A Weekly Report

on

INVOICE ASSISTANT

at

OneOnic Solution

by

Mr. Tank Viraj Rupeshbhai

EC034, 21ECUBG069

B.Tech. (EC) Semester - VIII

Faculty Supervisor	External Guide
Dr. NarendraKumar Chauhan	Mr. Umesh Ghediya

In Partial Fulfillment of Requirement of

Bachelor of Technology - Electronics & Communication

Submitted To

Department of Electronics & Communication Engineering

Faculty of Technology

Dharmsinh Desai University, Nadiad - 387001.

(Feb 2025)

Final Task :- Invoice Assistant

Introduction :-

Up to now, we have done many small parts of projects. We learned how to read text from invoices, save that text in Google Sheets, and even answer questions about invoices using smart tools. Now, we have combined all these learnings to create one complete project – our Invoice Assistant.

Imagine it as a friendly robot that can take a picture or PDF of a bill, read it carefully, and then answer your questions in a simple way. This project brings together everything we have learned so far, making it a powerful and easy-to-use tool.

Workflow :-

- Install Required Libraries :
  Install important tools like google.generativeai, pdf2image, and telegram.ext to process invoices and handle messages.
- Set Up Environment Variables :
  Load API keys and configure Google Gemini so that our Invoice Assistant can communicate with the it.
- Log Events for Debugging and Monitoring :
  Logging is used to keep track of important events, such as file uploads, processing errors, and responses from Gemini. This helps in debugging issues and improving the system's reliability.
- Ask for Input in Telegram Bot :
  The bot starts by greeting the user and asking them to upload an invoice file in either image or PDF format.
- Upload the File from the User :
  The user sends an invoice file through Telegram, and the bot processes the upload request.

- Check File Type and Convert if Needed :
  The bot checks if the uploaded file is an image or a PDF. If it is a PDF, it is converted into an image so it can be processed correctly.

- Give File to Gemini and Extract the Text :
  The converted image is sent to Google Gemini, which reads and extracts all important details from the invoice.

- Organize the Extracted Data :
  The extracted information is structured in a well-organized format to make it easy to read and understand.

- Print the Extracted Text to the User and Generate JSON File :
  The bot sends the extracted invoice details as a message to the user. A JSON file containing the structured data is also generated and shared.

- Ask If the User Has Any Questions About the Invoice and Answer Them :
  The bot allows the user to ask questions related to the invoice, such as total amount, invoice number, or supplier details. The answers are provided based on the extracted data.

- End the Session to Start a New Session :
  The user can end the session by using a command, clearing the current data, and making the bot ready for the next invoice.

Code :-

Block 1

```
import os
import asyncio
import time
from dotenv import load_dotenv
from telegram import Update
from telegram.ext import Application, CommandHandler, MessageHandler,
filters, ContextTypes
from pdf2image import convert_from_path
import google.generativeai as genai
import json
import logging
```

```
from tenacity import retry, stop_after_attempt, wait_exponential,
retry_if_exception_type
from telegram.error import NetworkError, TimedOut
```

Explanation

This block imports all the necessary libraries to set up the Telegram bot, process invoices, and handle errors. It includes modules for handling operating system tasks, asynchronous execution, and loading environment variables securely. The Telegram bot framework allows interaction with users, while pdf2image helps convert PDFs into images for text extraction. Google Gemini is integrated for extracting structured data from invoices. Logging ensures proper tracking of activities, and tenacity provides a retry mechanism to handle network failures, ensuring smooth bot operation. These imports lay the foundation for the entire project.

Block 2

```
# Load environment variables from .env file
load_dotenv()
genai.configure(api_key=os.getenv("GOOGLE_API_KEY"))


# Global configurations
USER_SESSIONS = {}
QUESTION_PROMPT_TEMPLATE = """
        You are an expert in understanding invoices.
        You will receive input images as invoices &
        You are given with the data and user query.
        You just need to answer based on the information provided.
        Answer in a conversational manner, as if talking to a human.
        Any questions outside the information in the invoice will be
ignored and not answered.
        Thank you!
            Answer this question based strictly on the invoice content:
                Question: {question}

                Follow these guidelines:
                1. Be specific and use exact values from the invoice when
possible
                2. If information is missing/unclear, state "Not clearly
specified in the invoice"
                3. Format currency values with their symbols (e.g., ₹, $,
€)
                4. Never guess or assume values not shown in the invoice
"""
EXTRACTION_PROMPT = """
               You are an expert in understanding and extracting detailed
information from invoices of any kind.
```

You will receive various invoice images as input and need to provide accurate and comprehensive answers based on the extracted details.

