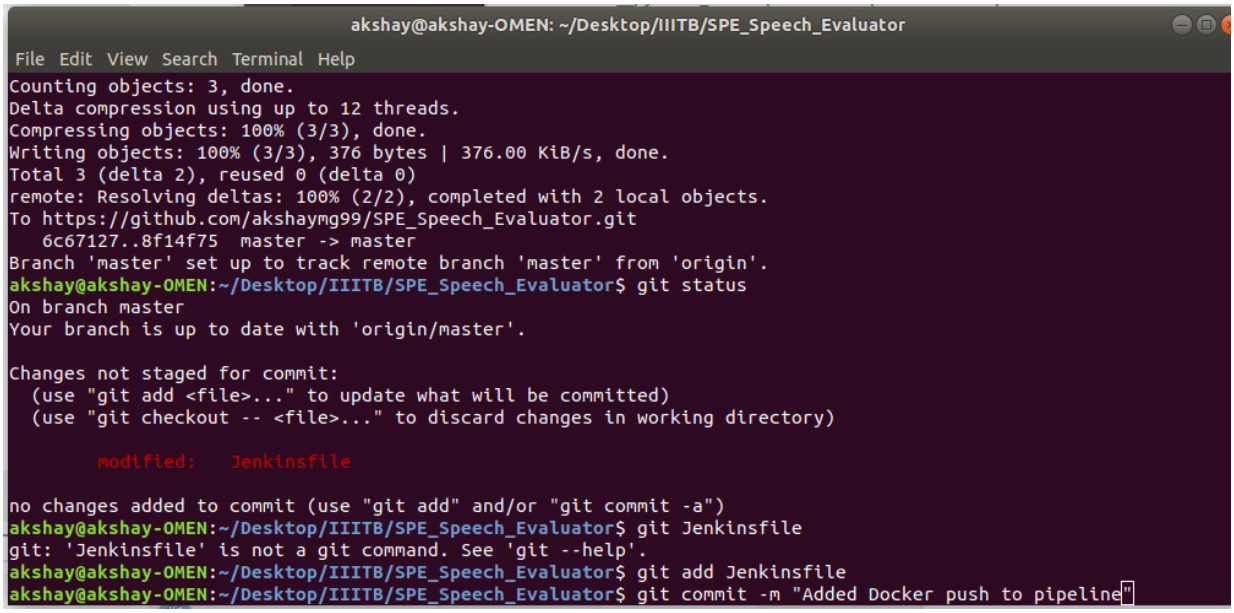
`git init` // for initializing local system directory as git directory

`git add . or git add /files` // for adding files to staging area

`git commit –m "message"` // to take a snapshot of the current state of directory

`git pull <remote repo url>` // to synchronize remote repository with local repo

`git push u origin <branch>` // to push the local repo files to remote repository



Other git commands:

`git revert <commit>` // undoes all changes made in <commit>

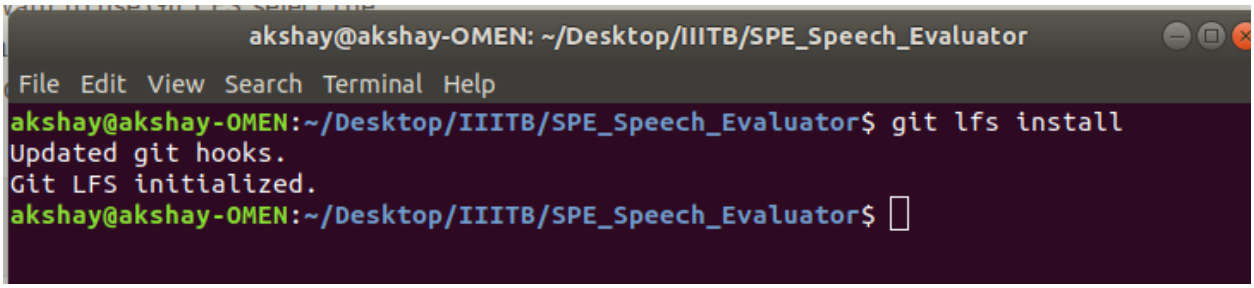
`git status` // list which files are staged, unstaged & untracked

`git log` // display commit history

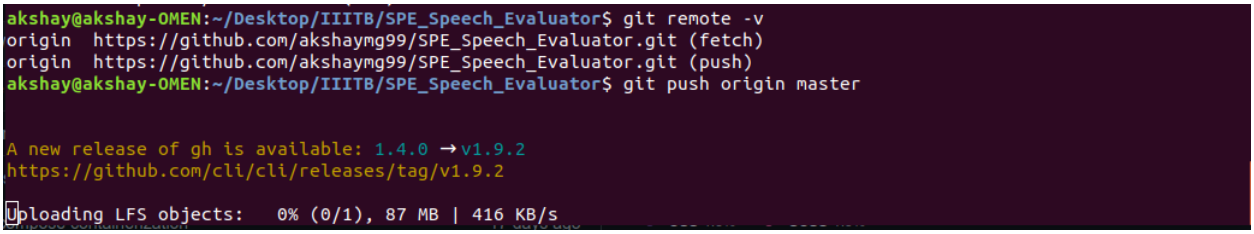
`git checkout b <branch>` // create & checkout a new branch

`git merge <branch>` // merge <branch> into current branch

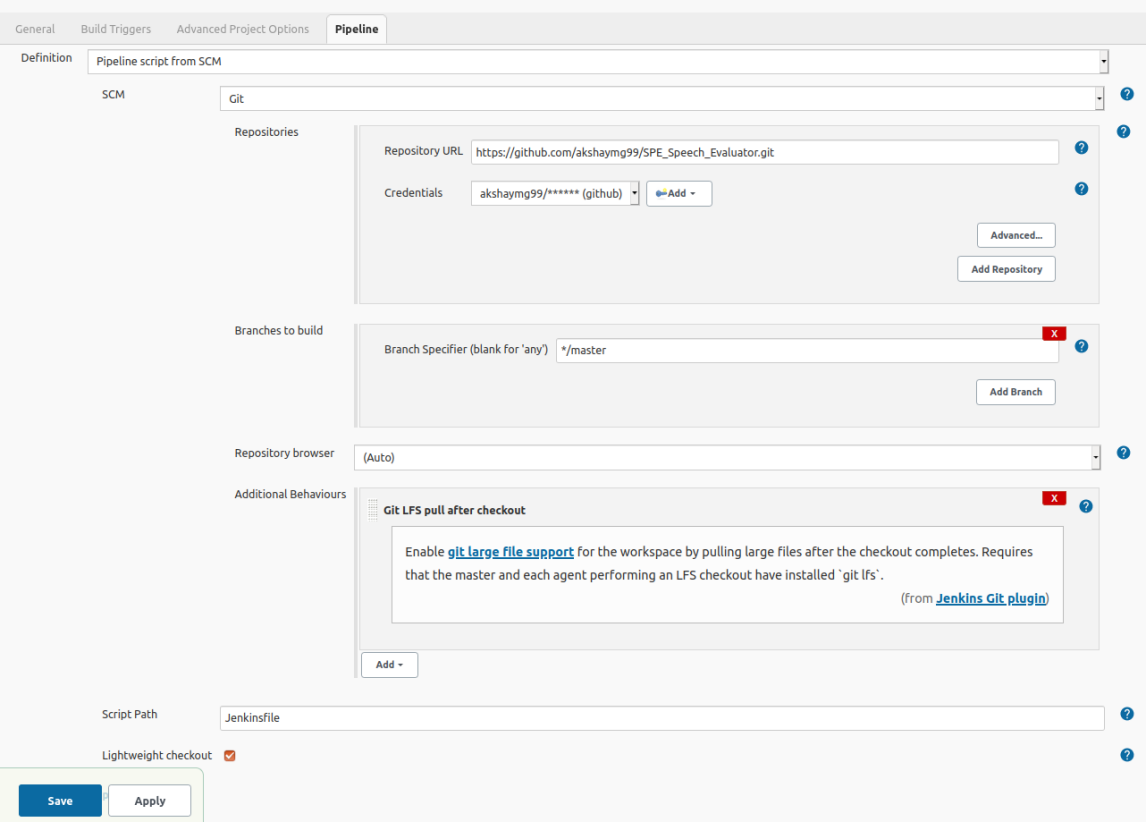
We are also using Git LFS for this project to store large model in repository. We have to install it separately in our system as shown below and also have to enable Git LFS pipeline in Jenkins.



Git LFS pushing the object to its repo:



Configuring Git LFS in Jenkins Job:



Jenkins pulling the git LFS object

Build Stage

Build stage consists of installing dependencies and making the application ready to be complied/ installed. We used `pip3 freeze > requirements.txt` command to export the dependency requirements to a text file, which we are using it to install in Docker container later in this project. We are using pip3 install -r requirements.txt command in Dockerfile to install all the requirements in docker build stage. Below Figure [13] shows the dependencies getting installed while running Jenkins job.

