UNIVERSIDAD SANTO TOMÁS
PRIMER CLAUSTRO UNIVERSITARIO DE COLOMBIA

FACULTAD DE INGENIERÍA ELECTRÓNICA

Proyecto # 3 Corte

Implementación de chatbot Deepseek en Pepper

Presentado a: Ing. Diego Alejandro Barragan Vargas
Juan Diego Báez Guerrero, Cód.: 2336781;

***Resumen*—Abstract— This paper details the implementation of a chatbot using DeepSeek API, integrated into the Pepper humanoid robot. The system is designed to allow conversational interaction via voice input processed by Pepper, which forwards the user's message to a chatbot deployed on a Flask server inside a Docker container. The communication protocol ensures Pepper's response is dynamically generated based on external API interaction. The development process includes containerization with Docker and real-time voice capture. Documentation is available in Overleaf and GitHub for reproducibility.**

***Resumen*—Resumen— Este documento detalla la implementación de un chatbot usando la API DeepSeek, integrado en el robot humanoide Pepper. El sistema permite la interacción conversacional mediante entrada de voz procesada por Pepper, que envía el mensaje del usuario a un chatbot desplegado en un servidor Flask dentro de un contenedor Docker. El protocolo de comunicación asegura que la respuesta de Pepper se genere dinámicamente con base en la interacción con la API externa. El desarrollo incluye contenedorización con Docker y captura de voz en tiempo real. La documentación está disponible en Overleaf y GitHub para su reproducibilidad.**

I. Introducción

El uso de inteligencia artificial para mejorar la interacción humano-robot ha avanzado considerablemente en la última década. Los chatbots basados en procesamiento de lenguaje natural (NLP) y aprendizaje profundo permiten una comunicación más fluida y natural entre humanos y sistemas autónomos [2]. Este proyecto busca integrar un chatbot desarrollado con la API DeepSeek en el robot Pepper, permitiéndole capturar voz, procesar el mensaje, enviarlo al servidor y devolver una respuesta basada en IA.

Para la implementación, se diseñó una arquitectura en la que el **servidor** corre en un **contenedor Docker**, proporcionando una API accesible para Pepper. Por su parte, el **cliente** alojado dentro de Pepper captura la entrada de voz, la envía al servidor y reproduce la respuesta generada. La estructura básica de ambos componentes se muestra en las Figuras **??** y **??**, respectivamente.

Este informe documenta la instalación, configuración y funcionamiento del chatbot dentro de Pepper, además de explorar el impacto de la contenedorización en la escalabilidad y flexibilidad del sistema.

II. Marco Teórico

A. Chatbots y Procesamiento de Lenguaje Natural

Los chatbots utilizan algoritmos de procesamiento de lenguaje natural (NLP) para comprender y generar respuestas en conversaciones [1]. Modelos avanzados como BERT han demostrado ser eficaces en este campo [2].

B. DeepSeek API

La API DeepSeek proporciona capacidades de procesamiento conversacional utilizando redes neuronales avanzadas [3]. Permite la interacción fluida con usuarios, facilitando tareas como generación de texto, comprensión de contexto y asistencia automatizada.

C. Arquitectura Cliente-Servidor en Pepper

El robot Pepper está diseñado para procesar comandos mediante Python y su SDK, que permite la gestión de voz y movimiento [5]. El sistema se basa en una arquitectura cliente-servidor, donde el cliente dentro de Pepper captura la voz y la envía al servidor Flask para procesar la respuesta.

D. Contenedorización con Docker

Docker facilita la creación de entornos virtualizados mediante contenedores que encapsulan dependencias y configuraciones [4]. Su implementación en este proyecto permite la ejecución aislada del servidor y mejora la portabilidad del chatbot en múltiples sistemas.

E. Captura y Procesamiento de Voz

Pepper usa 'ALSpeechRecognition' para detectar palabras clave y 'ALAnimatedSpeech' para generar respuestas en voz [5]. La integración con DeepSeek se logra mediante comunicación HTTP entre el cliente y el servidor.