The extracted information should cover a wide range of invoices, including food invoices, e-commerce invoices, utility bills, and any other type of invoice.

Dont use "\n" in extracted json format anywhere.

Below is an example structure, but please adapt and modify the structure as needed for different invoice types:

```
{
    "Supplier": {
        "Name": "Supplier Name",
        "Address": "123 Supplier St, City, ZIP",
        "PAN Number": "PAN123456789",
        "GST Number": "GST0000",
        "Contact": "xxxxxxxxxxx"
    },
    "ReceiptDetails": {
        "Billing Address": "Customer Billing Address",
        "Shipping Address": "Customer Shipping Address",
        "Invoice Number": "INV123456",
        "Date": "YYYY-MM-DD",
        "Time": "HH:MM:SS",
    },
    "Items": [
        {
            "ItemName": "Product Name or Service",
            "Quantity": 2,
            "Unit Price": "50.00",
            "Total Price": "100.00"
        },
        {
            "ItemName": "Another Product",
            "Quantity": 3,
            "Unit Price": "30.00",
            "Total Price": "90.00"
        }
    ],

    "Payment Details" :{
        "Payment Method": "Credit Card / Cash / Bank Transfer / Other",
        "Currency": "USD",
        "Total Amount": "250.00",
        "Taxes": "25.00",
        "Discounts": "5.00"
    }
    "Tax calculation" :{
     ......
    }
}
```

Instructions for Processing Invoices:

```
                1. Tax Calculations:
                    Sum all taxes (e.g., CGST, SGST, and any others) and
ensure the total tax amount is accurate.
                    If tax percentages are listed, verify they sum up
correctly to the total tax amount.
                    Ensure that the taxable amount plus total taxes equals
the final amount after applying any discounts.

                2. Item Extraction:
                    Split item details correctly even if item names or
descriptions span multiple rows.
                    Treat every line within the same column as part of a
single item.
                    If quantity is more than one then unit price and total
price can not be same
                    If unit price is not given of an item then divide
amount by quantity to get Unit price

                3. Invoice Generation Time:
                    Look for common formats such as "am/PM" to identify
when the invoice was generated or paid.

                4. Data Verification:
                    Cross-verify extracted data to ensure it matches the
expected invoice totals:
                      Verify if the total amount equals the sum of item
prices plus taxes, minus any discounts.
                      Check if the total quantity of items matches the sum
of quantities for each item if provided.

                5. Subtotal Calculation:
                    Ignore extracting 'Subtotal' directly from the image.
                    The actual 'Subtotal' should be calculated as:
                     Subtotal = TotalAmount - TaxAmount + DiscountAmount`
                    Ensure the validation that:
                     FinalTotal = Subtotal + TaxAmount - DiscountAmount`

                6. Currency Extraction:
                   Don't just use the currency sumbol in json as it is.
Find out the name of that currency adn then add that name in currency.
                """


# Configure logging
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    level=logging.INFO)
logger = logging.getLogger(__name__)


# Retry decorator for network operations
network_retry = retry(
    stop=stop_after_attempt(3),
```

```
    wait=wait_exponential(multiplier=1, min=2, max=10),
    retry=retry_if_exception_type((NetworkError, TimeoutError,
ConnectionError)),
    before_sleep=lambda _: logger.warning("Retrying due to network
error..."),
    reraise=True
)
```

Explanation

The load_dotenv() function loads environment variables into the program, while genai.configure(api_key=os.getenv("GOOGLE_API_KEY")) retrieves the Google Gemini API key. This keeps sensitive information out of the code, making the project safer and easier to manage.

The USER_SESSIONS dictionary helps keep track of active users and their uploaded invoices. The QUESTION_PROMPT_TEMPLATE ensures that Gemini only answers based on the invoice content, providing accurate responses. The EXTRACTION_PROMPT defines how invoice details should be extracted, including supplier information, payment details, and tax calculations. It also includes specific rules to improve accuracy, such as calculating subtotals correctly and verifying extracted amounts.

