Photo by Matteo Di Iorio on Unsplash

Building a Receipt Text Extraction App with Python and LLM: Unleashing the Power of LLMs

Build a cutting-edge receipt text extraction app using Streamlit, Flask, Django, Fast API, or Gradio

Senthil E · Following

Published in Level Up Coding · 49 min read · Apr 9, 2024

Introduction:

In this article, we'll dive into the world of receipt text extraction and explore how you can harness the capabilities of Python and OpenAI's API to build a cutting-edge app that simplifies your life. Whether you're a business owner looking to streamline your expense tracking process or a developer eager to explore the potential of LLMs, this article has something for you.

We'll start by discussing the concept of receipt text extraction and its significance in today's digital landscape. Then, we'll introduce you to OpenAI's powerful Language Model and how it can be leveraged to extract meaningful information from receipt images. Along the way, we'll showcase how Python, with its rich ecosystem of libraries and frameworks, serves as the perfect companion for building such an app.

But that's not all! We'll take things a step further by demonstrating how you can deploy your receipt text extraction app using popular web frameworks like Streamlit, Flask, or FastAPI. These frameworks provide a seamless and intuitive way to create interactive web interfaces, making your app accessible and user-friendly.

So, whether you're a seasoned Python developer or just starting your journey into the world of machine learning, this article will equip you with the knowledge and tools necessary to build a powerful receipt text extraction app. Get ready to unlock the secrets hidden within receipts and discover how Python, OpenAI, and web frameworks can revolutionize your data entry process.

Let’s dive in.

Contents:

Traditional Approaches to Receipt Text Extraction:

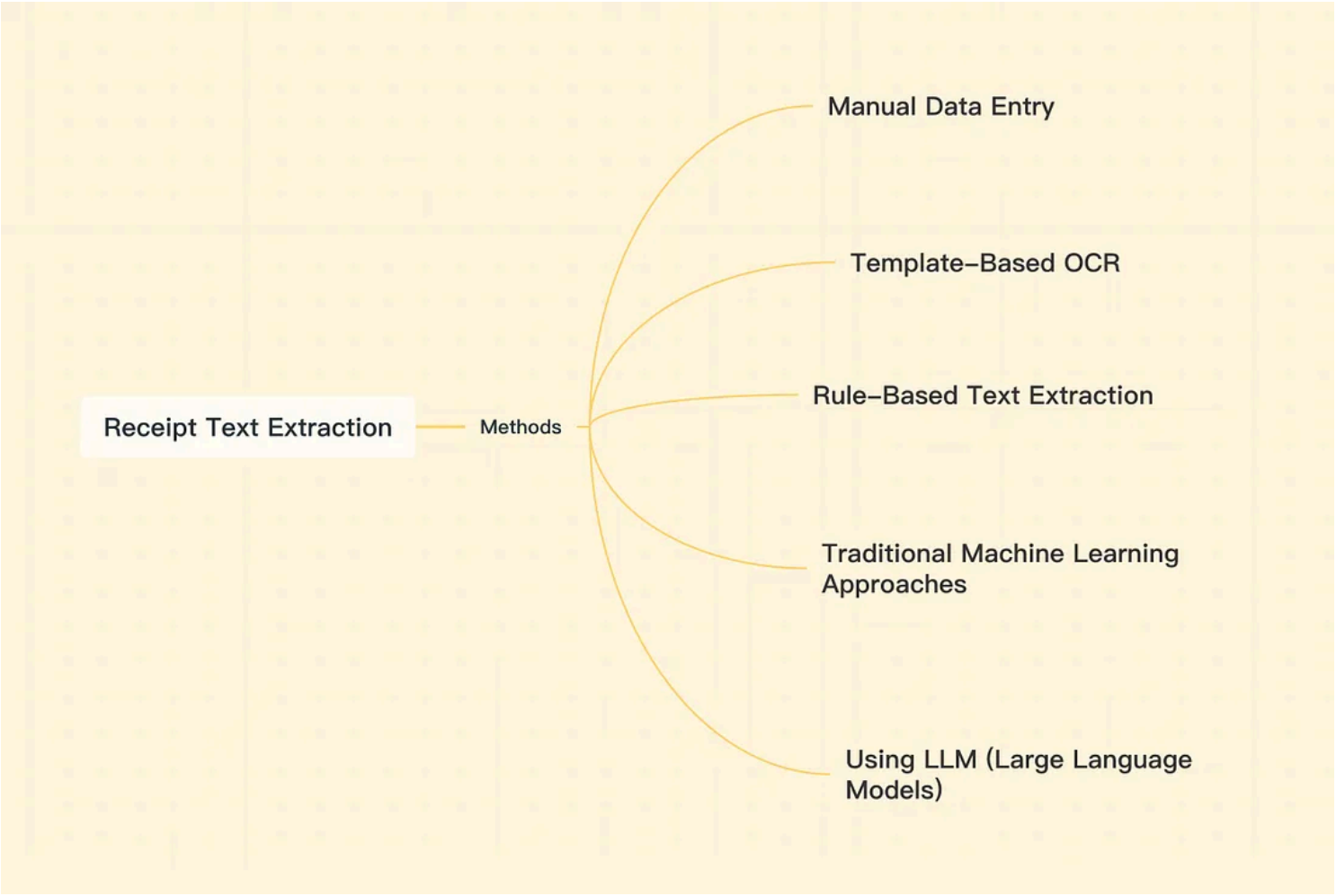


Image by the Author

1. Manual Data Entry:

— **Description:** Manually typing in the information from receipts into a database or spreadsheet.
— **Limitations:**
— Time-consuming and labor-intensive process, especially for large volumes of receipts.
— Prone to human errors, such as typos or misinterpretation of handwriting.
— Difficult to scale and handle increasing amounts of receipts.
— Requires significant human resources and can be costly.