III. Procedimiento y Resultados

A. Ingreso al proyecto

Para comenzar el desarrollo, se creó una carpeta llamada `PepperChatBot` como entorno raíz del proyecto, tal como se muestra en la Figura 1.

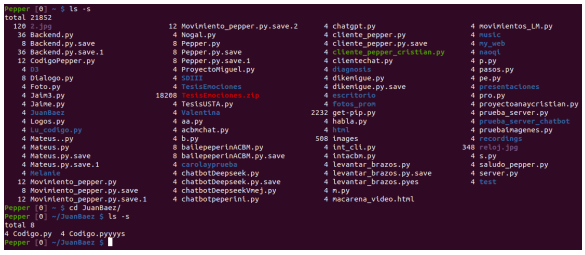

Figura 1: Ingreso a la carpeta raíz del proyecto PepperChatBot.

B. Instalación de requerimientos con pip

Finalmente, para completar el entorno Python en local, se instaló el archivo `requirements.txt` con pip, incluyendo Flask y requests. El proceso puede apreciarse en la Figura 2.

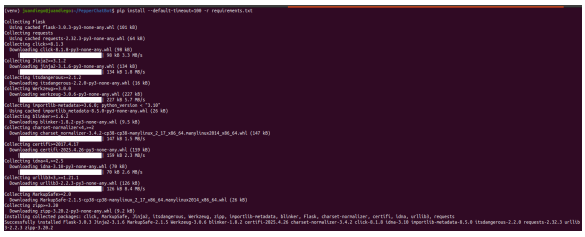

Figura 2: Instalación de dependencias mediante pip en entorno virtual.

C. Instalación del entorno y dependencias

Se comenzó creando un entorno virtual llamado PepperChatBot, como se muestra en la Figura 3. Este entorno facilita la instalación de librerías necesarias para el servidor Flask.

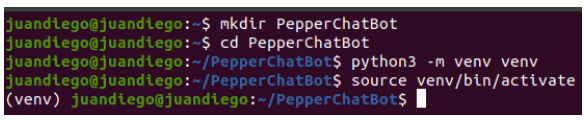

Figura 3: Creación del entorno virtual para el proyecto PepperChatBot.

Luego, se instaló Flask y requests desde el archivo requirements.txt. El proceso puede verse en la Figura 4.

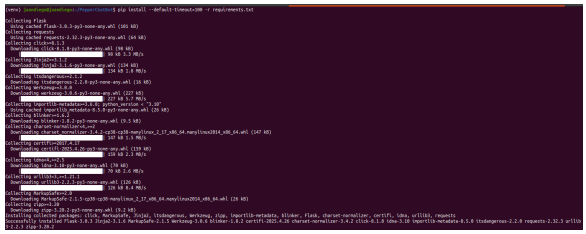

Figura 4: Instalación de dependencias del servidor.

D. Servidor Flask con DeepSeek API

El servidor fue configurado usando Flask, exponiendo el puerto 9559 para exposición. La Figura 5 muestra el código final empleado, donde se define la estructura del JSON enviado y la configuración del modelo `deepseek-chat`.

```
GNU nano 4.8
from flask import Flask, request, jsonify
import requests

app = Flask(__name__)

API_KEY = 'sk-53751d5c6f344a5dbc0571de9f51313e'
API_URL = 'https://api.deepseek.com/v1/chat/completions'

def enviar_mensaje(mensaje, modelo='deepseek-chat'):
    headers = {
        'Authorization': f'Bearer {API_KEY}',
        'Content-Type': 'application/json'
    }

    data = {
        'model': modelo,
        'messages': [
            {
                "role": "system",
                "content": "Eres un asistente calando con Jarvis pero tienes la personalidad de Bruce Wayne"
            },
            {
                "role": "user",
                "content": mensaje
            }
        ]
    }

    try:
        response = requests.post(API_URL, headers=headers, json=data)
        response.raise_for_status()
        return response.json()['choices'][0]['message']['content']
    except Exception as e:
        return f"Error: {str(e)}"

@app.route('/chat', methods=['POST'])
def chat():
    data = request.json
    mensaje = data.get('question', '')
    respuesta = enviar_mensaje(mensaje)
    return jsonify({"respuesta": respuesta})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=9559)
```

Figura 5: Servidor Flask con integración a la API de DeepSeek.