Logging is used to monitor system events, track errors, and help with debugging. This makes it easier to find and fix issues if something goes wrong. The network_retry decorator helps handle network-related failures by retrying failed operations up to three times. It uses an exponential backoff strategy to avoid overwhelming the system and logs warnings before each retry, making the bot more reliable.

Block 3

```
# Handles the /start command.
async def start(update: Update, context: ContextTypes.DEFAULT_TYPE):
    await update.message.reply_text(
        "Hi! I'm your Invoice Assistant. Please send me an invoice file
(image or PDF) to get started."
)
```

Explanation

The start function responds to the /start command by greeting the user and instructing them to upload an invoice file. It runs asynchronously to ensure smooth interaction without delays.

Block 4

```
# Handles file uploads with connection error handling
async def handle_file(update: Update, context: ContextTypes.DEFAULT_TYPE):
    user = update.message.from_user
    try:
        document = update.message.document

        # Check if the file format is supported
        if not document.file_name.lower().endswith(('.pdf', '.jpg',
'.jpeg', '.png')):
            await update.message.reply_text("⚠️ Unsupported file format!
Please upload a PDF or image.")
            return

        # Create user directory
        user_dir = f"user_{user.id}"
        os.makedirs(user_dir, exist_ok=True)

        # Retryable download operation
        @network_retry
        async def download_with_retry():
            file = await document.get_file()
            file_path = os.path.join(user_dir, document.file_name)
            await file.download_to_drive(file_path)
            return file_path

        await update.message.reply_text("🔄 Processing your document...")
        file_path = await download_with_retry()

        # Convert PDF files to images
        if document.file_name.lower().endswith('.pdf'):
            file_path = await convert_pdf_to_image(file_path, user_dir)
            if not file_path:
                await update.message.reply_text("❌ Failed to process
PDF.")
                return

        # Store file path and process
        USER_SESSIONS[user.id] = {"image_path": file_path}
        await process_invoice_image(update, file_path, user_id=user.id)

    except Exception as e:
        logger.error(f"File handling error: {str(e)}")
        await update.message.reply_text("⚠️ Connection issue detected.
Please try again later.")
```

Explanation

The handle_file function manages file uploads from users while handling connection errors. It first checks if the uploaded file is a supported format (PDF or image). If valid, it creates a user-specific directory and downloads the file using a retry mechanism to handle network failures.

If the file is a PDF, it is converted into an image for further processing. The file path is then stored in USER_SESSIONS, and the invoice is processed. If any error occurs during this process, an error message is logged, and the user is notified.

Block 5

```
# Converts PDF to JPEG image using threads.
async def convert_pdf_to_image(pdf_path: str, output_dir: str):
    try:
        images = await asyncio.to_thread(convert_from_path, pdf_path)
        image_path = f"{output_dir}/invoice.jpg"
        await asyncio.to_thread(images[0].save, image_path, "JPEG")
        return image_path
    except Exception as e:
        print(f"PDF conversion error: {e}")
        return None
```

Explanation

The convert_pdf_to_image function converts a PDF into a JPEG using threading to keep the process efficient. It saves the first page as invoice.jpg in the output directory. If an error occurs, it logs the issue and returns None.