2. Template-Based OCR (Optical Character Recognition):

— **Description:** Using pre-defined templates to extract text from receipts based on the expected layout and structure.
— **Limitations:**
— Relies on consistent and standardized receipt formats, which is rarely the case in real-world scenarios.
— Struggles with variations in receipt layouts, such as different fonts, spacing, or orientations.

formats.

```
# Pseudo code for template-based OCR receipt text extraction

# Import necessary libraries
import cv2  # OpenCV for image processing
import pytesseract  # Tesseract OCR library
import re  # Regular expressions for pattern matching

# Define the template for the receipt layout
receipt_template = {
    "merchant_name": {"x": 100, "y": 50, "width": 200, "height": 30},
    "transaction_date": {"x": 100, "y": 100, "width": 150, "height": 30},
    "total_amount": {"x": 100, "y": 200, "width": 100, "height": 30},
    # Define other fields based on the expected layout
}

# Load the receipt image
receipt_image = cv2.imread("receipt.jpg")

# Preprocess the image (e.g., resize, convert to grayscale, apply threshold)
preprocessed_image = preprocess_image(receipt_image)

# Extract text from the preprocessed image using Tesseract OCR
extracted_text = pytesseract.image_to_string(preprocessed_image)

# Initialize a dictionary to store the extracted fields
extracted_fields = {}

# Iterate over the defined template fields
```

```
for field, coordinates in receipt_template.items():
    # Extract the region of interest (ROI) based on the coordinates
    x, y, width, height = coordinates.values()
    roi = preprocessed_image[y:y+height, x:x+width]

    # Apply OCR to the ROI
    field_text = pytesseract.image_to_string(roi)

    # Clean and process the extracted text
    cleaned_text = clean_text(field_text)

    # Store the extracted field in the dictionary
    extracted_fields[field] = cleaned_text

# Perform additional processing or validation on the extracted fields
# ...

# Output the extracted fields
print(extracted_fields)
```

Python Libraries for OCR:

Tesseract OCR (pytesseract):

Tesseract is an open-source OCR engine that can be used with Python through the pytesseract library.

It supports a wide range of languages and provides accurate text recognition.

Website: https://github.com/madmaze/pytesseract

OpenCV (cv2):

OpenCV is a popular computer vision library that provides various image processing functionalities.

It can be used for preprocessing images before applying OCR.

Website: https://opencv.org/

EasyOCR:

EasyOCR is a Python library that provides a simple and easy-to-use interface for OCR.

It supports multiple languages and offers good accuracy.

Website: https://github.com/JaidedAI/EasyOCR

Google Cloud Vision API:
Google Cloud Vision API is a cloud-based OCR service provided by Google.
It offers accurate text recognition and supports multiple languages.
Website: https://cloud.google.com/vision
Amazon Textract:
Amazon Textract is a cloud-based OCR service provided by Amazon Web Services (AWS).
It offers advanced features like key-value pair extraction and table recognition.
Website: https://aws.amazon.com/textract/

3.Rule-Based Text Extraction:

— **Description:** Defining a set of rules and regular expressions to extract specific information from receipts based on patterns and keywords.
— **Limitations:**
— Requires extensive domain knowledge and manual effort to define and maintain the rules.
— Rules can become complex and difficult to manage as the variety of receipt formats increases.
— Struggles with handling variations in terminology, abbreviations, or language used in receipts.
— Limited scalability and adaptability to new receipt formats or changes in existing ones.

```
# Pseudo code for rule-based receipt text extraction

# Import necessary libraries
import re  # Regular expressions for pattern matching
import spacy  # Natural Language Processing (NLP) library

# Define rules and patterns for extracting specific information
merchant_name_pattern = r"Merchant Name:\s*(.+)"
```

```
transaction_date_pattern = r"Date:\s*(\d{2}/\d{2}/\d{4})"
total_amount_pattern = r"Total Amount:\s*(\$\d+\.\d{2})"
# Define other rules and patterns based on the receipt format

# Load the receipt text
receipt_text = """
Merchant Name: ABC Store
Date: 05/23/2023
Item 1: $10.99
Item 2: $5.50
Total Amount: $16.49
"""

# Initialize a dictionary to store the extracted fields
extracted_fields = {}

# Apply rules and patterns to extract specific information
merchant_name_match = re.search(merchant_name_pattern, receipt_text)
if merchant_name_match:
    extracted_fields["merchant_name"] = merchant_name_match.group(1)

transaction_date_match = re.search(transaction_date_pattern, receipt_text)
if transaction_date_match:
    extracted_fields["transaction_date"] = transaction_date_match.group(1)

total_amount_match = re.search(total_amount_pattern, receipt_text)
if total_amount_match:
    extracted_fields["total_amount"] = total_amount_match.group(1)

# Perform additional text processing and extraction using NLP techniques
nlp = spacy.load("en_core_web_sm")  # Load the English language model
doc = nlp(receipt_text)

# Extract entities using named entity recognition (NER)
for entity in doc.ents:
    if entity.label_ == "ORG":
        extracted_fields["merchant_name"] = entity.text
    elif entity.label_ == "DATE":
        extracted_fields["transaction_date"] = entity.text
    elif entity.label_ == "MONEY":
        extracted_fields["total_amount"] = entity.text

# Perform additional processing or validation on the extracted fields
# ...

# Output the extracted fields
print(extracted_fields)
```

Python Libraries for Rule-Based Text Extraction:

Regular Expressions (re):

The re module in Python provides support for regular expressions, which are powerful tools for pattern matching and text extraction.

Regular expressions allow you to define patterns and rules to extract specific information from text.

Website: https://docs.python.org/3/library/re.html

spaCy:

spaCy is a popular natural language processing (NLP) library for Python.

It offers various NLP capabilities, including named entity recognition (NER), part-of-speech tagging, and dependency parsing.

spaCy can be used to extract entities and perform advanced text processing tasks.

Website: https://spacy.io/

NLTK (Natural Language Toolkit):

NLTK is another widely used NLP library for Python.