E. Cliente implementado en Pepper

El cliente fue escrito en Python usando el SDK de Pepper, aprovechando servicios como `ALAnimatedSpeech`. La Figura 6 muestra el código implementado.

```
# -*- coding: UTF-8 -*-

from __future__ import print_function, Unicode_literals

import time, json, requests

# --- Ajusta la IP/puerto de TU PC ----------------------------

SERVER_IP   = "192.168.0.114"   # ←^←^← tu PC con Flask

SERVER_PORT = 9559

URL         = "http://{}:{}/chat".format(SERVER_IP, SERVER_PORT)

# -------------------------------------------------------------


VOCABULARIO  = [u"hola", u"adiós", u"gracias", u"chat"]

HTTP_TIMEOUT = 8   # segundos


# ---------- utilidades -----------------------------------

def uprint(texto):

    """Imprime siempre en UTF-8 (evita UnicodeEncodeError en Python 2)."""

    if hasattr(sys.stdout, "buffer"):

        sys.stdout.buffer.write((Unicode(texto) + u"\n").encode("UTF-8"))

    else:

        print(texto.encode("UTF-8"))


# ---------- lógica principal -----------------------------

def main(sesión):

    tts = sesión.service("ALTextToSpeech")

    asr = sesión.service("ALSpeechRecognition")


    asr.setLanguage("Spanish")
```

Archivos

Figura 6: Script del cliente implementado en Pepper para la comunicación con el servidor.

F. Conexión con Pepper y despliegue del cliente

Se accedió al robot Pepper vía SSH, como se evidencia en la Figura 7. Desde allí, se creó una carpeta para almacenar el script del cliente y se verificó la ejecución correcta del código mediante `python cliente.py`.

```
juandiego@juandiego:~$ ssh nao@192.168.0.106
Password:
Pepper [0] ~ $
```

Figura 7: Conexión SSH con el robot Pepper desde el terminal Ubuntu.

G. Despliegue del servidor con Docker

Para garantizar la portabilidad del servidor, se construyó un contenedor Docker personalizado con el archivo `Dockerfile` que especifica dependencias y comandos de ejecución. La Figura 8 muestra su contenido.

```
 GNU nano 2.3.2                                File: Dockerfile

FROM python:3.10-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY server.py .

EXPOSE 9559

CMD ["python", "server.py"]
```

Figura 8: Dockerfile utilizado para contenerizar el servidor con Flask y DeepSeek.

H. Creación y edición de scripts

Tanto el servidor como el cliente fueron editados dentro de sus respectivos entornos. En las Figuras 9 y 10 se observa el uso de nano en terminal para editar server.py y cliente_pepper.py respectivamente.

```
(venv) juandiego@juandiego:~/PepperChatBot$ nano server.py
```

Figura 9: Edición del archivo `server.py` con `nano`.

```
Pepper [0] ~/JuanBaez $ mkdir Proyecto_tercer_corte
Pepper [0] ~/JuanBaez $ cd Proyecto_tercer_corte/
Pepper [0] ~/JuanBaez/Proyecto_tercer_corte $ nano cliente_pepper.py
```

Figura 10: Edición del archivo `cliente_pepper.py` en Pepper.