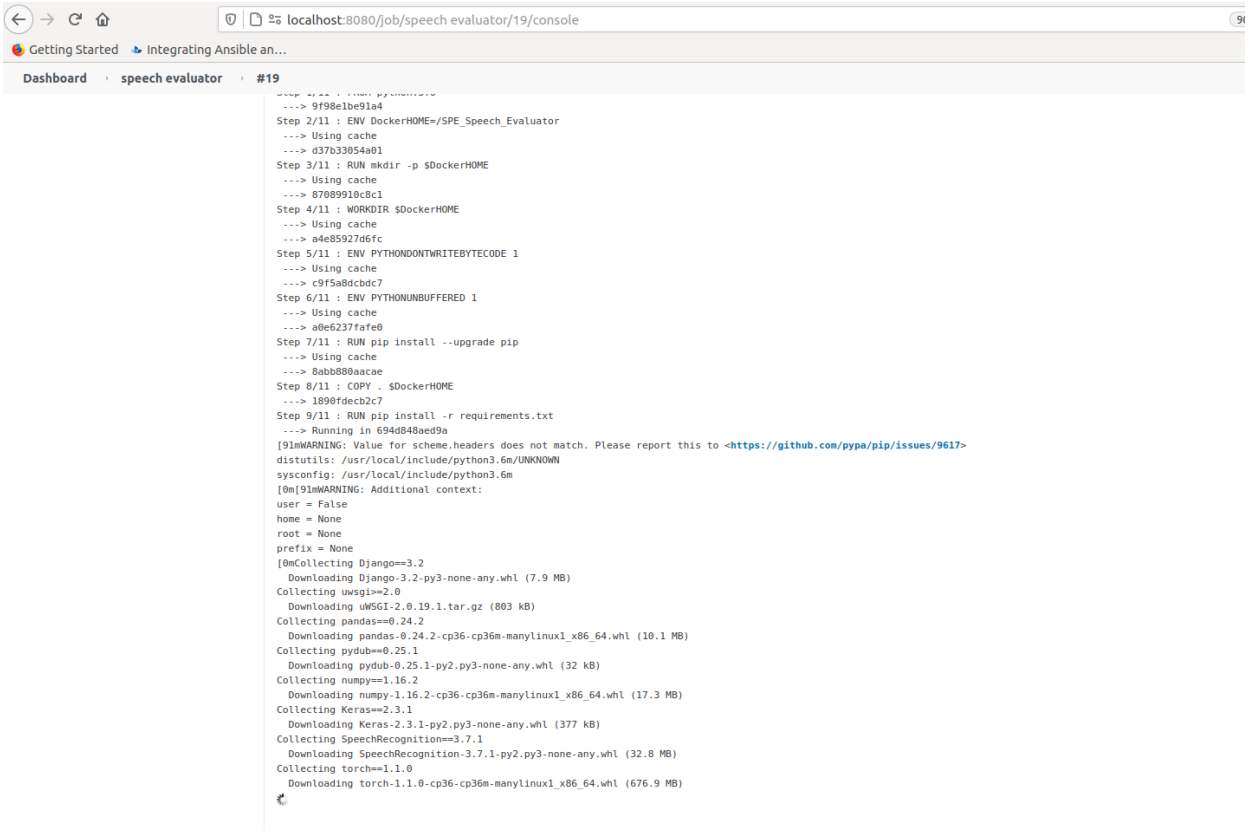


Figure [13] Dependency installation through requirements file

Testing stage

Testing in required to find any bugs or errors during application run, so developers write unit test cases to check the operations of the program. We are using Django's inbuilt test framework for testing the web-application. We are performing 3 types of tests for our application – webpage retrieval tests, csrf token check which helps in blocking unauthorized access, and lastly a logic test in which we are providing text-corpus as input and checking how will it output the result webpage. Figure [14] shows the test code which we consist of the test types we talked earlier.

```python
from django.test import Client
from django.test import TestCase
import unittest

class TestApp(TestCase):
    ##  Checking if all webpages are retrieved successfully (status code-> 200 should specify the success)
    def test_home_page(self):
        self.c = Client()
        csrf_client = Client(enforce_csrf_checks=True)  ## enforcing CSRF Checks
        self.response = self.c.get('')          ## home page retrieval
        print(self.response.status_code)        ## output = 200 specifies the page was retrieved successfully
        self.assertEqual(self.response.status_code, 200, msg='Home page retrieved successfully')

    def test_record_page(self):
        self.c = Client()
        csrf_client = Client(enforce_csrf_checks=True)  ## enforcing CSRF Checks
        self.response = self.c.get('/record')    ## record.html page retrieval
        print(self.response.status_code)         ## output = 200 specifies the page was retrieved successfully
        self.assertEqual(self.response.status_code, 200, msg='Recording page retrieved successfully')

    def test_result_page(self):
        self.c = Client()
        csrf_client = Client(enforce_csrf_checks=True)  ## enforcing CSRF Checks
        self.response = self.c.get('/result')     ## result.html page retrieval
        print(self.response.status_code)          ## output = 200 specifies the page was retrieved successfully
        self.assertEqual(self.response.status_code, 200, msg='Result page retrieved successfully')
        print(self.response.content)                ## checking content of the webpage

    def test_Analysis_logic(self):
        self.c = Client()
        self.table_1 = []
        self.temp = {}
        self.temp['Sentence'] = "Hi, My name is Akshay, I am pursing Masters from IIITB"
        self.temp['Analysis'] = "Correct"
        self.table_1.append(self.temp)
        self.res = []
        self.res.append(0)
        self.res.append(340)
        self.res.append(50)
        self.vocab_analysis = "Average. Your knowledge of english vocabulary words is average. There is room for improvement"
        self.data = {'table_1': self.table_1, 'vocab_strength': self.res[1], 'vocab_analysis': self.vocab_analysis, 'unique_words': self.res[2]}
        self.response = self.c.post('/record', self.data)
        print(self.response.status_code)
```

Figure [14] Testing code for the web-application

Here, we are performing CSRF check in each webpage and later a logic check on result page output. Figure [15] shows the output of test case running. Status code 200 means successful page retrieval and the html content is the output of result page on providing text corpus input to it.

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
200
.200
.200
.200
b'\n<!DOCTYPE html>\n<html lang="en" class="no-js">\n<head>\n    <meta charset="utf-8">\n    <meta http-equiv="X-UA-Compatible" content="IE=edge">\n    <meta name="viewport" content="width=device-width, initial-scale=1">\n
    <title>Recording Page</title>\n    <link href="https://fonts.googleapis.com/css?family=IBM+Plex+Sans:400,600" rel="stylesheet">\n    <link rel="stylesheet" href=/static/dist/css/style.css >\n\t<script src="https://unpkg.
com/animejs@3.0.1/lib/anime.min.js"></script>\n    <script src="https://unpkg.com/scrollreveal@4.0.0/dist/scrollreveal.min.js"></script>\n    <style>\n        table{\n            border: 2px solid wheat;\n            paddin
g: 15px;\n\n        }\n        td {\n            text-align: center;\n        }\n        th {\n            color: white;\n            text-align: center;\n            background-color: #80ab66;\n        }\n\n\n        p {\n
        display: inline;\n            font-size: 40px;\n            margin-top: 0px;\n        }\n    </style>\n</head>\n\n<body class="is-boxed has-animations">\n    <div class="body-wrap">\n        <header class="site-header
">\n            <div class="container">\n                <div class="site-header-inner">\n                    <div class="brand header-brand">\n                        <h1 class="m-0">\n\t\t\t\t\t\t\t<a href="#">\n\t\t\t\t\t\t\t\
----------------------------------------------------------------------
Ran 4 tests in 0.013s

OK
Destroying test database for alias 'default'...
```

Figure [15] Testing output

Here, it shows the summary as Ran 4 test cases and there were no failures. To run the tests, we are using “python manage.py test” command in Django application’s root directory.

Continuous Integration

Continuous Integration (CI) refers to the process of integrating code changes with the existing code as and when it is written. We are using Jenkins tool for Integration all the stages of DevOps. It automates from git pull whenever there is new code push to deployment of application. After installing Jenkins, we also need to add credentials of git and Dockerhub which we need to access later. Figure [16] shows adding of git credentials in Jenkins.

Figure [16] Adding Git credentials into Jenkins

Jenkins pipeline

A Jenkins pipeline gives us a graphical view of the various steps of a CI/CD pipeline. It allows us to link different Jenkins jobs and allows us to check their progress during execution. Here in this project, the first job consists of 4 stages: SCM checkout, SCM git pull, Docker build and Dockerhub push. After successfully completing these stages, Jenkins triggers another job, which is a Rundeck job, it pulls the latest docker image from hub, and deploys it in system, here the deployment system is our local system itself. This way Jenkins handles all the tasks automatically from git pull to triggering deployment task. Below Figure [17] shows the Jenkinsfile for our project. We can see different stages named for their operations. Figure [18] shows the Visualization of pipeline stages, which helps in developers understanding how each stage is being executed and what amount of time does each stage take.

```
pipeline{
    agent any

    stages{

        stage('SCM git pull'){
            steps{
                git credentialsId: 'github',
                url: 'https://github.com/akshaymg99/SPE_Speech_Evaluator.git'
            }
        }

        stage('Docker Build'){
            steps{
                sh "docker build . -t akshaymg99/speech-spe:latest "
            }
        }

        stage('DockerHub Push'){
            steps{
                withCredentials([string(credentialsId: 'docker-hub', variable: 'dockerHubPwd')]) {
                    sh "docker login -u akshaymg99 -p ${dockerHubPwd}"
                }
                sh "docker push akshaymg99/speech-spe:latest "
            }