It provides a range of tools and resources for text processing, including tokenization, stemming, and named entity recognition.

NLTK can be used in conjunction with regular expressions for rule-based text extraction.

Website: https://www.nltk.org/

TextBlob:

TextBlob is a Python library that provides a simple API for performing various NLP tasks.

It offers functionalities like sentiment analysis, part-of-speech tagging, and noun phrase extraction.

TextBlob can be used to enhance rule-based text extraction by leveraging its NLP capabilities.

Website: https://textblob.readthedocs.io/

fuzzywuzzy:

fuzzywuzzy is a Python library for string matching and comparison.
It can be used to handle variations in terminology, abbreviations, or language used in receipts.
fuzzywuzzy provides functions for fuzzy string matching, which can help in extracting information even when there are slight variations in the text.
Website: https://github.com/seatgeek/fuzzywuzzy4.

Traditional Machine Learning Approaches:

— **Description:** Training machine learning models, such as Support Vector Machines (SVM) or Random Forests, on annotated receipt datasets to extract text.
— **Limitations:**
— Requires a large annotated dataset of receipts, which can be time-consuming and expensive to obtain.
— Relies on handcrafted features and manual feature engineering, which may not capture all the relevant information.
— Limited generalization ability to handle unseen receipt formats or variations.
— Requires retraining the models whenever new receipt formats or features are introduced.

```
# Pseudo code for traditional machine learning receipt text extraction

# Import necessary libraries
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Prepare the annotated dataset
annotated_receipts = [
    {"text": "Merchant Name: ABC Store\nDate: 05/23/2023\nItem 1: $10.99\nItem 2
    # Add more annotated receipt examples
```

```
]

# Split the dataset into training and testing sets
train_receipts, test_receipts = train_test_split(annotated_receipts, test_size=0

# Extract features from the receipt text
vectorizer = CountVectorizer()  # or TfidfVectorizer()
train_features = vectorizer.fit_transform([receipt["text"] for receipt in train_
test_features = vectorizer.transform([receipt["text"] for receipt in test_receip

# Train machine learning models for each field
field_models = {}

for field in ["merchant_name", "transaction_date", "total_amount"]:
    # Prepare the training labels for the current field
    train_labels = [receipt["labels"].get(field, "") for receipt in train_receip

    # Train an SVM or Random Forest classifier for the current field
    model = SVC()  # or RandomForestClassifier()
    model.fit(train_features, train_labels)

    field_models[field] = model

# Evaluate the models on the testing set
for field, model in field_models.items():
    test_labels = [receipt["labels"].get(field, "") for receipt in test_receipts
    predicted_labels = model.predict(test_features)
    accuracy = accuracy_score(test_labels, predicted_labels)
    print(f"Accuracy for {field}: {accuracy}")

# Extract information from a new receipt using the trained models
new_receipt_text = "Merchant Name: XYZ Store\nDate: 06/01/2023\nItem 1: $15.99\n
new_receipt_features = vectorizer.transform([new_receipt_text])

extracted_fields = {}
for field, model in field_models.items():
    predicted_label = model.predict(new_receipt_features)
    extracted_fields[field] = predicted_label[0]

# Output the extracted fields
print(extracted_fields)
```

Python Libraries for Traditional Machine Learning Approaches:

scikit-learn (sklearn):
scikit-learn is a widely used Python library for machine learning tasks.
It provides a comprehensive set of tools for data preprocessing, feature extraction, model training, and evaluation.
scikit-learn offers various machine learning algorithms, including Support Vector Machines (SVM), Random Forests, and more.
Website: https://scikit-learn.org/
NLTK (Natural Language Toolkit):
NLTK is a popular Python library for natural language processing (NLP) tasks.
It provides utilities for text preprocessing, tokenization, stemming, and feature extraction.
NLTK can be used in conjunction with scikit-learn for text-based machine learning tasks.
Website: https://www.nltk.org/
spaCy:
spaCy is another powerful NLP library for Python.
It offers advanced features for text preprocessing, named entity recognition, part-of-speech tagging, and more.
spaCy can be used to extract additional features from the receipt text to enhance the machine learning models.
Website: https://spacy.io/

Limitations of Traditional Approaches:

The traditional approaches to receipt text extraction suffer from several limitations that hinder their effectiveness and scalability:

1. Lack of Flexibility: Traditional approaches struggle to handle the wide variety of receipt formats and layouts encountered in real-world scenarios.

They often rely on fixed templates or rules, making them inflexible and difficult to adapt to new or unseen receipt formats.

2. Manual Effort and Domain Knowledge: Traditional approaches often require significant manual effort and domain expertise to define templates, rules, or features for text extraction. This process can be time-consuming and requires continuous updates and maintenance as receipt formats evolve.

3. Limited Scalability: As the volume and variety of receipts increase, traditional approaches face challenges in scaling efficiently. Manual data entry becomes impractical, and rule-based systems become complex and difficult to manage.

4. Sensitivity to Variations: Traditional approaches are sensitive to variations in receipt layouts, fonts, spacing, or terminology. They may struggle to accurately extract information when faced with inconsistencies or deviations from expected patterns.

5. Lack of Contextual Understanding: Traditional approaches often lack the ability to understand the contextual meaning and relationships between different elements in a receipt. They rely on predefined patterns and fail to capture the nuances and semantics of the text.

6. Limited Language Support: Traditional approaches may have limited support for multiple languages or may require separate models or rules for each language, making it challenging to process receipts from different regions or countries.

In contrast to these traditional approaches, the solution proposed in this article leverages the power of OpenAI's Language Model (LLM) to overcome

these limitations. LLMs, with their ability to understand and generate human-like text, offer a more flexible, scalable, and accurate approach to receipt text extraction. They can handle a wide variety of receipt formats, adapt to variations, and capture contextual information, providing a more robust and efficient solution compared to traditional methods.