Block 6

```
# Processes an invoice image, extracts data using Gemini, and sends the
result to the user.
async def process_invoice_image(update: Update, image_path: str, user_id:
int):
    try:
        # Retryable Gemini API call
        @network_retry
        async def get_gemini_response():
            return await asyncio.to_thread(
                genai.GenerativeModel('gemini-1.5-
flash').generate_content,
                [EXTRACTION_PROMPT, {"mime_type": "image/jpeg", "data":
open(image_path, "rb").read()}]
            )

        response = await get_gemini_response()
        extracted_text = response.text

        # Clean and validate JSON response
        cleaned_json = extracted_text.strip('```').replace('json\n', '',
1).strip()

        try:
```

```
            json_data = json.loads(cleaned_json)
            user_dir = f"user_{user_id}"
            base_name = os.path.splitext(os.path.basename(image_path))[0]
            json_filename = f"{base_name}_data.json"
            json_path = os.path.join(user_dir, json_filename)

            with open(json_path, 'w') as json_file:
                json.dump(json_data, json_file, indent=4)

            # Retryable send operations
            @network_retry
            async def send_responses():
                await update.message.reply_text("☑ Extraction complete!
Here's the JSON data:")
                await update.message.reply_text(f"{extracted_text}")
                await update.message.reply_text("☑ Here's the JSON
File:")
                await
update.message.reply_document(document=open(json_path, 'rb'),
filename=json_filename)
                await update.message.reply_text(" 💡 You can now ask
questions using /ask")
            await send_responses()

        except json.JSONDecodeError:
            await update.message.reply_text("⚠ The response wasn't valid
JSON. Here's the raw text:")
            await update.message.reply_text(extracted_text)

    except Exception as e:
        logger.error(f"Processing error: {str(e)}")
        await update.message.reply_text("⚠ Service unavailable. Please
try again later.")
```

Explanation

The process_invoice_image function sends the invoice image to Google Gemini for text extraction and processes the extracted data. It uses a retry mechanism to handle network failures when calling Gemini. The extracted text is cleaned and converted into JSON format. If valid, the data is saved as a JSON file in the user's directory and sent to the user. If the response is not valid JSON, the raw text is shared instead. Any errors during processing are logged, and the user is informed if the service is unavailable.

Block 7

```
# Handles user questions about the uploaded invoice
async def ask_question(update: Update, context:
ContextTypes.DEFAULT_TYPE):
```

```
    user = update.message.from_user
    try:
        if not USER_SESSIONS.get(user.id, {}).get("image_path"):
            await update.message.reply_text("⚠️ Please upload an invoice
first!")
            return

        question = " ".join(context.args)
        full_prompt = QUESTION_PROMPT_TEMPLATE.format(question=question)

        # Retryable Gemini API call
        @network_retry
        async def get_answer():
            return await asyncio.to_thread(
                genai.GenerativeModel('gemini-1.5-
flash').generate_content,
                [full_prompt, {"mime_type": "image/jpeg",
                               "data":
open(USER_SESSIONS[user.id]["image_path"], "rb").read()}]
            )

        response = await get_answer()
        await update.message.reply_text(f"📝
Response:\n\n{response.text}")
        await update.message.reply_text("💡 Ask another question or /end
to finish")

    except Exception as e:
        logger.error(f"Question error: {str(e)}")
        await update.message.reply_text("⚠️ Service unavailable. Please
try your question again.")
```

Explanation

The ask_question function allows users to ask questions about their uploaded invoice. It first checks if the user has uploaded an invoice; if not, it prompts them to do so. The user's question is formatted into a structured prompt for Google Gemini, which extracts relevant details from the invoice. The function uses a retry mechanism to handle network failures. The bot then sends Gemini's response back to the user and invites them to ask more questions or end the session. If an error occurs, it is logged, and the user is notified.

Block 8

```
# Cleans up conversation resources.
async def end_conversation(update: Update, context:
ContextTypes.DEFAULT_TYPE):
    user = update.message.from_user
    USER_SESSIONS.pop(user.id, None)
```

```
    await update.message.reply_text("💬 Session ended. Start again with
/start.")
```

Explanation

The end_conversation function clears the user's session by removing their data from USER_SESSIONS, ensuring a fresh start for the next interaction. It then sends a message confirming that the session has ended and instructs the user to restart with the /start command if they want to process another invoice.

Block 9

```
# Main application setup. Initializes the bot and registers handlers.
def main():
    application =
Application.builder().token(os.getenv("TELEGRAM_BOT_TOKEN")).read_timeout(
30).write_timeout(30).build()

    # Register handlers
    handlers = [
        CommandHandler('start', start),
        CommandHandler('ask', ask_question),
        CommandHandler('end', end_conversation),
        MessageHandler(filters.Document.ALL, handle_file)
    ]

    for handler in handlers:
        application.add_handler(handler)

    # Start the bot in polling mode
    application.run_polling()


if __name__ == "__main__":
    main()
```

Explanation

The main function sets up and starts the Telegram bot. It initializes the bot with the API token stored in environment variables and configures timeout settings. Command handlers for /start, /ask, and /end are registered, along with a message handler to process uploaded files. The bot runs in polling mode, continuously checking for new messages and responding to user interactions.