        }


    }

}
```

Figure [17] Jenkins pipeline linking various stages of development

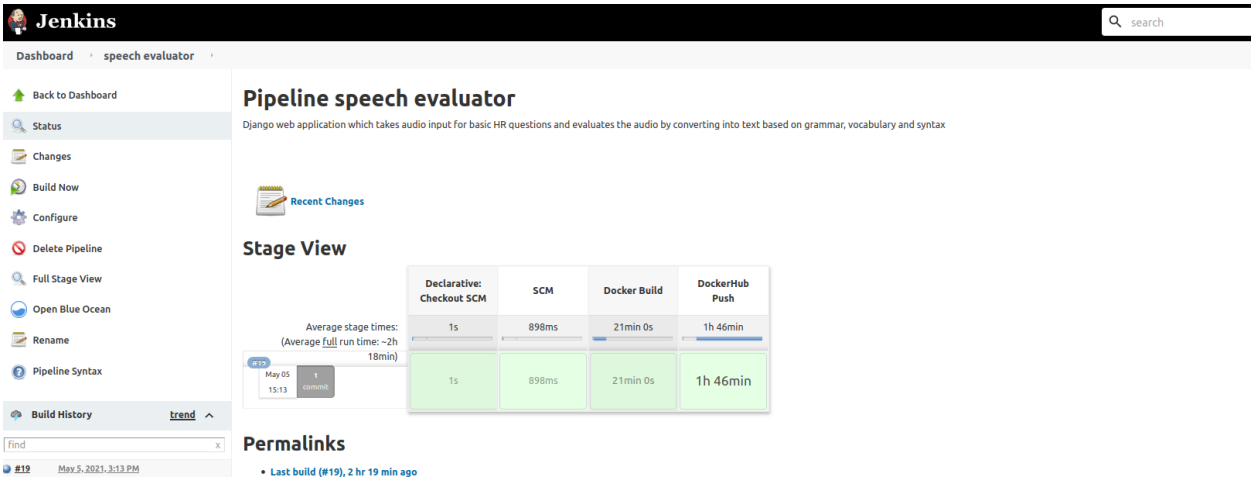


Figure [18] Jenkins pipeline visualization

We are also using Jenkins to trigger the deployment task which is a Rundeck Job. We will explain this Rundeck job later in detail in Deployment part. After configuring a Rundeck Job, it gives a unique UUID key, which we should use in Jenkins to link that task with Jenkins. After linking, we need to setup a build trigger which after successful completion of previous job (in which we have pushed the docker image to hub) it starts the Rundeck job. Figure [19] shows configuring Jenkins with Rundeck job with UUID key.

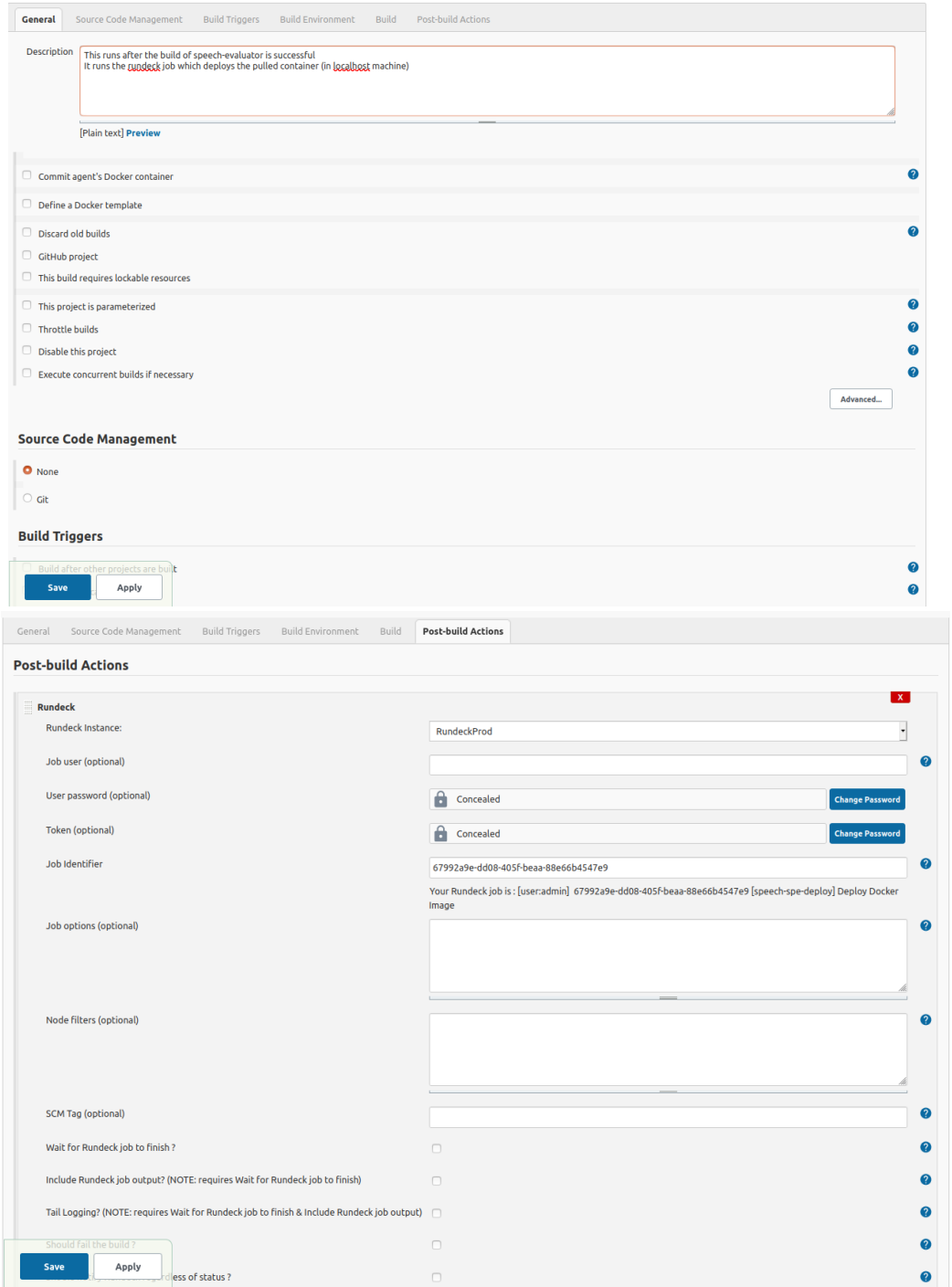


Figure [19] Configuring Jenkins with Rundeck UUID key to trigger it

We can also see that the successful completion of “speech_evaluator” job in Jenkins which consists of stages git pull to docker hub push triggers “deploy speech” job of Rundeck in Jenkins. Figure [20] shows successful trigger of Rundeck deploy job in console.

Figure [20] Jenkins triggering Rundeck deploy job

Figure [21] shows “deploy speech” running successfully in which it had executed the Rundeck job. More on Rundeck deployment will be explained in Deployment part of this report.

Figure [21] Jenkins executing Rundeck job

Following Figures [22], [23], [24], [25], [26], [27] shows Jenkins console output for running “speech evaluator” task (which has stages from git pull to docker hub push).