Image by the Author

Text Extraction App Using Streamlit and OpenAI Vision:

Overview:

This Streamlit application is designed to process receipt images. It sends these images to an LLM model, which extracts detailed information from the receipts. Then, it stores this information in a database. The app has two main parts: uploading and processing receipt images, and displaying the extracted data.

Detailed Explanation:

Application Setup:

The app starts by setting up a title using Streamlit's `st.title()` function, displaying "Receipt Extractor" as the heading on the web page.

API Key and Database Configuration:

An API key for OpenAI is defined. This key is required to access OpenAI's services for processing the receipt images.
Database connection parameters are specified (like host, user, password, database name). These parameters are used to connect to a MySQL database where the receipt data will be stored.

Defining What to Extract

A long text (`var_for`) describes in detail what information the AI should extract from the receipt images, including store details, transaction information, and line item details. It also specifies the format in which this data should be structured.

Encoding Images

A function `encode_image(image)` is defined to convert uploaded images into a base64 string. This conversion is necessary because the image needs to be sent over the internet to OpenAI's API in a text format.

Processing Images:

The `process_image(image)` function takes an uploaded image, encodes it, and then sends a request to OpenAI's API with the encoded image and the instructions on what information to extract.
The response from OpenAI's API, which includes the extracted data in JSON format, is then processed. The function extracts the relevant data from this JSON and inserts it into the MySQL database.
The database insertion involves two steps: first, inserting the receipt header information, and then inserting details about each line item on the receipt.

Streamlit Interface for Uploading Receipts:

The app uses Streamlit's tab feature to organize its interface into two tabs: one for uploading receipts and another for displaying the extracted data.

In the “Upload Receipt” tab, users can upload an image of their receipt. Once uploaded, the app displays the image and processes it to extract data.
After processing, the app displays a success message and the extracted data in JSON format.

Displaying Extracted Data:

- In the “Display the Data” tab, the app connects to the MySQL database and retrieves all stored receipt data.
  It then displays this data in two tables: one for receipt headers (like store name, transaction ID, date, etc.) and another for line items (including SKU, description, and price).
  This allows users to see all the information that has been extracted and stored from uploaded receipt images.

The code

```
import streamlit as st
import base64
import requests
import mysql.connector
import json
from datetime import datetime

# Streamlit app title
st.title("Receipt Extractor")

# OpenAI API Key
api_key = "Your Password"

# Database connection parameters
db_config = {
    'host': 'localhost',
    'user': 'root',
    'password': 'Your Password',
    'database': 'receipts'
}
```

```
var_for = """Given the receipt image provided, extract all relevant information

{
  "receipt_headers": {
    "store_name": "",
    "slogan": "",
    "address": "",
    "store_manager": "",
    "phone_number": "",
    "transaction_id": "",
    "date": "",
    "time": "",
    "cashier": "",
    "subtotal": 0,
    "sales_tax": 0,
    "total": 0,
    "gift_card": 0,
    "charged_amount": 0,
    "card_type": "",
    "auth_code": "",
    "chip_read": "",
    "aid": "",
    "issuer": "",
    "policy_id": "",
    "expiration_date": "",
    "survey_message": "",
    "survey_website": "",
    "user_id": "",
    "password": "",
    "eligibility_note": ""
  },
  "line_items": [
    {
      "sku": "",
      "description": "",
      "details": "",
      "price": 0
    }
  ]
}"""

# Function to encode the image
def encode_image(image):
    return base64.b64encode(image.read()).decode('utf-8')

# Function to process the uploaded image and update the database
def process_image(image):
    base64_image = encode_image(image)

    headers = {
```

```
    "Content-Type": "application/json",
    "Authorization": f"Bearer {api_key}"
}

payload = {
    "model": "gpt-4-vision-preview",
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": var_for
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{base64_image}"
                    }
                }
            ]
        }
    ],
    "max_tokens": 2048
}

response = requests.post("https://api.openai.com/v1/chat/completions", heade
response_json = response.json()
receipt_data_str = response_json['choices'][0]['message']['content']

# Find the JSON string within the extracted content
receipt_data_json_str = receipt_data_str.split('```json')[1].split('```')[0]
receipt_data = json.loads(receipt_data_json_str)

# Connect to the database
conn = mysql.connector.connect(**db_config)
cursor = conn.cursor()

# Insert into receipt_headers
header_insert_query = """
INSERT INTO receipt_headers (store_name, slogan, address, store_manager, pho
VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
"""

header_info = receipt_data['receipt_headers']
line_items = receipt_data['line_items']

# Format date, time, and expiration_date
formatted_date = datetime.strptime(header_info['date'], '%m/%d/%y').strftime
formatted_time = datetime.strptime(header_info['time'], '%I:%M %p').strftime
```

```
    formatted_expiration_date = datetime.strptime(header_info['expiration_date']

    # Prepare header values
    header_values = (
        header_info['store_name'],
        header_info['slogan'],
        header_info['address'],
        header_info['store_manager'],
        header_info['phone_number'],
        header_info['transaction_id'],
        formatted_date,
        formatted_time,
        header_info['cashier'],
        header_info['subtotal'],
        header_info['sales_tax'],
        header_info['total'],
        header_info['gift_card'],
        header_info['charged_amount'],
        header_info['card_type'],
        header_info['auth_code'],
        header_info['chip_read'],
        header_info['aid'],
        header_info['issuer'],
        header_info['policy_id'],
        formatted_expiration_date,
        header_info['survey_message'],
        header_info['survey_website'],
        header_info['user_id'],
        header_info['password'],
        header_info['eligibility_note']
    )

    # Insert header values
    cursor.execute(header_insert_query, header_values)
    receipt_id = cursor.lastrowid

    # Prepare and insert line items
    line_item_insert_query = """
    INSERT INTO line_items (receipt_id, sku, description, details, price)
    VALUES (%s, %s, %s, %s, %s)
    """

    for item in line_items:
        price = float(item['price'])
        line_item_values = (
            receipt_id,
            item['sku'],
            item['description'],
            item.get('details', ''),
            price
```

```
            )

            cursor.execute(line_item_insert_query, line_item_values)

    # Commit and close the connection
    conn.commit()
    cursor.close()
    conn.close()

    return receipt_data

# Streamlit tabs
tab1, tab2 = st.tabs(["Upload Receipt", "Display the Data"])

with tab1:
    st.header("Upload Receipt")
    uploaded_file = st.file_uploader("Choose an image", type=["jpg", "jpeg", "pn

    if uploaded_file is not None:
        # Display the uploaded image
        st.image(uploaded_file, caption='Uploaded Receipt', use_column_width=Tru

        # Process the uploaded image
        receipt_data = process_image(uploaded_file)

        # Display success message
        st.success("Message received successfully from the LLM.")

        # Display the JSON output
        st.json(receipt_data)


with tab2:
    st.header("Display the Data")

    # Connect to the database
    conn = mysql.connector.connect(**db_config)
    cursor = conn.cursor()

    # Fetch all records from the receipt_headers table, excluding the time colum
    cursor.execute("SELECT store_name, slogan, address, store_manager, phone_num
    headers = cursor.fetchall()

    # Display the headers in a table
    st.subheader("Receipt Headers")
    st.table(headers)

    # Fetch and display all records from the line_items table
    st.subheader("Line Items")
    cursor.execute("SELECT * FROM line_items;")
```

```
    items = cursor.fetchall()
    st.table(items)

    # Close the cursor and the connection
    cursor.close()
    conn.close()
```