Project Demonstration :-

To experience the Invoice Assistant in action, scan the QR code below. This will open the Telegram bot, where you can upload an invoice and ask questions about it.

How to Use the Bot:

1. Scan the QR code to open the bot in Telegram.
2. Click Start or type /start to begin.
3. Upload an invoice as a PDF or image (JPG, PNG).
4. Wait for the bot to process and extract details.
5. Ask any questions about the invoice using /ask <your question>.
6. Type /end to close the session.

Scan the QR Code Below to Start the Demo

Fig 11.1 Telegram Bot's QR Code

Results :-

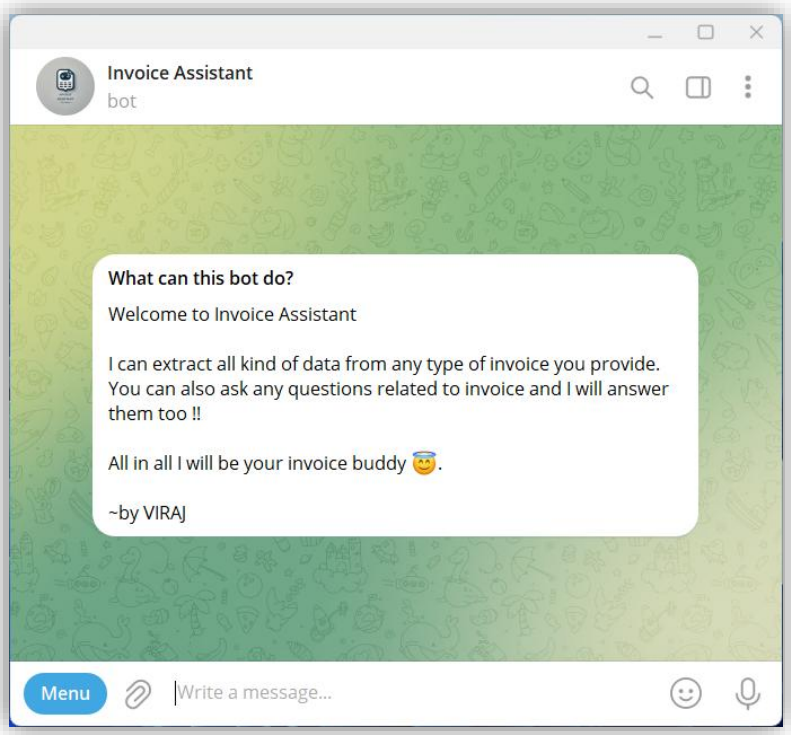

Fig 11.2 Telegram Bot Screenshot 1

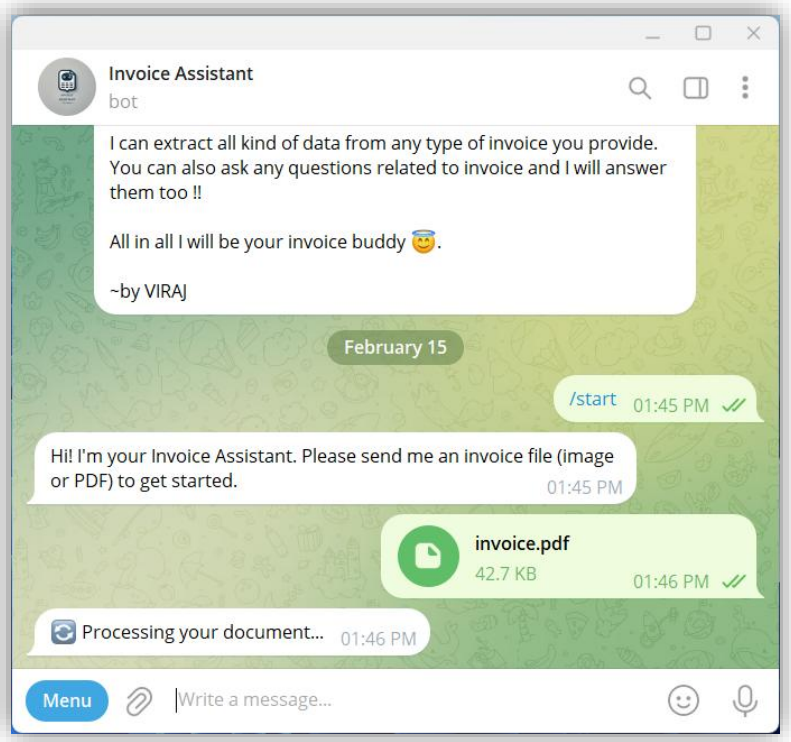

Fig 11.3 Telegram Bot Screenshot 2

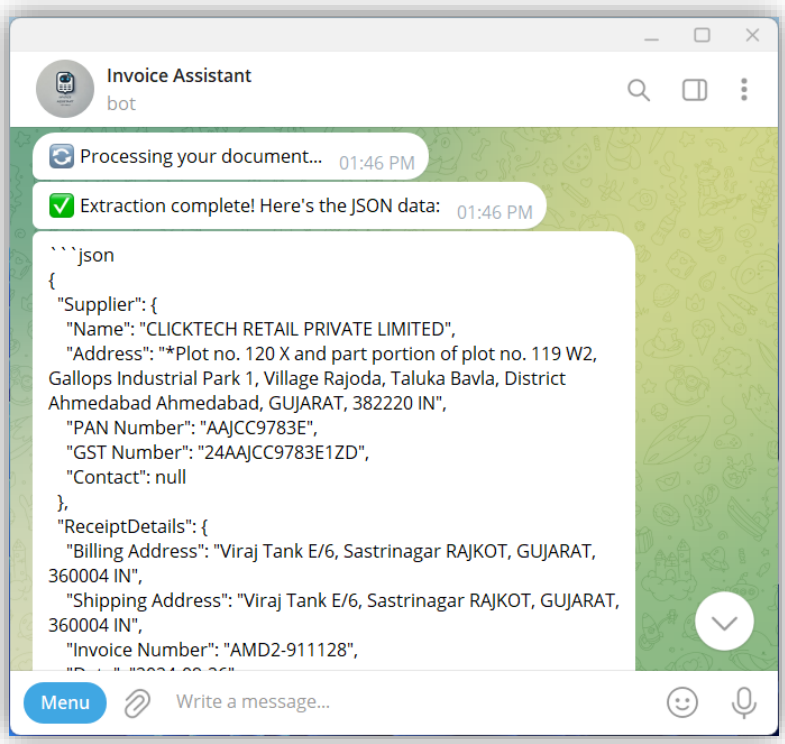