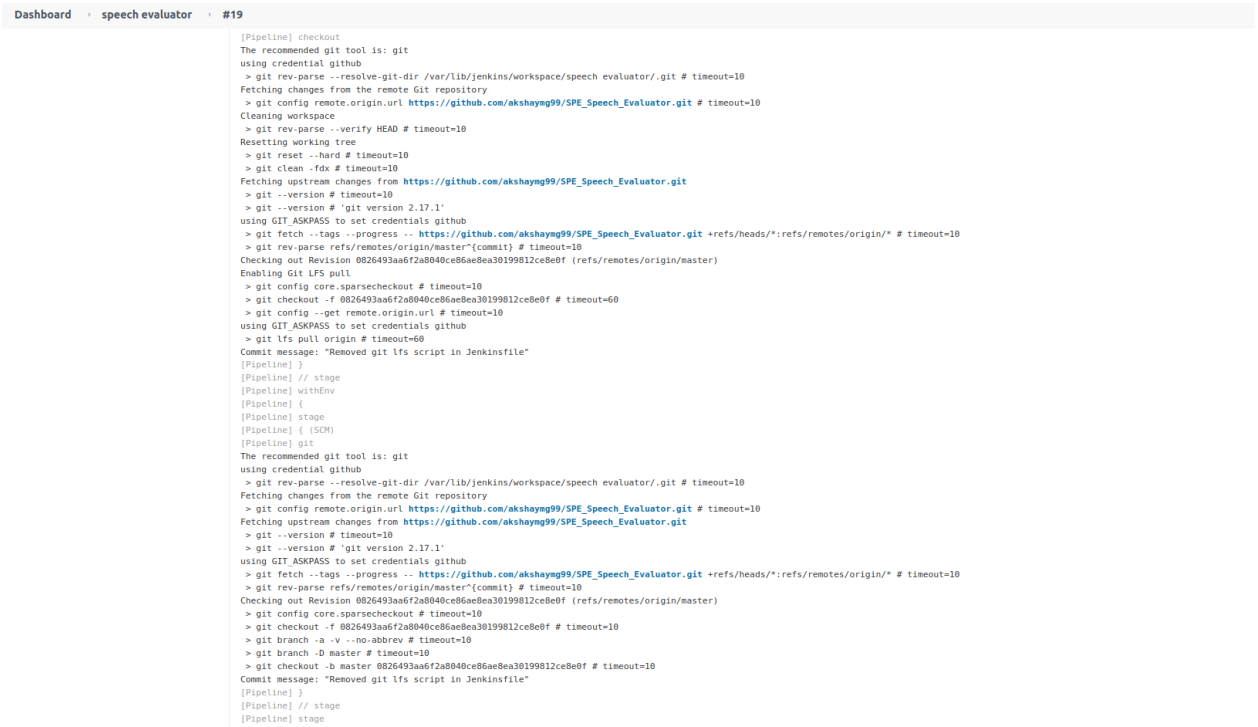


Figure [22] git checkout stage in Jenkins, note that it also uses git LFS

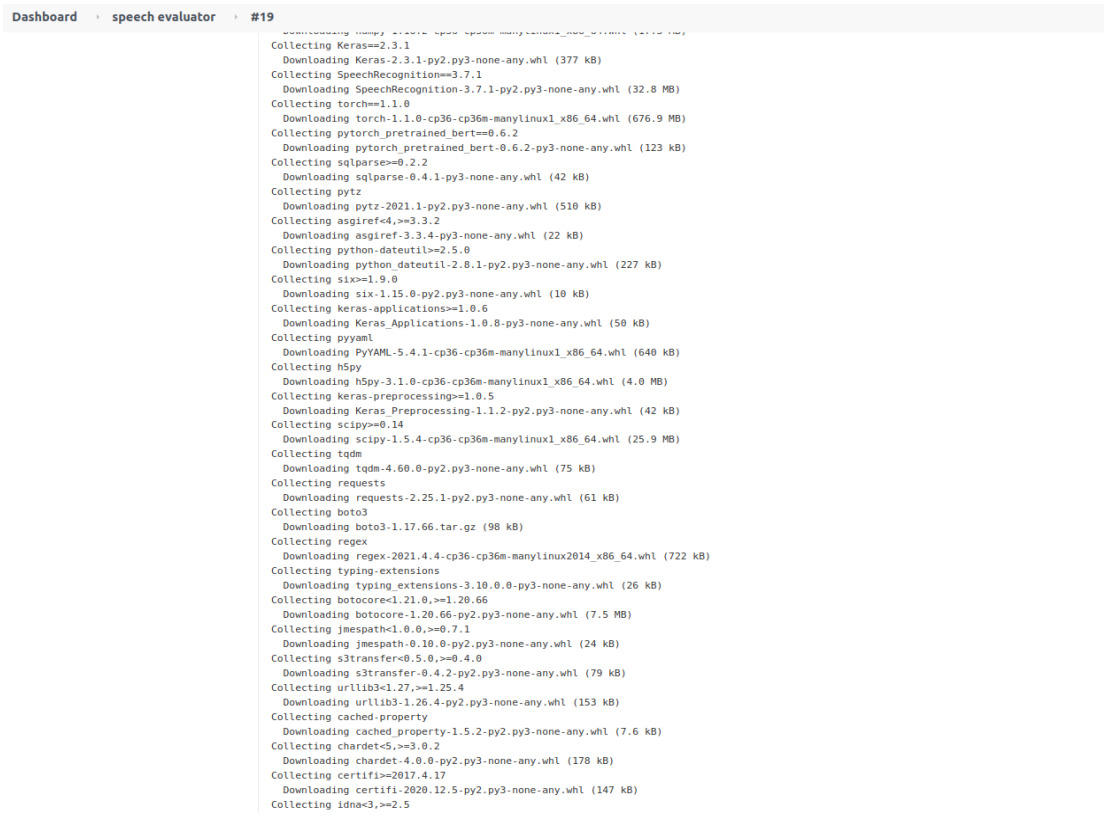


Figure [23] Installing requirements / dependencies from requirements.txt file

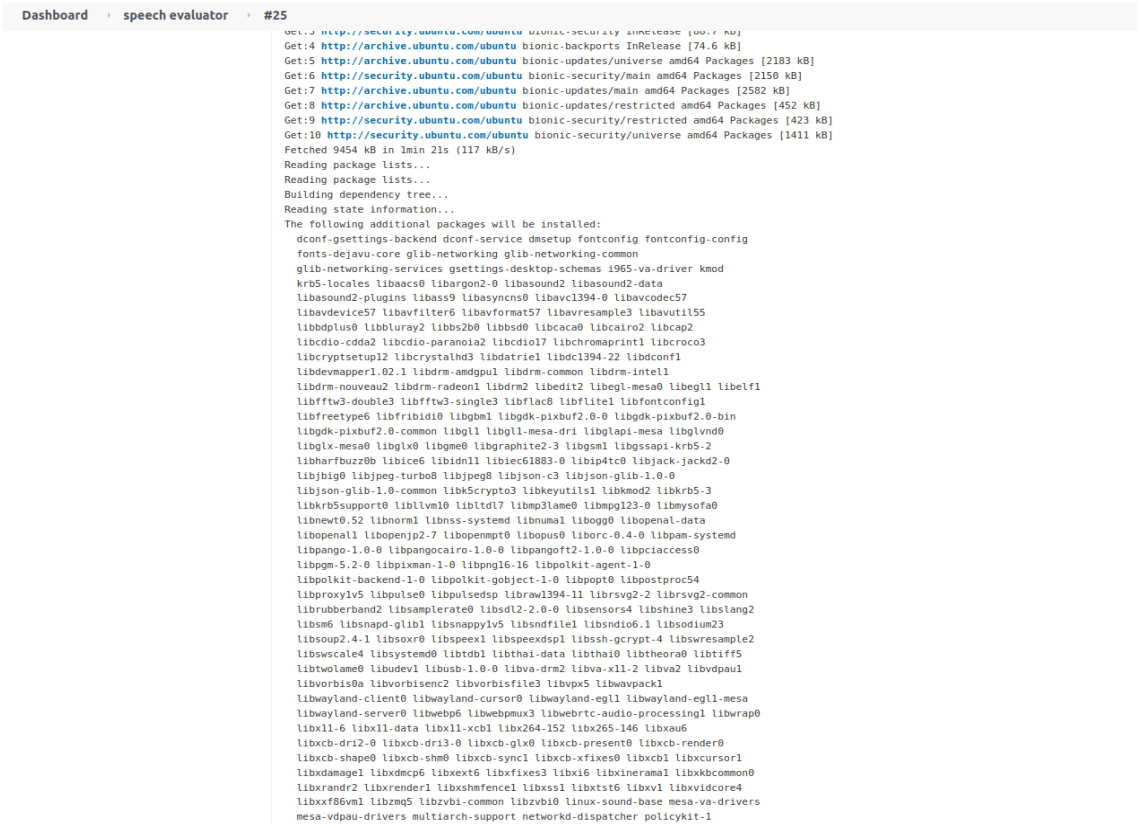


Figure [24] Installing sound drivers required for microphone usage for recording audio

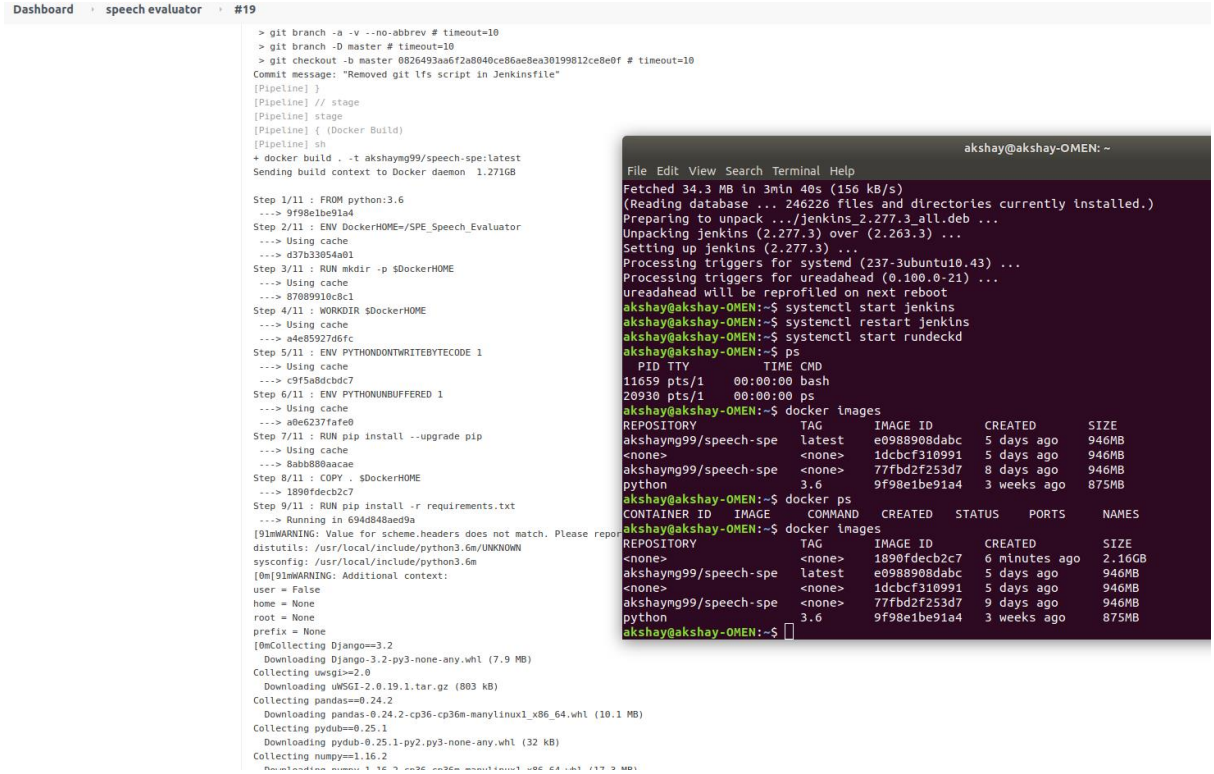


Figure [25] Docker image is built

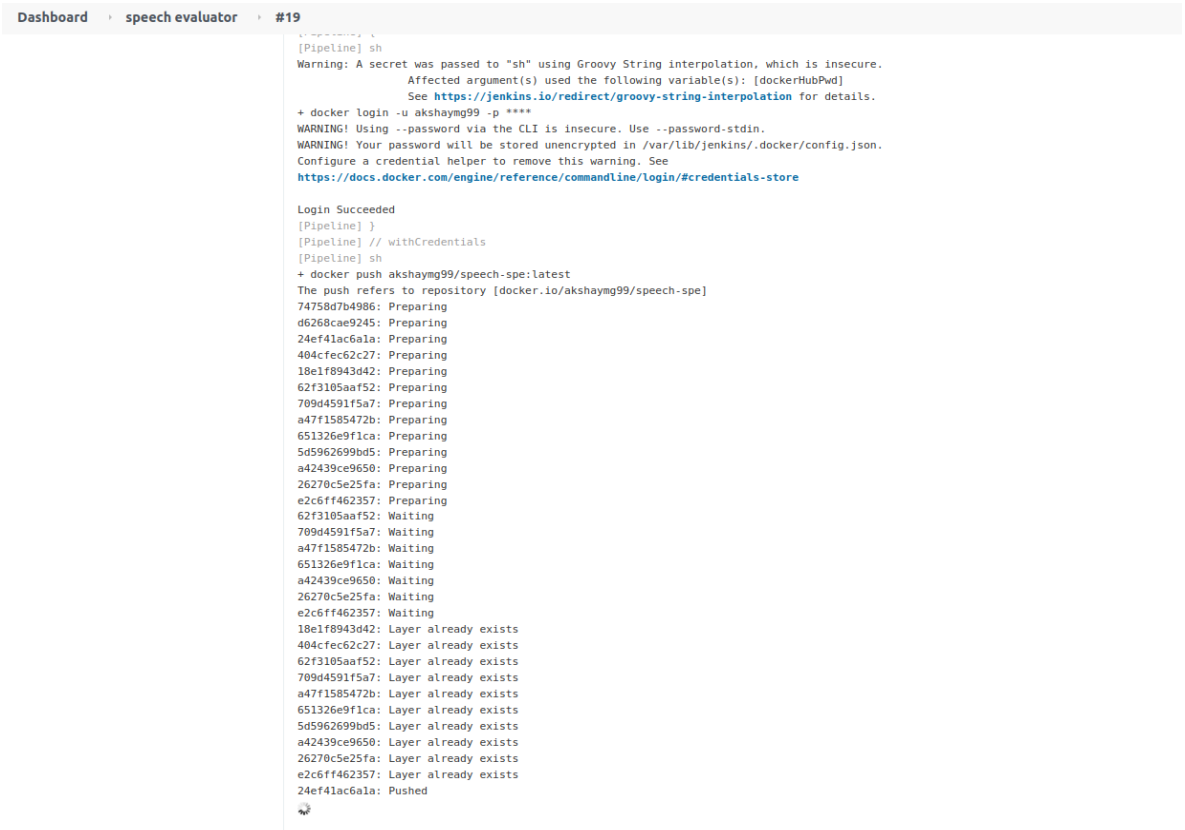


Figure [26] Docker image pushed to hub

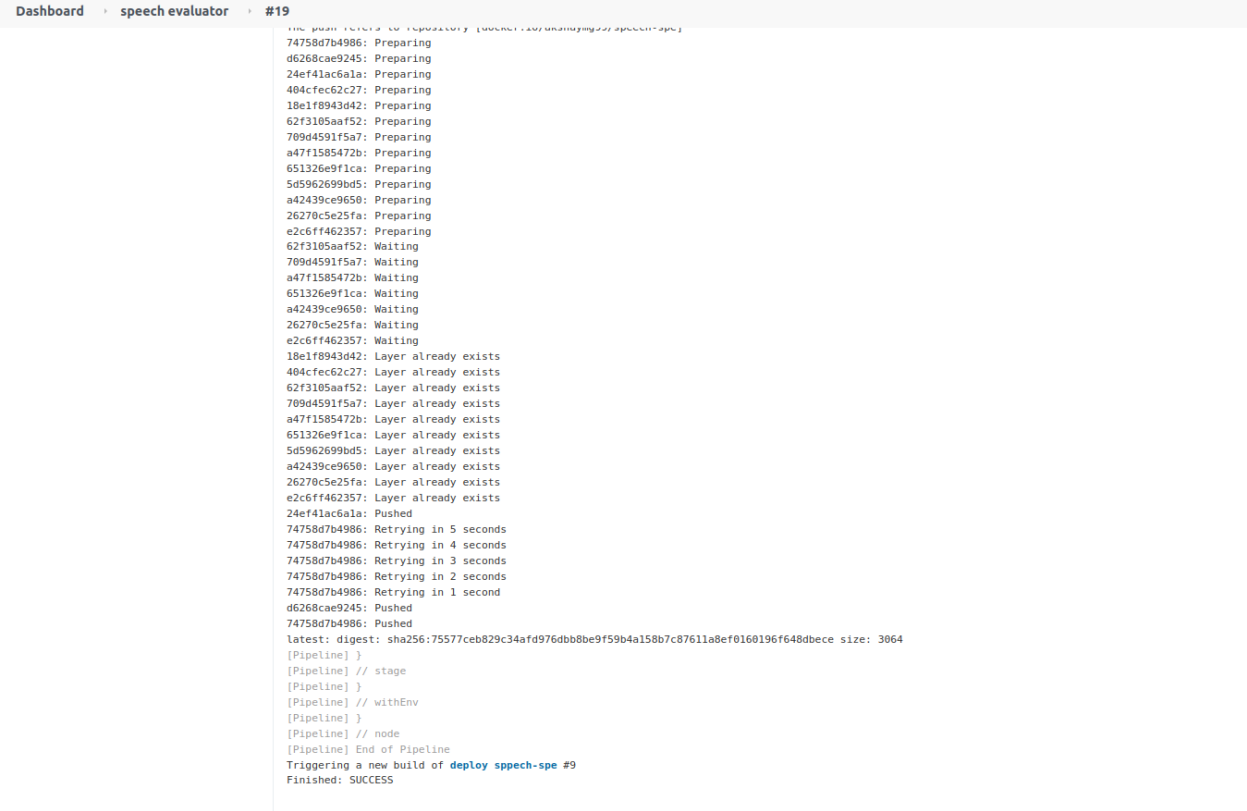


Figure [27] Build success of “speech_evaluator” job

Continuous Delivery

A deliverable in Software Engineering is an artifact that is ready to be delivered to the client. It thus is the end product of a SDLC. Continuous Delivery (CD) is the practice of generating deliverable as soon as code changes happen. CD requires that CI pipeline be in place first. Hence, CI is a prerequisite of CD. Here, the deliverable is in the form of a docker image. The image consists of everything that our project requires to run including OS, dependencies, a web server (here Django’s uwsgi), sound drivers for audio input etc. The tools used here for creating a CD pipeline are Docker and Jenkins.

Building and Publishing Images

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. So, first we create a Dockerfile. The Dockerfile is placed in the root directory of the project. It is shown below in Figure [28].