1. **Title Displayed:** “Receipt Extractor” appears as the main title on the web page.
   - **Image Upload:** Users can upload receipt images through a simple interface.
   - **Image Processing:**
   The uploaded receipt image is converted to a text format and sent to an AI service.
   The AI service extracts detailed information from the image based on predefined instructions.
   Data Storage:
   Extracted data is organized and stored in a MySQL database.
   — This includes both general receipt information and detailed line items.
   - **Data Display:**
   — Users can view a summary of extracted data directly after uploading an image.
   — All extracted data stored in the database can be viewed in tabular format on a separate tab.

Screenshots:

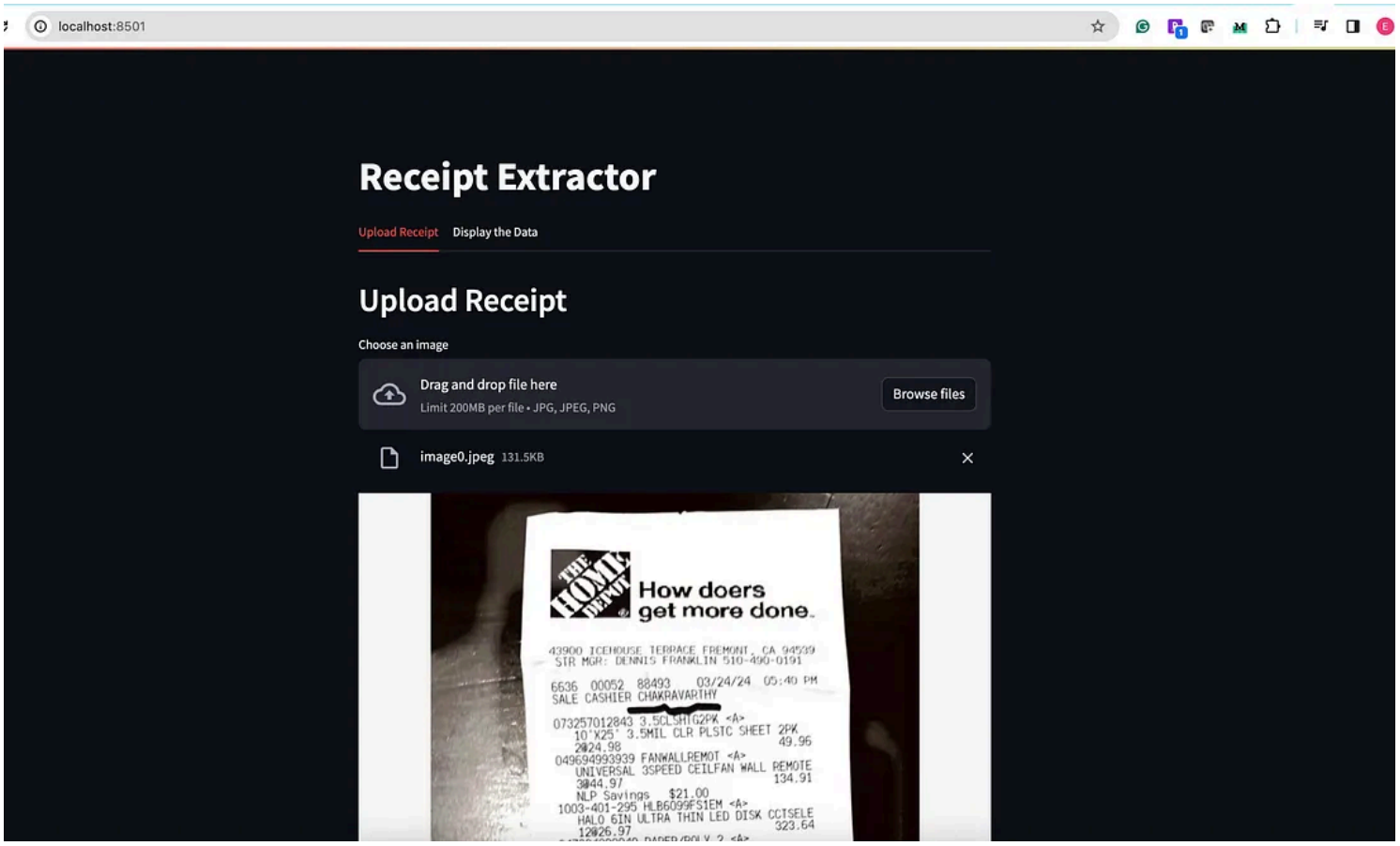