Fig 11.4 Telegram Bot Screenshot 3

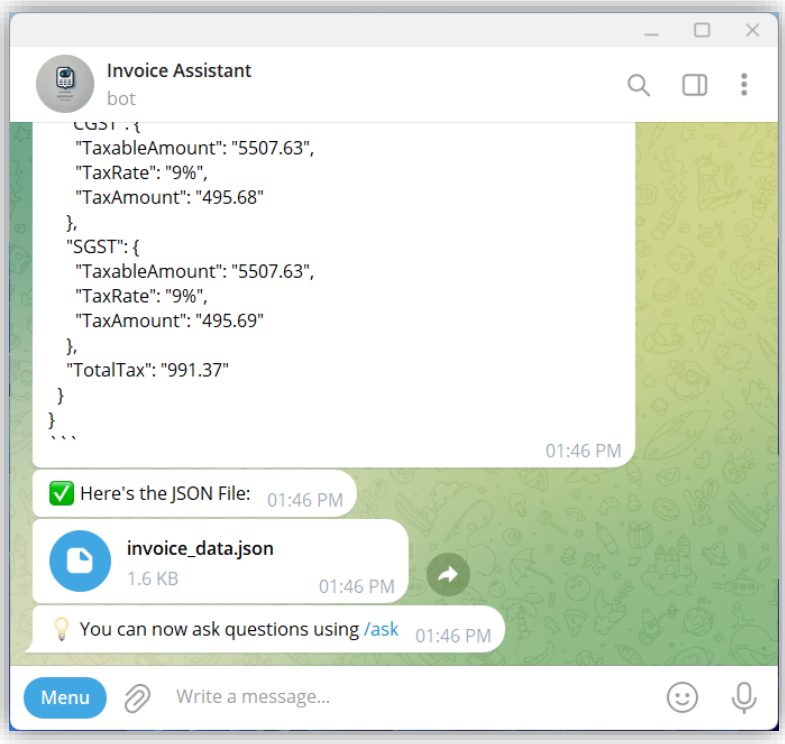

Fig 11.5 Telegram Bot Screenshot 4

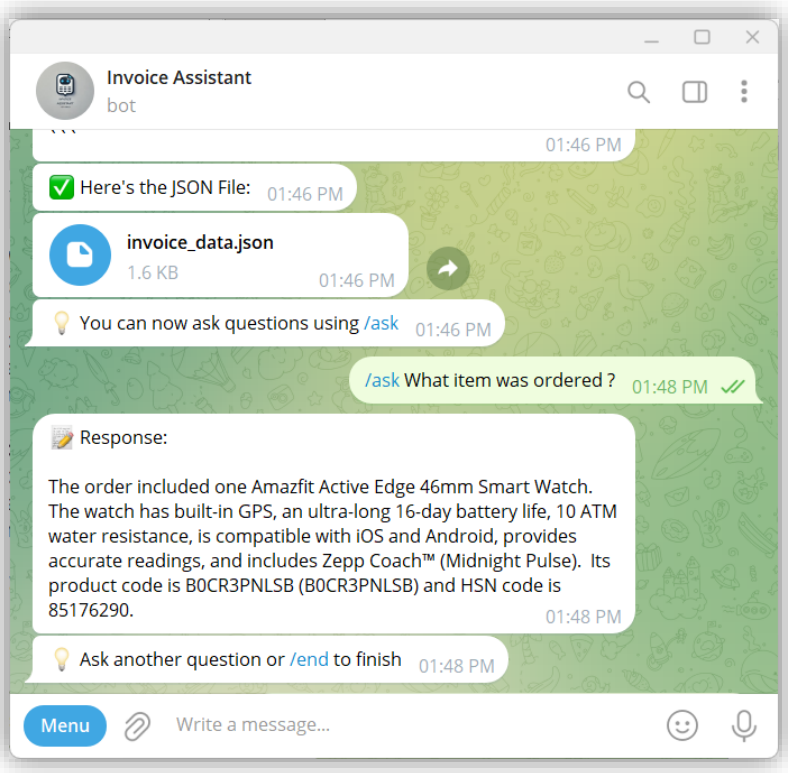

Fig 11.6 Telegram Bot Screenshot 5

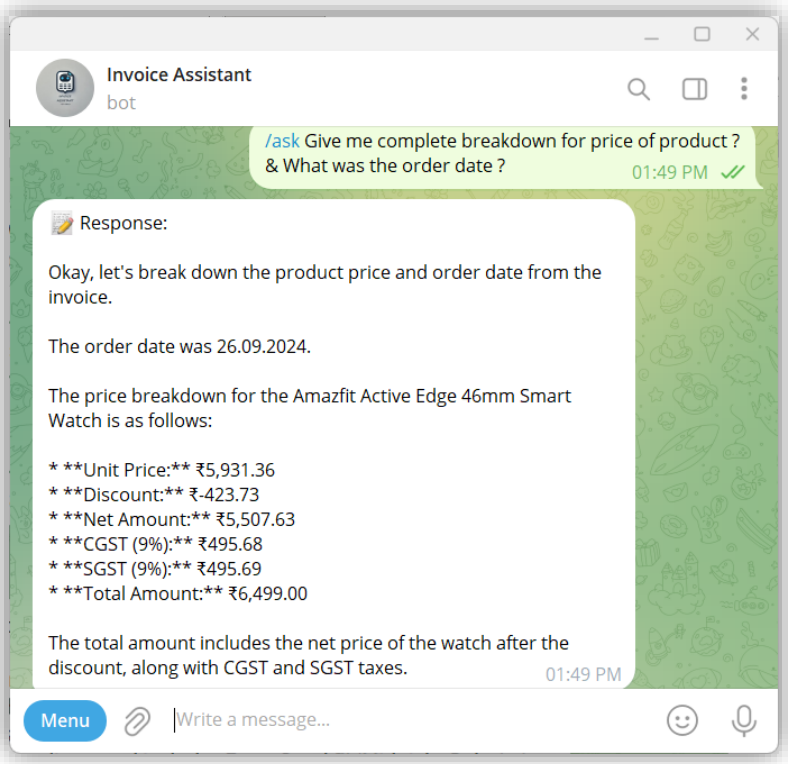

Fig 11.7 Telegram Bot Screenshot 6

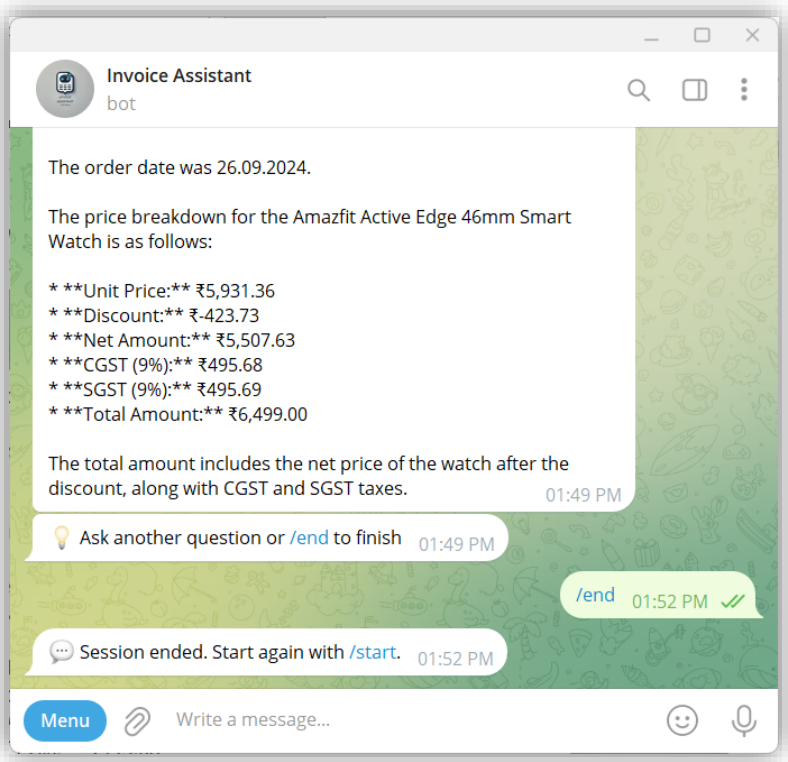

Fig 11.8 Telegram Bot Screenshot 7

Observation :-

- The bot successfully extracts and organizes invoice details into a structured JSON format.
- Google Gemini provides accurate results, even for different invoice layouts.
- When providing multiple invoices at a time it extracts data from all invoices but answers only to the last invoice uploaded.
- Sometimes when there is connection issue we have retry mechanism which ensures smooth operation.
- Logging helps in debugging issues and tracking user interactions efficiently.
- Some invoices with unclear text or poor image quality may result in incomplete or inaccurate extraction.
- In this we can still optimise the input prompts for further accuracy for complex layouts.

Conclusion :-

The Invoice Assistant makes it easier to extract and analyse invoice data by automatically pulling structured details from PDFs and images. With Google Gemini, the system can process different invoice formats and accurately answer user queries. Features like retry mechanisms and logging help keep the bot running smoothly, even when network issues occur.

It's also important to note that the bot runs only while the code is actively running in VS Code; if the code is terminated, the bot stops responding. For a more robust deployment, platforms like AWS, Google Cloud Platform, Microsoft Azure, or similar services could be used.

That said, there are a few limitations. The accuracy of extraction depends on how clear the uploaded invoice is, blurry or low-quality images can lead to missing details. The free version of Google Gemini also has request limits, which might slow things down when processing multi-page invoices. Additionally, the bot has difficulty handling handwritten invoices or those with unusual layouts, which may require further improvements in OCR technology.

Despite these challenges, the Invoice Assistant is a useful tool for managing invoices efficiently, reducing manual work, and improving overall accuracy.