```
FROM ubuntu:18.04

RUN apt-get update && apt-get install -y \
    python3.6 \
    python3-pip

RUN mkdir /SPE_Speech_Evaluator

WORKDIR /SPE_Speech_Evaluator

COPY . /SPE_Speech_Evaluator

RUN apt-get -y install locales
RUN touch /usr/share/locale/locale.alias
ENV LANG=en_US.UTF-8 \ LANGUAGE=en_US \ LC_ALL=en_US.UTF-8

RUN echo $LANG
RUN python3 -c 'import locale; print(locale.getpreferredencoding())'

RUN apt-get update \
        && apt-get install -y pulseaudio alsa-utils alsa-base ffmpeg

RUN apt-get update \
        && apt-get install libportaudio2 libportaudiocpp0 portaudio19-dev libsndfile1-dev -y \
        && pip3 install pyaudio

RUN apt-get install -y pulseaudio

CMD python3 -c "import pyaudio"

COPY requirements.txt /SPE_Speech_Evaluator

RUN pip3 install -r requirements.txt

COPY . /SPE_Speech_Evaluator

EXPOSE 8000

CMD python3 manage.py runserver 0.0.0.0:8000
```

Figure [28] Dockerfile placed in root of project

In Jenkins pipeline, we configure a stage to build docker image of our web-application and publish it to DockerHub. In the above Dockerfile, we use base OS as Ubuntu 18.04, and install python3 and pip first. Then we set the working directory and install and set language as utf-8 in environment. Then we install necessary sound drivers, alsa drivers and codecs which are required to record sound through microphone device. Pulse audio and pyaudio is required to record audio. Then we install all the required dependencies through requirements.txt file. We expose port 8000 of built docker container through which we can access our web-application. The last command is Django's server run command.

Jenkins builds and pushes the image to Hub through commands "`docker build . -t akshaymg99/speech-spe:latest`" and "`docker push akshaymg99/speech-spe:latest`" as shown in Jenkinsfile Figure [17].

We can also see the pushed docker image in our Docker hub account as shown in Figure [29].

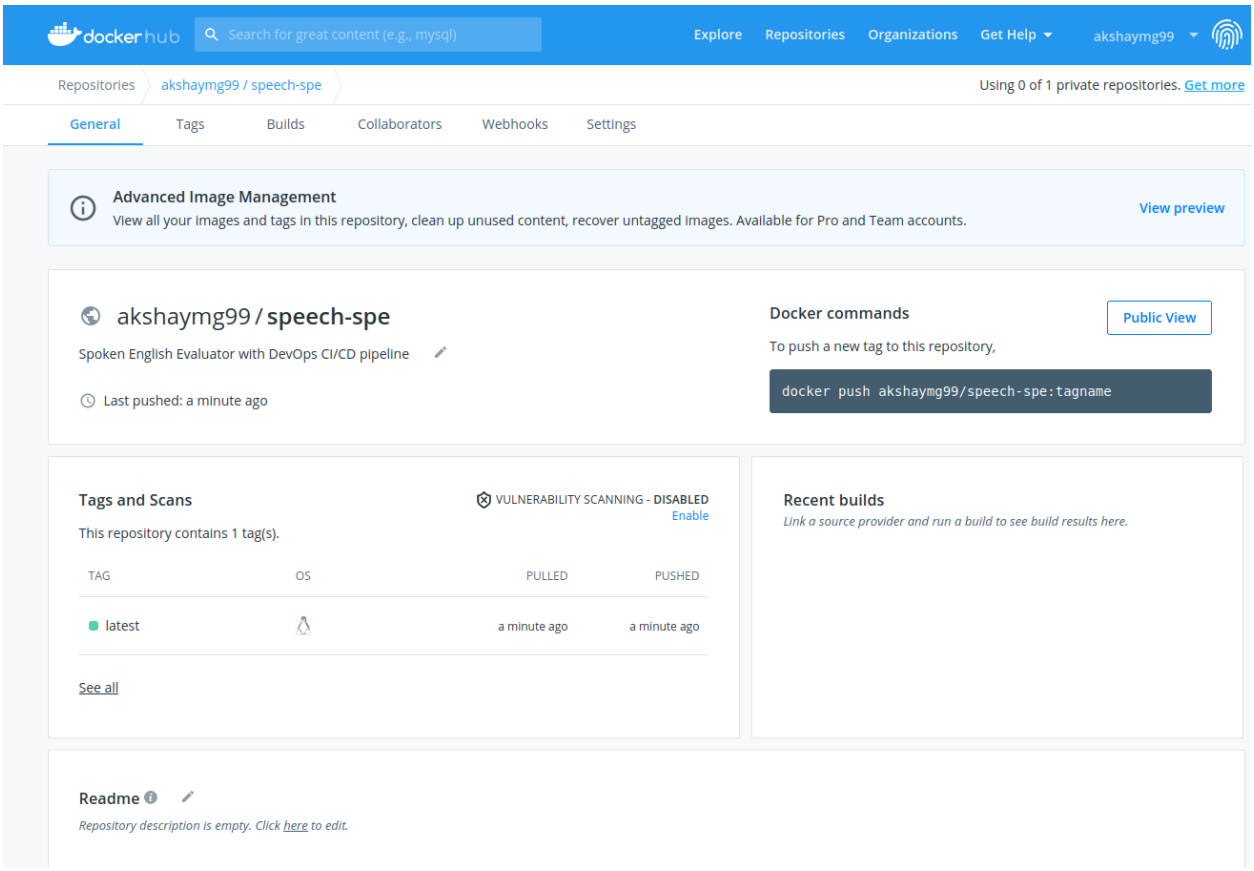


Figure [29] Docker Hub account with pushed images

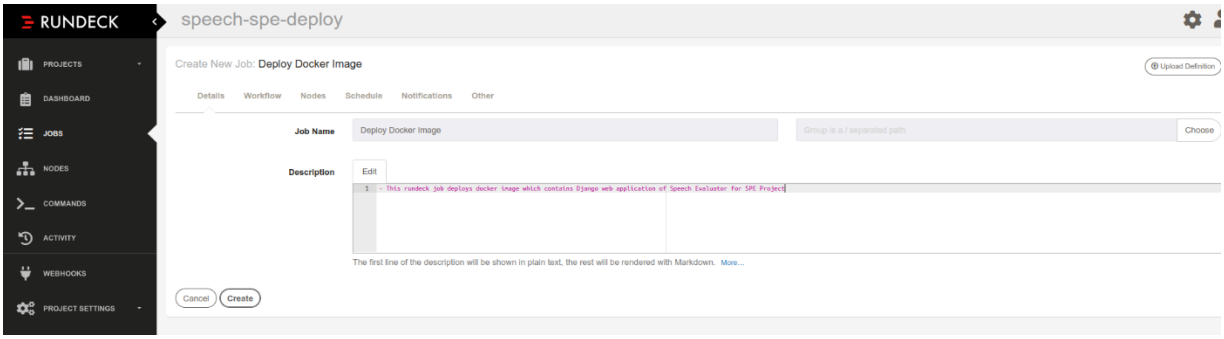
Continuous Deployment

Continuous deployment is a strategy for software releases wherein any code commit that passes the automated testing phase is automatically released into the production environment, making changes that are visible to the software's users.
After the deliverable (image in our case) is created and published to Dockerhub, we use Rundeck to fetch the image and deploy it in a container. The Rundeck job is invoked by Jenkins.

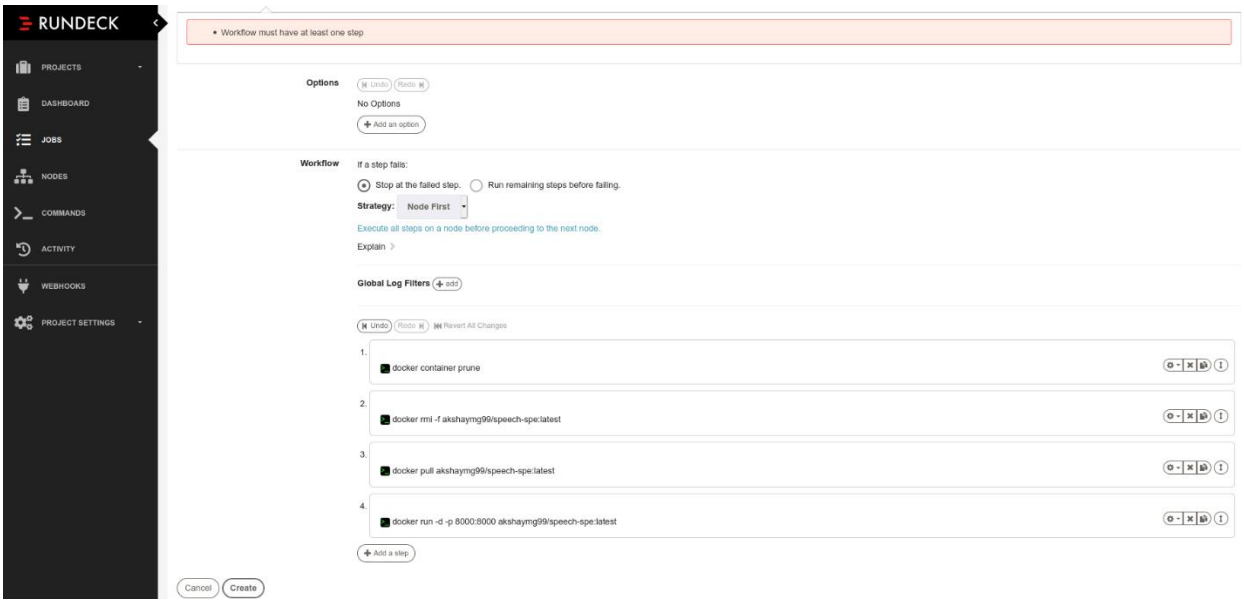
Creating Rundeck Project and Job

We create a Rundeck project and Rundeck job that deploys the generated Docker image locally. To do so, we follow these steps:

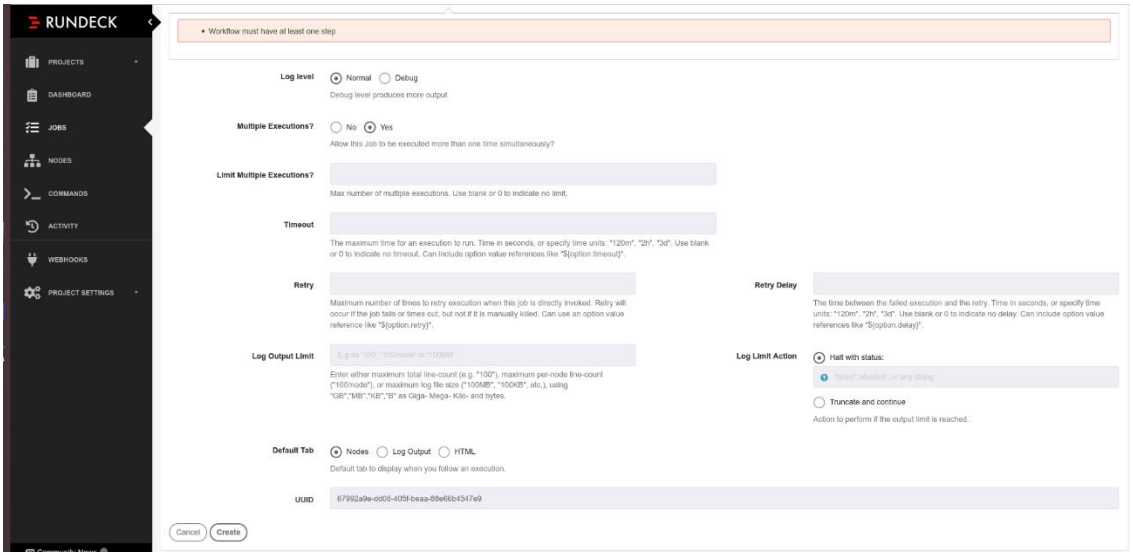
1. Create a new Rundeck job



2. Workflow description



3. Configure Node to execute it locally
4. Note down the UUID of the job to put it in Jenkins



We have shown how to use Jenkins to invoke Rundeck deploy job using UUID in Continuous Integration Part. We have also explained on how to use trigger to start the execution of this job. Figure [30] shows docker deployed locally running in Rundeck console.

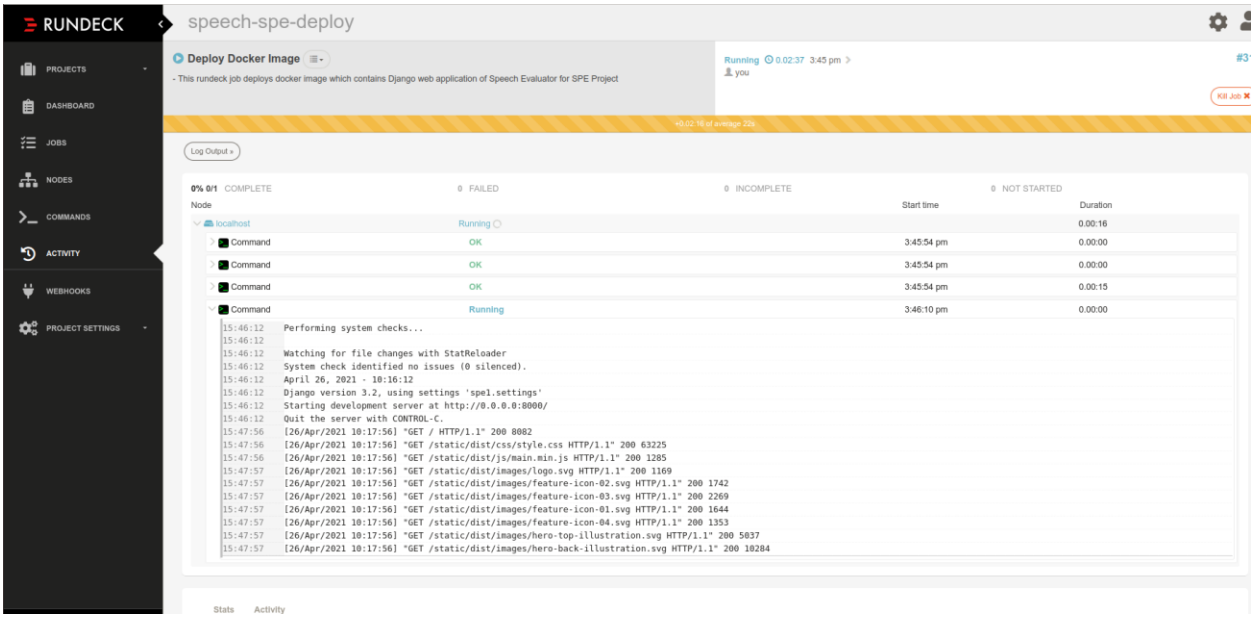


Figure [30] Rundeck deploying docker container locally

Continuous Monitoring

Monitoring refers to monitoring the deployed artifacts in real-time to keep a check on faults and measure performance. This is done by analyzing logs generated by the system. The Elastic Stack, also known as ELK Stack, comprises of Elasticsearch, Logstash, Kibana and Beats.

Elasticsearch is a modern search and analytics engine which is based on Apache Lucene. It is used as to store and index logs and can be then queried to extract meaningful insights. It can be used for numerous types of data including textual, numerical, geospatial, structured, and unstructured.

Logstash is a tool that is used for parsing logs. It is very useful in parsing unstructured logs and giving them structure so that logs can be efficiently searched and analyzed. Log aggregated and processed by Logstash go through 3 stages collection, processing and dispatching.

Kibana adds a visualization layer to the Elastic Stack. It is a browser-based user interface that can be used to search, analyze and visualize the data stored in Elasticsearch indices.

Beats are a collection of open source log shippers that act as agents installed on the

different servers in your environment for collecting logs or metrics. These shippers are designed to be lightweight in nature they leave a small installation footprint, are resource efficient, and function with no dependencies. Common Beats that are used today included Filebeat, Metricbeat, Winlogbeat, Packetbeat, etc.
Here we use the Elastic Stack to perform log analysis in this project. We are use Filebeat to collect logs generated by Django's web application, which are then given to Elasticsearch for analysis. Kibana is used to charts to visualize the data diagrammatically.

Configuring Logging in Django

Django provides loggers to generate logs and push them to either files or Logstash as required. Below Figure [31] shows Loggers configured in Django framework. We have configured it to generate logs and store them in a file. These settings are present in setting.py file

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "root": {"level": "INFO", "handlers": ["file"]},
    "handlers": {
        "file": {
            "level": "INFO",
            "class": "logging.FileHandler",
            "filename": os.path.join(BASE_DIR, 'debug.log'),
            "formatter": "app",
        },
    },
    "loggers": {
        "django": {
            "handlers": ["file"],
            "level": "INFO",
            "propagate": True
        },

        "django.request": {
            'handlers': ['file'],
            'level': 'INFO',
        },
        "": {
            'level': 'INFO',
            'handlers': ['file'],
        },

    },
    "formatters": {
        "app": {
            "format": (
                u"%(asctime)s [%(levelname)-8s] "
                "(%(module)s.%(funcName)s) %(message)s"
            ),
            "datefmt": "%Y-%m-%d %H:%M:%S",
        },
    },
}