I. Ejecución e interacción

La ejecución de ambos scripts demostró una interacción exitosa entre Pepper y el servidor. En la Figura 11 se muestra el mensaje de espera y escucha del robot Pepper.

```
Pepper [0] ~/JuanBaez/Proyecto_tercer_corte $ nano cliente_pepper.py
Pepper [0] ~/JuanBaez/Proyecto_tercer_corte $ python cliente_pepper.py
[W] 1748300754.753794 3788 qi.path.sdklayout: No Application was created, trying to deduce paths
('Pepper est\xc3\xa1 listo para escuchar palabras:', ['hola', 'adi\xc3\xb3s', 'gracias', 'chat'])
```

Figura 11: Ejecución del cliente Pepper: escucha e interacción con el servidor.

J. Verificación de errores

```
Pepper [0] ~/JuanBaez/Proyecto_tercer_corte $ python cliente_pepper.py
Traceback (most recent call last):
  File "cliente_pepper.py", line 87, in <module>
    main(app.session)
  File "cliente_pepper.py", line 34, in main
    asr.setVocabulary(VOCABULARIO, False)   # False: solo esas palabras
RuntimeError:   NuanceContext::addContext
        A grammar named "modifiable_grammar" already exists.
Pepper [err 1] ~/JuanBaez/Proyecto_tercer_corte $
```

Figura 12: Error relacionado con el contexto de vocabulario en Pepper.

K. Verificación de despliegue con Docker

Una vez creado el contenedor, se procedió a construir la imagen desde el `Dockerfile`, lo cual se muestra en la

Figura 13. Esta operación garantizó que todas las dependencias y archivos del servidor quedaran contenidas en una única imagen.

Figura 13: Construcción de imagen Docker para el servidor Flask con DeepSeek.

Luego, se ejecutó el contenedor con el nombre `chatbot-deepseek`, exponiendo el puerto 9559 para que Pepper pudiera comunicarse con el servidor. La Figura 14 presenta la ejecución del contenedor.

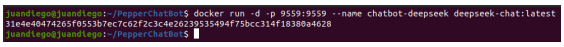

Figura 14: Ejecución del contenedor Docker.

Al detener el contenedor, se verificó su ausencia con el comando `docker ps`. Este paso se muestra en la Figura 15 y evidencia el control adecuado del ciclo de vida del contenedor.

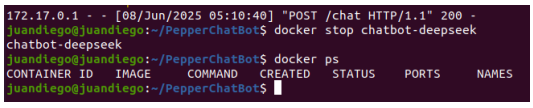

Figura 15: Finalización del contenedor con `docker stop`.

Adicionalmente, en la Figura 16 se presenta la visualización de logs, confirmando que el servidor Flask estaba escuchando correctamente en todas las interfaces disponibles.

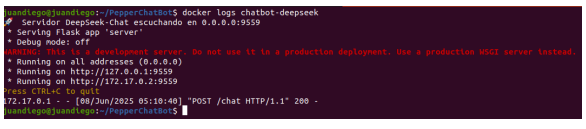

Figura 16: Logs de funcionamiento del contenedor Docker.

Finalmente, la Figura 17 presenta la prueba realizada mediante `curl`, que evidencia la recepción de un mensaje por el servidor y su correcta respuesta generada por DeepSeek.

Figura 17: Prueba de funcionamiento del servidor dentro del contenedor Docker.

Para verificar que el contenedor seguía corriendo, se volvió a consultar con `docker ps`, como se muestra en la Figura 18.

Figura 18: Verificación del contenedor en ejecución.

IV. CONCLUSIONES

Resume los hallazgos principales del estudio, responde a los objetivos planteados y sugiere posibles mejoras o investigaciones futuras.

REFERENCIAS do

[1] B. Liu, *Sentiment Analysis and Opinion Mining*. Springer, 2020.
[2] J. Devlin, *BERT: Pre-training of Deep Bidirectional Transformers*. ACM, 2019.
[3] M. Altman, "DeepSeek API for Conversational AI," *IEEE Transactions on AI*, 2022.
[4] D. Bernstein, "Containers and the Future of Cloud Computing," *ACM Cloud Computing*, 2014.
[5] SoftBank Robotics, "Naoqi SDK Documentation," [En línea]. Disponible en: https://developer.softbankrobotics.com. [Accedido: 08-jun-2025].