Image by the Author

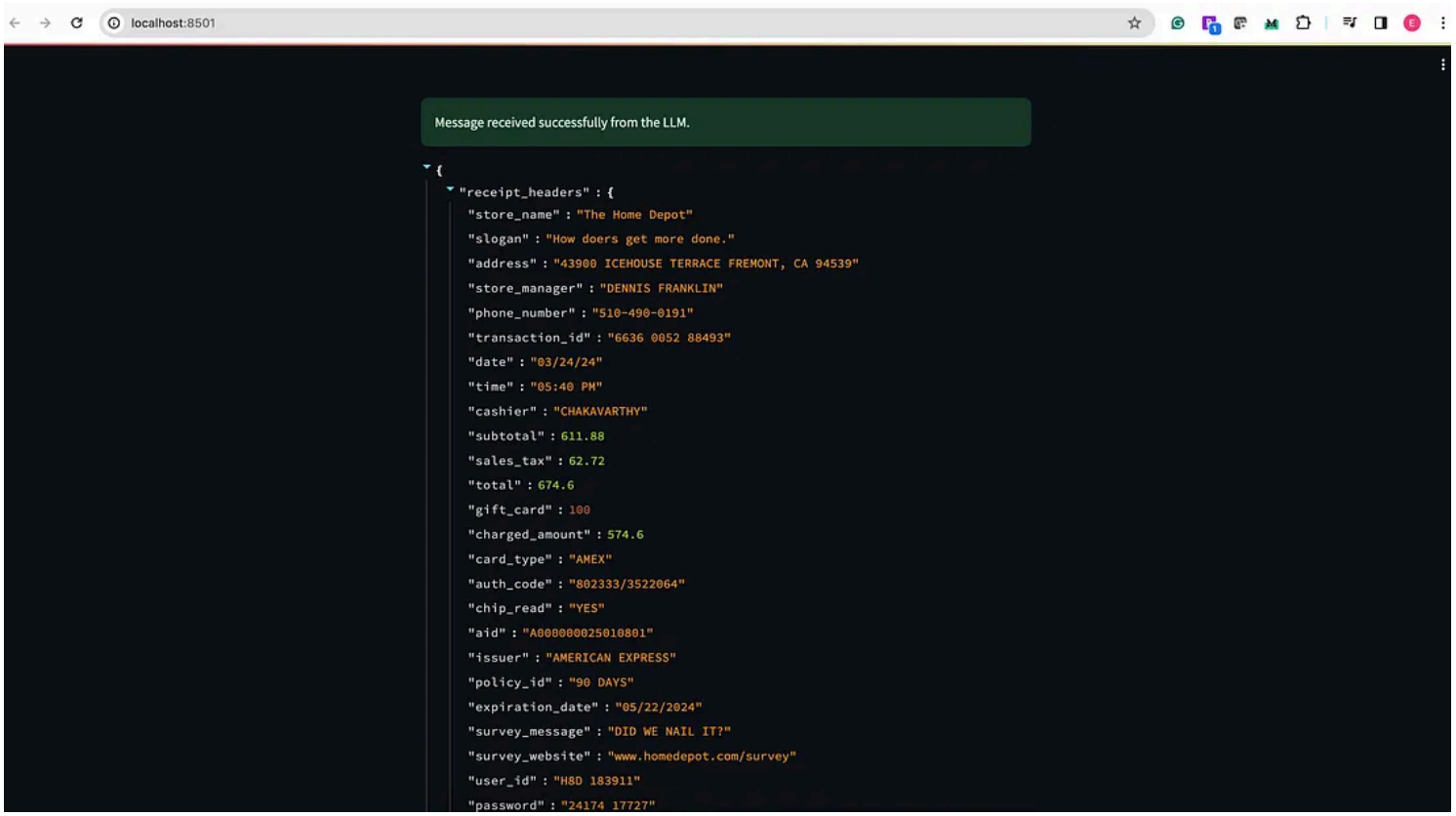


Image by the Author

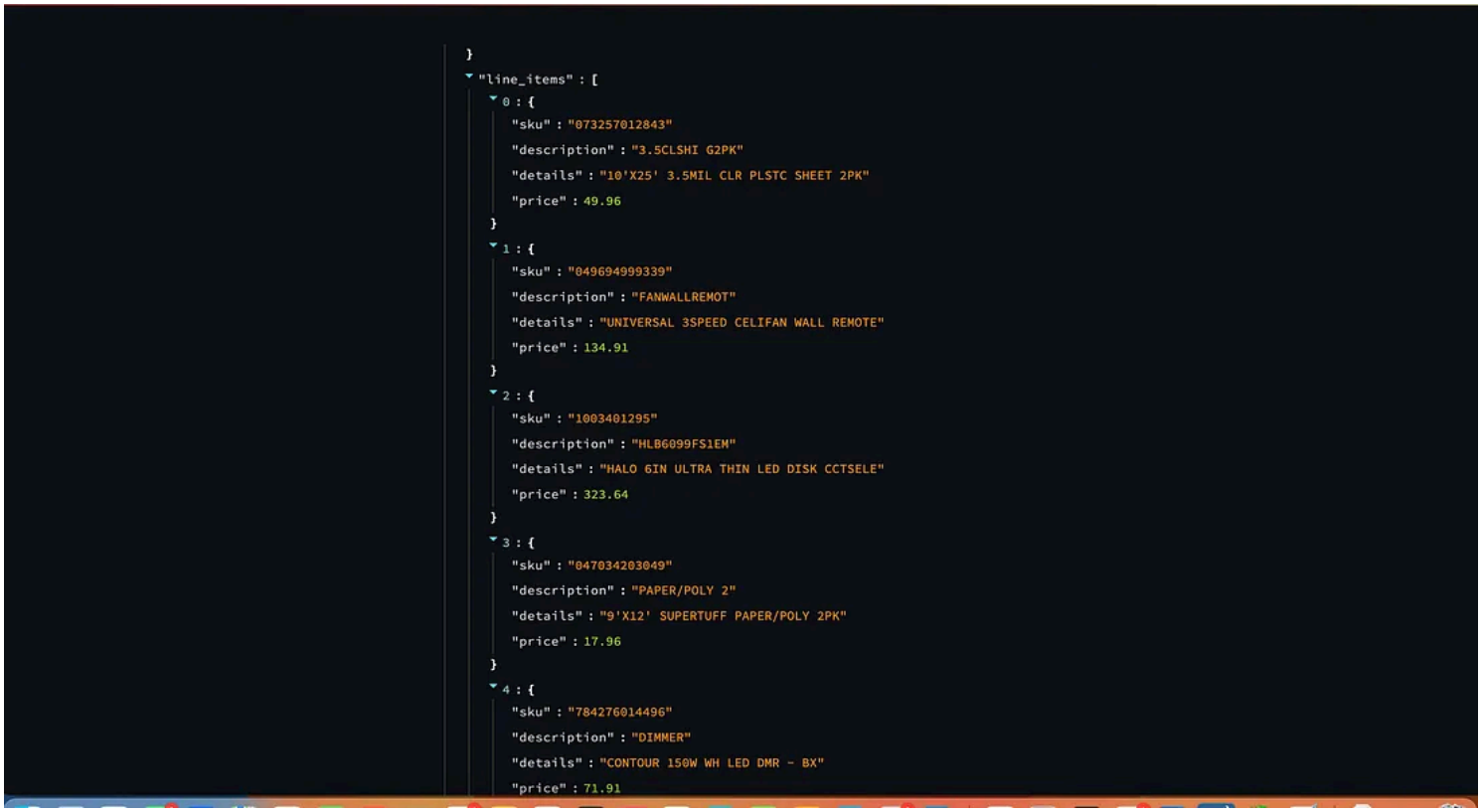


Image by the Author

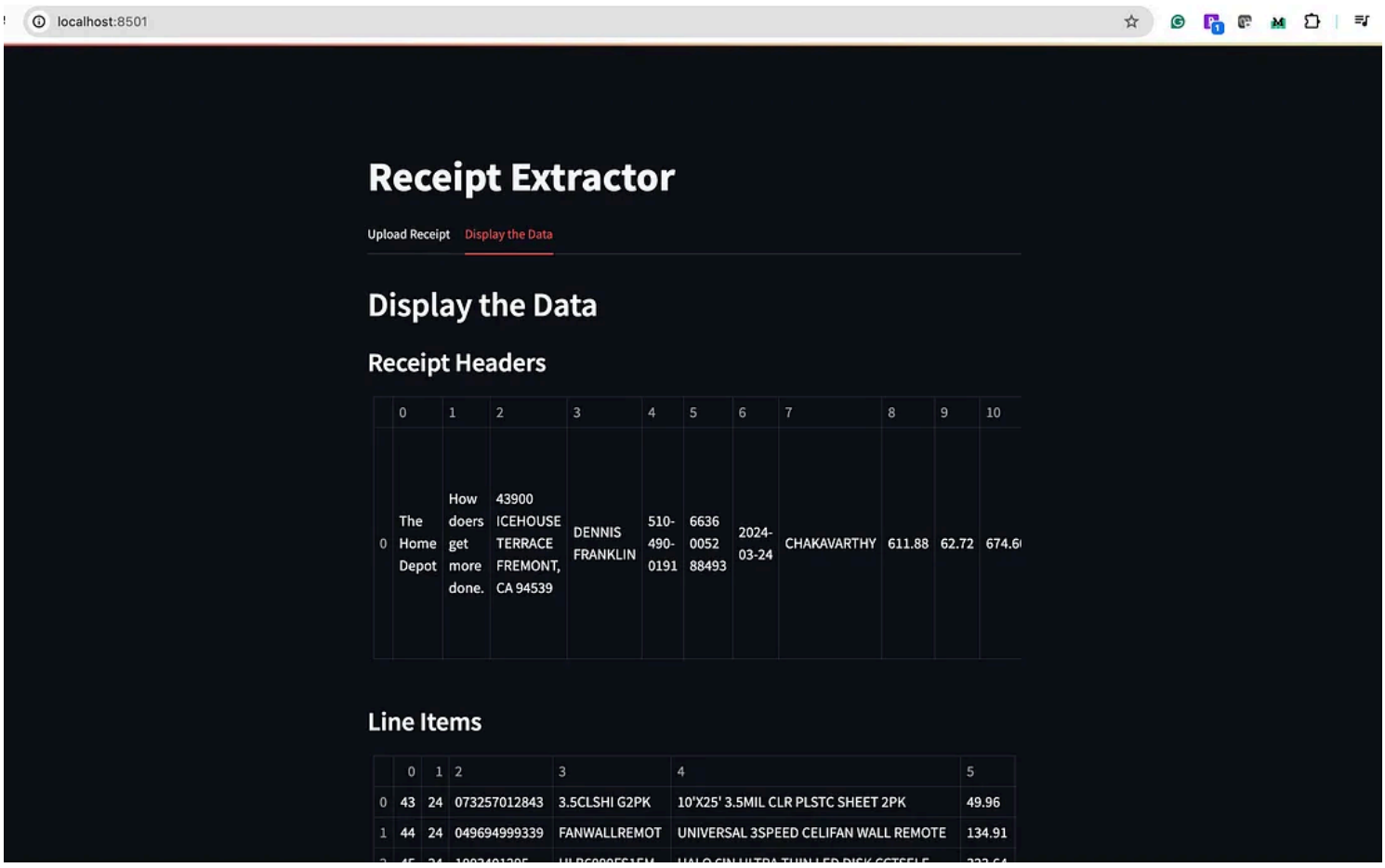


Image by the Author

	0	1	2	3	4	5	6	7	8	9	10
0	The Home Depot	How doers get more done.	43900 ICEHOUSE TERRACE FREMONT, CA 94539	DENNIS FRANKLIN	510-490-0191	6636 0052 88493	2024-03-24	CHAKAVARTHY	611.88	62.72	674.6