```

Figure [31] Logging configuration in Django

Then we use Filebeat to fetch the logs present in the root directory of project (/debug.log file). Configuring of Filebeat is shown in below figures. It is present at /etc/filebeat/filebeat.yml

```
# For more available modules and options, please see the filebeat.reference.yml sample
# configuration file.

# ============================== Filebeat inputs ===============================

filebeat.inputs:

# Each - is an input. Most options can be set at the input level, so
# you can use different inputs for various configurations.
# Below are the input specific configurations.

- type: log

  # Change to true to enable this input configuration.
  enabled: true

  # Paths that should be crawled and fetched. Glob based paths.
  paths:
    - /var/log/*.log
    - /home/akshay/Desktop/IIITB/SPE_Speech_Evaluator/*.log
    #- c:\programdata\elasticsearch\logs\*

  # Exclude lines. A list of regular expressions to match. It drops the lines that are
  # matching any regular expression from the list.
  #exclude_lines: ['^DBG']

  # Include lines. A list of regular expressions to match. It exports the lines that are
  # matching any regular expression from the list.
  #include_lines: ['^ERR', '^WARN']

  # Exclude files. A list of regular expressions to match. Filebeat drops the files that
  # are matching any regular expression from the list. By default, no files are dropped.
  #exclude_files: ['.gz$']

  # Optional additional fields. These fields can be freely picked
  # to add additional information to the crawled log files for filtering
  #fields:
```

YAML ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

```
  # Optional additional fields. These fields can be freely picked
  # to add additional information to the crawled log files for filtering
  #fields:
  #  level: debug
  #  review: 1

# ============================== Filebeat modules ==============================

filebeat.config.modules:
  # Glob pattern for configuration loading
  path: ${path.config}/modules.d/*.yml

  # Set to true to enable config reloading
  reload.enabled: true

  # Period on which files under path should be checked for changes
  #reload.period: 10s

# ======================= Elasticsearch template setting =======================

setup.template.settings:
  index.number_of_shards: 1
  #index.codec: best_compression
  #_source.enabled: false


# ================================== General ===================================

# The name of the shipper that publishes the network data. It can be used to group
# all the transactions sent by a single shipper in the web interface.
#name:

# The tags of the shipper are included in their own field with each
# transaction published.
#tags: ["service-X", "web-tier"]

# Optional fields that you can specify to add additional information to the
```

YAML ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS

```
#fields:
#  env: staging

# ================================== Dashboards =====================================
# These settings control loading the sample dashboards to the Kibana index. Loading
# the dashboards is disabled by default and can be enabled either by setting the
# options here or by using the `setup` command.
setup.dashboards.enabled: true

# The URL from where to download the dashboards archive. By default this URL
# has a value which is computed based on the Beat name and version. For released
# versions, this URL points to the dashboard archive on the artifacts.elastic.co
# website.
#setup.dashboards.url:

# =================================== Kibana ========================================

# Starting with Beats version 6.0.0, the dashboards are loaded via the Kibana API.
# This requires a Kibana endpoint configuration.
setup.kibana:

  # Kibana Host
  # Scheme and port can be left out and will be set to the default (http and 5601)
  # In case you specify and additional path, the scheme is required: http://localhost:5601/path
  # IPv6 addresses should always be defined as: https://[2001:db8::1]:5601
  host: "localhost:5601"

  # Kibana Space ID
  # ID of the Kibana Space into which the dashboards should be loaded. By default,
  # the Default Space will be used.
  #space.id:

# ============================== Elastic Cloud ==============================

# These settings simplify using Filebeat with the Elastic Cloud (https://cloud.elastic.co/).

# The cloud.id setting overwrites the `output.elasticsearch.hosts` and
# `setup.kibana.host` options.
```

YAML Tab Width: 8 Ln 1, Col 1 INS

```
# Configure what output to use when sending the data collected by the beat.

# ---------------------------- Elasticsearch Output ----------------------------
output.elasticsearch:
  # Array of hosts to connect to.
  hosts: ["localhost:9200"]
  index: "logstash-%{+yyyy.MM.dd}"
setup.template:
  name: 'logstash'
  pattern: 'logstash-*'
  enabled: false
  # Protocol - either `http` (default) or `https`.
  #protocol: "https"

  # Authentication credentials - either API key or username/password.
  #api_key: "id:api_key"
  #username: "elastic"
  #password: "changeme"

# ------------------------------ Logstash Output -------------------------------
#output.logstash:
  # The Logstash hosts
  #hosts: ["localhost:5044"]

  # Optional SSL. By default is off.
  # List of root certificates for HTTPS server verifications
  #ssl.certificate_authorities: ["/etc/pki/root/ca.pem"]

  # Certificate for SSL client authentication
  #ssl.certificate: "/etc/pki/client/cert.pem"

  # Client Certificate Key
  #ssl.key: "/etc/pki/client/cert.key"

# ================================ Processors =====================================
processors:
  - add_host_metadata:
      when.not.contains.tags: forwarded
```

YAML Tab Width: 8 Ln 1, Col 1 INS

Filebeat fetches the logs and sends it to Elasticsearch. There, we can make an index pattern to filter the log data. Figure [32] shows creating index pattern in Elasticsearch.

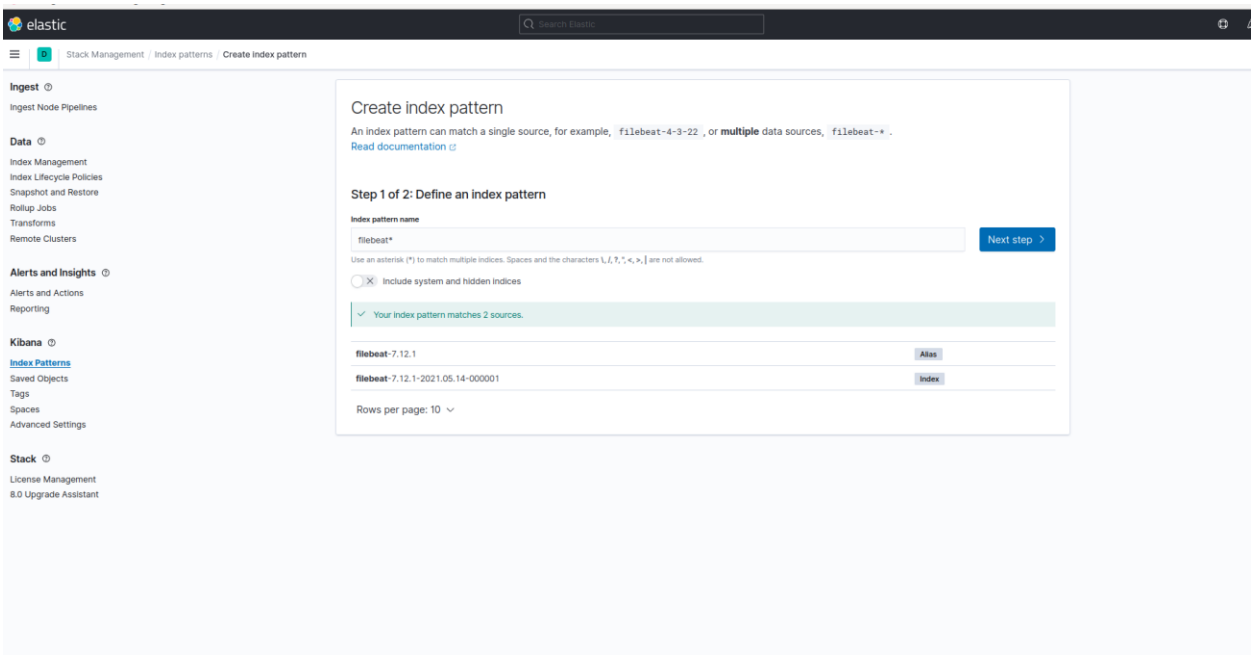


Figure [32] Creating index pattern

Next we select that saved index pattern for visualization as shown in Figure [33]

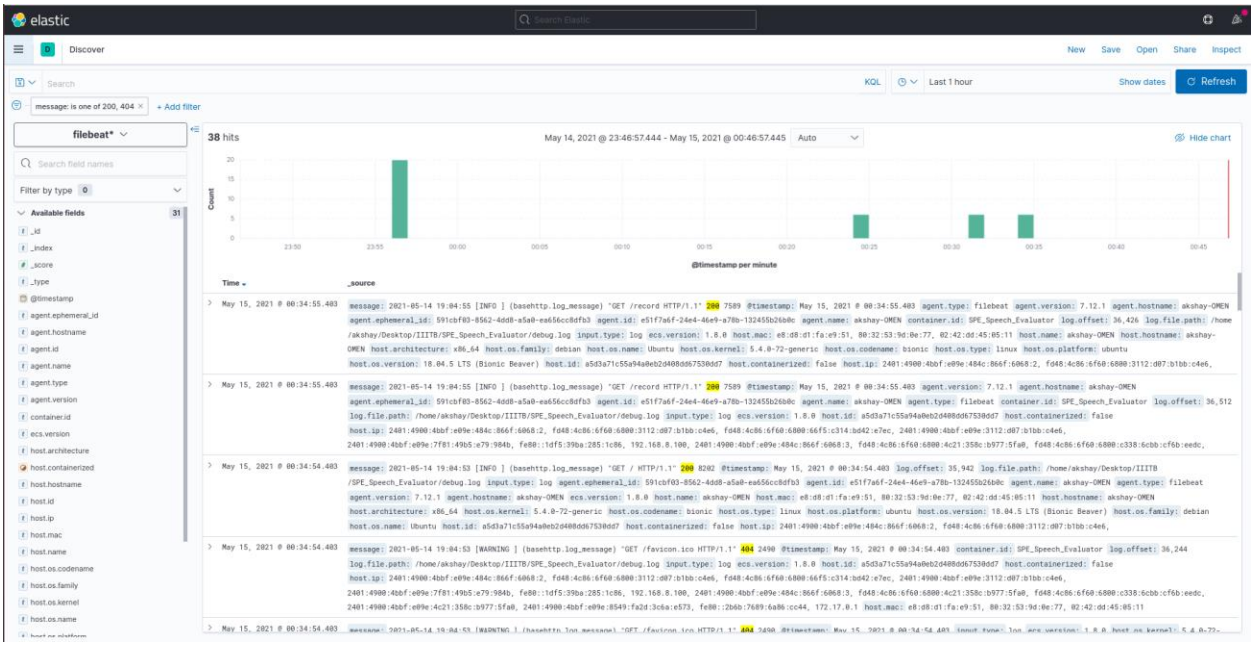
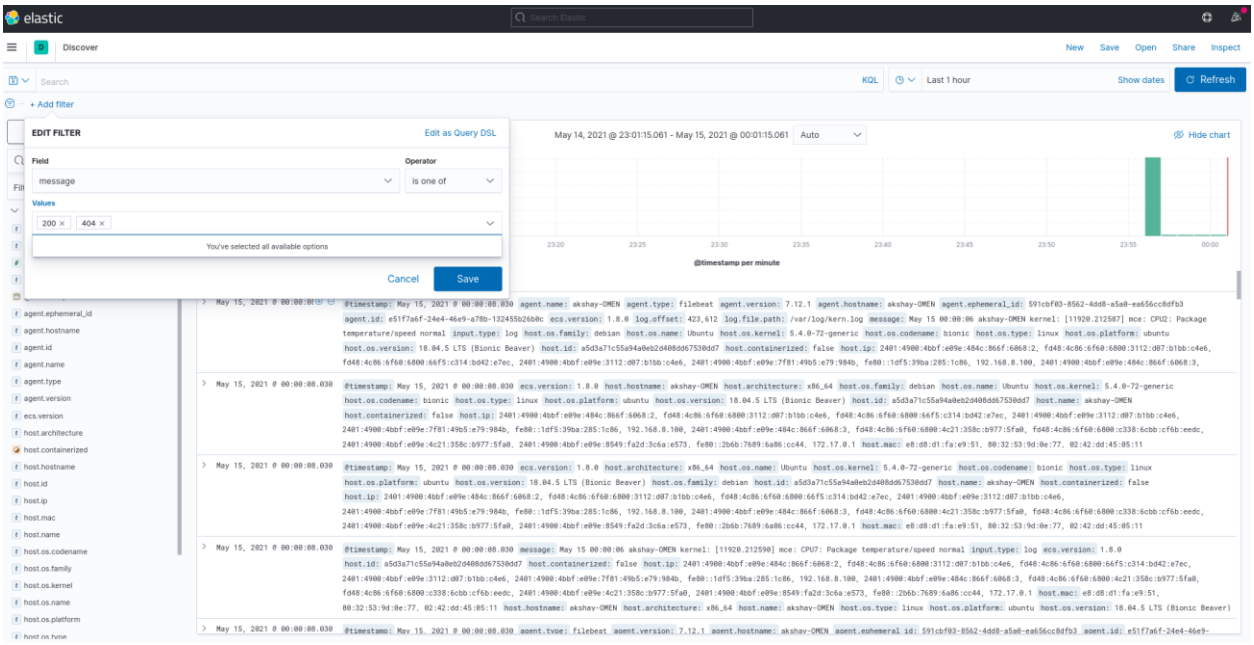
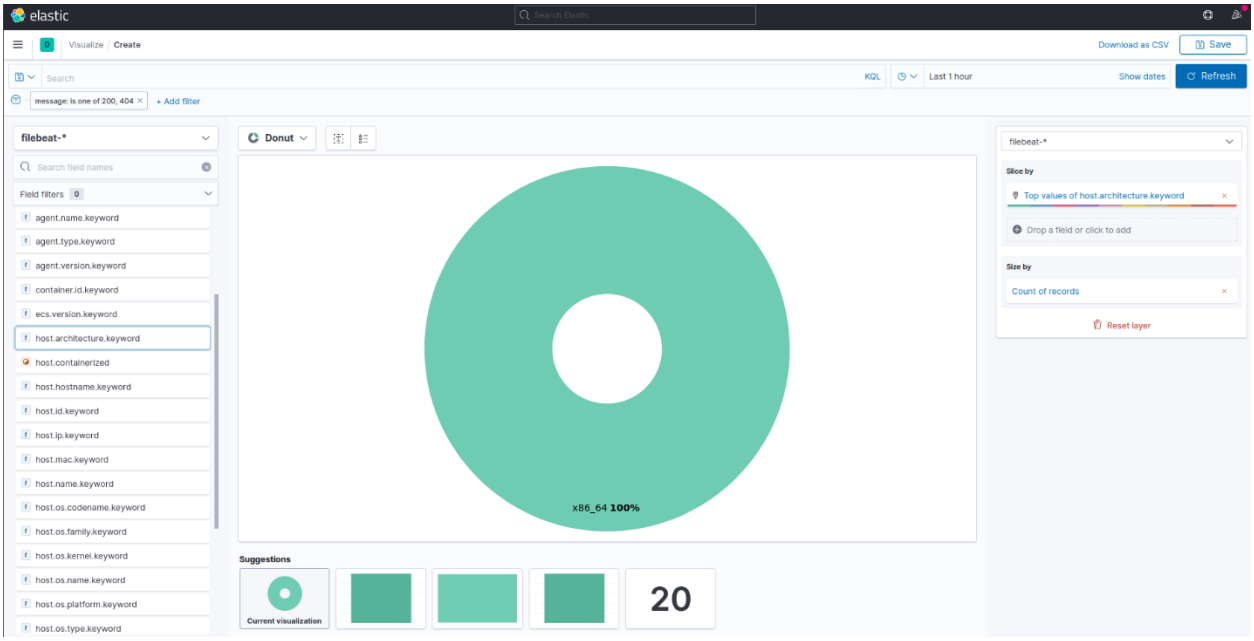


Figure [33] Selecting index pattern

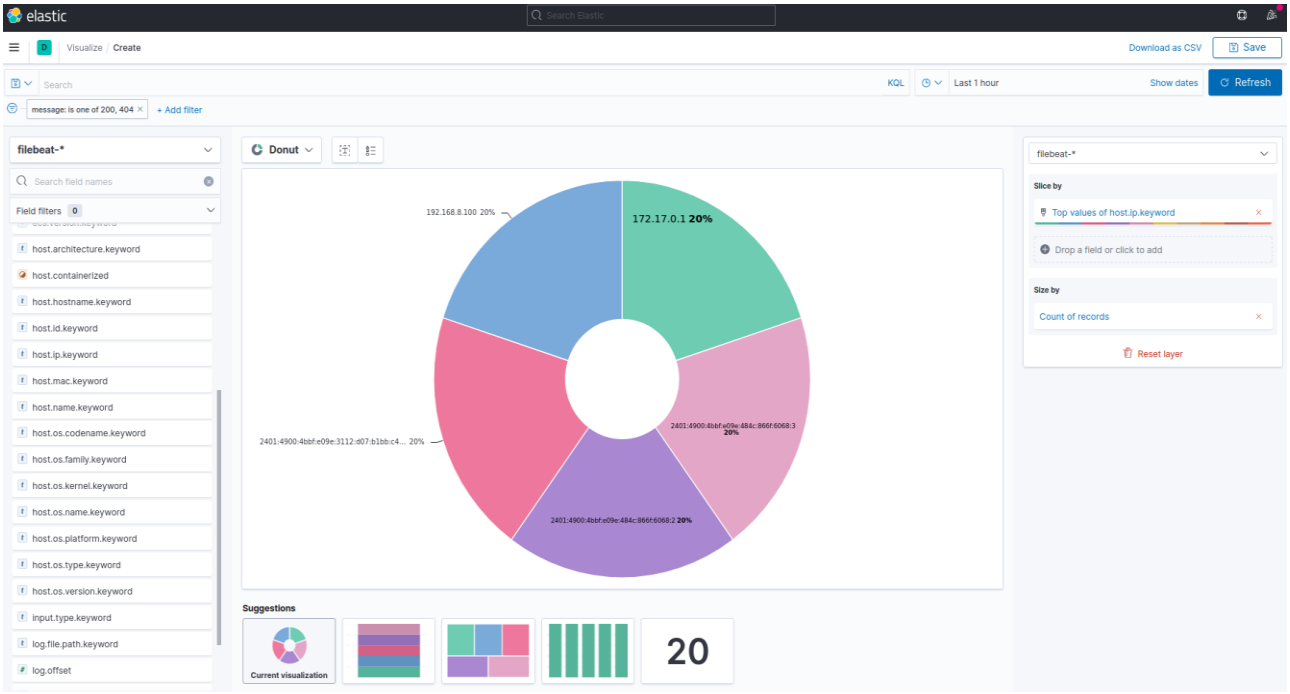
Then we add filters to select data which contains status code 200 and 404 which will help us identify when the webpages where successfully retrieved or not. We save this query here.



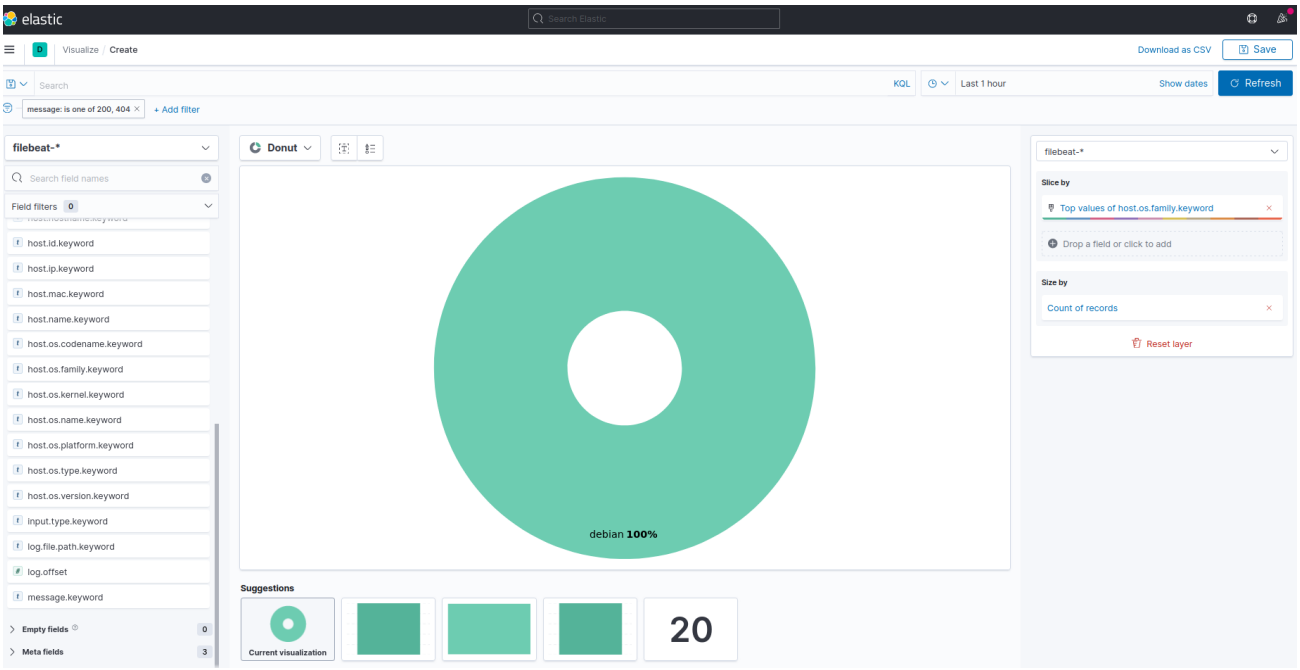
After that, we open Kibana visualization, and select the saved query to display visualizations. Below are some interesting analytics obtained visually from the logs.
This shows information about the host architecture in the form of a pie chart.



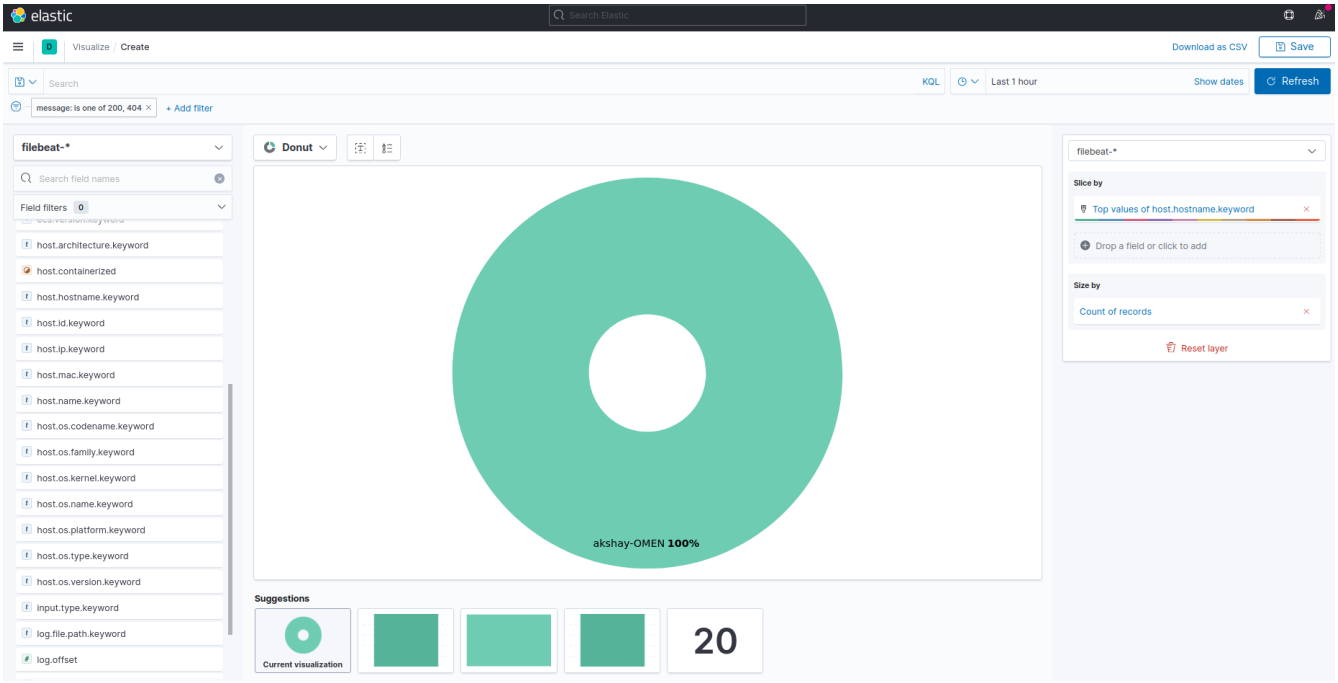
Next figure shows information about host IP address.



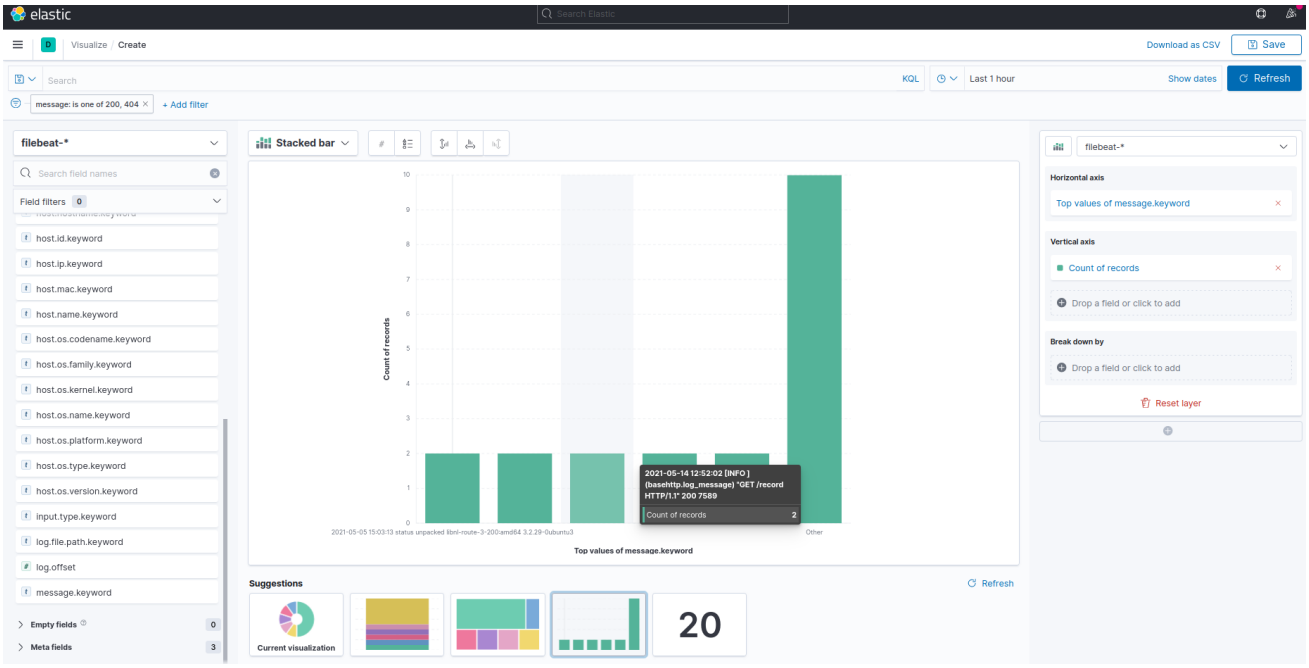
Below figure shows information on host OS type:



Next Figures shows information about hostname:



The following Figure shows information on message requests in the form of bar chart:



Results and Discussions:

The web-application calculates result in the form of sentence analysis, vocabulary strength and unique word count. Figure [33] shows these result displayed in result webpage.

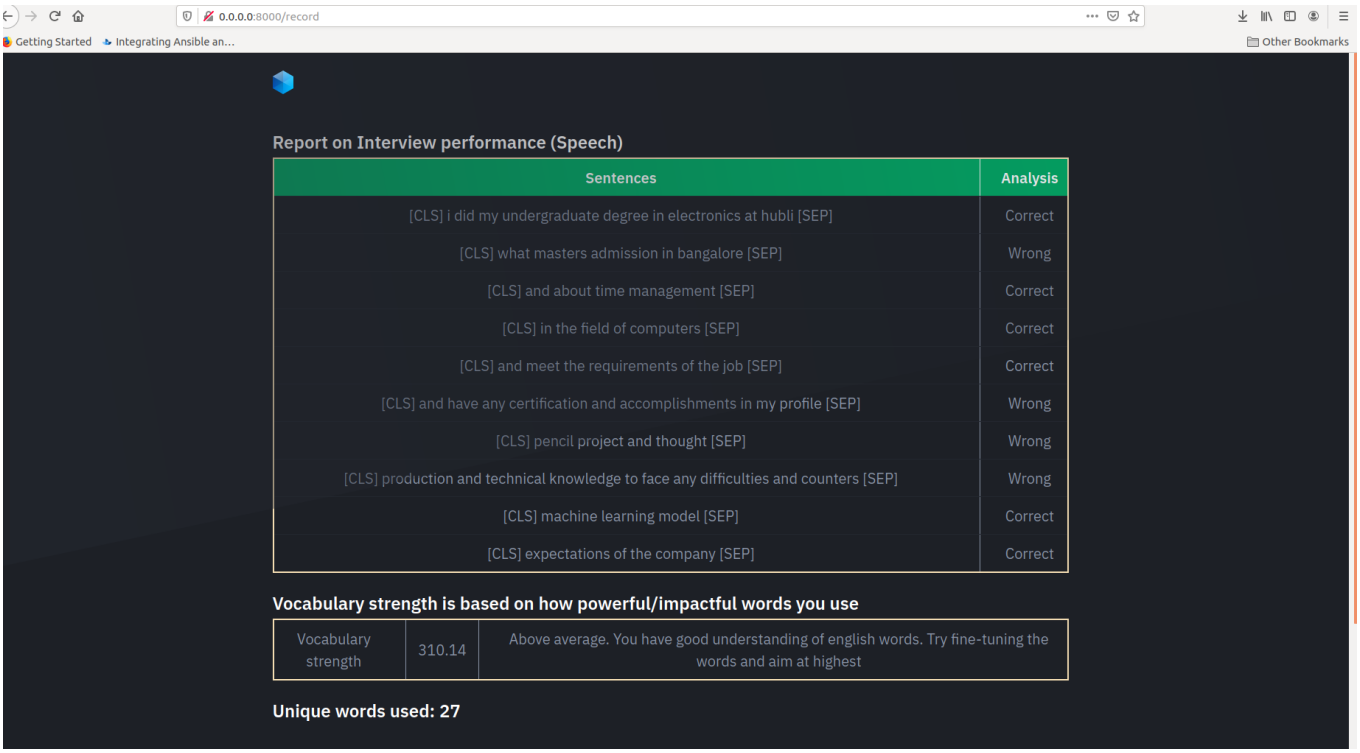


Figure [33] Result displayed on webpage

We find that speech to text conversion is not perfect and it also sometimes fails to properly separate the sentences as we speak. We used available open source google speech recognition library, so we couldn't make this feature perfect as more optimal libraries which were not free to use. This is the only drawback of this application.

Scope for Future Work

We built an audio based system to evaluate speech for interviews. It could be taken forward and additional video analysis could also be added in future. Features such as body language analysis, emotion / confidence level analysis, eye contact all could be checked in a person's response video. This would make interview performance analysis much better.

Conclusion

In this project, we built a responsive web-application and automated the entire SDLC using DevOps toolchain. This makes the development team and operations team work productively as the DevOps pipeline gives the comfort of making code changes easily and also reduces the chances of encountering errors in production. The toolchain allows companies to quickly build, test and deploy new versions of their products.