Line Items

	0	1	2	3	4	5
0	43	24	073257012843	3.5CLSHI G2PK	10'X25' 3.5MIL CLR PLSTC SHEET 2PK	49.96
1	44	24	049694999339	FANWALLREMOT	UNIVERSAL 3SPEED CELIFAN WALL REMOTE	134.91
2	45	24	1003401295	HLB6099FS1EM	HALO 6IN ULTRA THIN LED DISK CCTSELE	323.64
3	46	24	047034203049	PAPER/POLY 2	9'X12' SUPERTUFF PAPER/POLY 2PK	17.96
4	47	24	784276014496	DIMMER	CONTOUR 150W WH LED DMR - BX	71.91
5	48	24	078477295613	1G DEC SL WH	1G WHT SCREWLESS DECORA WALLPLT	13.50

Image by the Author

JSON Output:


```
{
  "receipt_headers": {
    "store_name": "The Home Depot",
    "slogan": "How doers get more done.",
    "address": "43900 ICEHOUSE TERRACE FREMONT, CA 94539",
    "store_manager": "DENNIS FRANKLIN",
    "phone_number": "510-490-0191",
    "transaction_id": "6636 0052 88493",
    "date": "03/24/24",
    "time": "05:40 PM",
    "cashier": "CHAKAVARTHY",
    "subtotal": 611.88,
    "sales_tax": 62.72,
    "total": 674.6,
    "gift_card": 100,
    "charged_amount": 574.6,
    "card_type": "AMEX",
```

```
    "auth_code": "802333/3522064",
    "chip_read": "YES",
    "aid": "A000000025010801",
    "issuer": "AMERICAN EXPRESS",
    "policy_id": "90 DAYS",
    "expiration_date": "05/22/2024",
    "survey_message": "DID WE NAIL IT?",
    "survey_website": "www.homedepot.com/survey",
    "user_id": "H8D 183911",
    "password": "24174 17727",
    "eligibility_note": "Entries must be completed within 14 days of purchase. E
  },
  "line_items": [
    {
      "sku": "073257012843",
      "description": "3.5CLSHI G2PK",
      "details": "10'X25' 3.5MIL CLR PLSTC SHEET 2PK",
      "price": 49.96
    },
    {
      "sku": "049694999339",
      "description": "FANWALLREMOT",
      "details": "UNIVERSAL 3SPEED CELIFAN WALL REMOTE",
      "price": 134.91
    },
    {
      "sku": "1003401295",
      "description": "HLB6099FS1EM",
      "details": "HALO 6IN ULTRA THIN LED DISK CCTSELE",
      "price": 323.64
    },
    {
      "sku": "047034203049",
      "description": "PAPER/POLY 2",
      "details": "9'X12' SUPERTUFF PAPER/POLY 2PK",
      "price": 17.96
    },
    {
      "sku": "784276014496",
      "description": "DIMMER",
      "details": "CONTOUR 150W WH LED DMR - BX",
      "price": 71.91
    },
    {
      "sku": "078477295613",
      "description": "1G DEC SL WH",
      "details": "1G WHT SCREWLESS DECORA WALLPLT",
      "price": 13.5
    }
  ]
```

```
}
```

MySQL Database:

```
import mysql.connector

# Database connection parameters
db_config = {
    'host': 'localhost',
    'user': 'root',
    'password': 'Your Password',  # Replace with your actual password
    'database': 'receipts'
}

# Connect to the database
conn = mysql.connector.connect(**db_config)
cursor = conn.cursor()

# Fetch and display all records from the receipt_headers table
print("Receipt Headers:")
cursor.execute("SELECT * FROM receipt_headers where receipt_id = '18';")
headers = cursor.fetchall()
for header in headers:
    print(header)

# Fetch and display all records from the line_items table
print("\nLine Items:")
cursor.execute("SELECT * FROM line_items;")
items = cursor.fetchall()
for item in items:
    print(item)

# Close the cursor and the connection
cursor.close()
conn.close()
```

```
Receipt Headers:
(18, 'The Home Depot', 'How doers get more done.', '43900 ICEHOUSE TERRACE FREMO
```

```
Line Items:
(7, 18, '073257012843', '3.5CLSHI 6P2K', "10'X25' 3.5MIL CLR PLSTC SHEET 2PK", D
(8, 18, '049694993939', 'FANWALLREMOT <A>', 'UNIVERSAL 3SPEED CEILFAN WALL REMOT
(9, 18, '1003-401-295', 'HLB6099FS1EM <A>', 'HALO 6IN ULTRA THIN LED DISK CCTSEL
(10, 18, '047034203049', 'PAPER/POLY 2 <A>', "9'X12' SUPERUFF PAPER/POLY 2PK", D
(11, 18, '784276014496', 'DIMMER <A>', 'CONTOUR 150W WH LED DMR - BX', Decimal('
(12, 18, '078477259613', '1G DEC SL WH <A>', '1G WHT SCREWLESS DECORA WALLPLT',
```

The tables are:

```
Table: receipt_headers

Columns:
receipt_id
int AI PK
store_name
varchar(255)
slogan
varchar(255)
address
varchar(255)
store_manager
varchar(255)
phone_number
varchar(50)
transaction_id
varchar(255)
date
date
time
time
cashier
varchar(255)
subtotal
decimal(10,2)
sales_tax
decimal(10,2)
total
decimal(10,2)
gift_card
decimal(10,2)
charged_amount
decimal(10,2)
```

```
card_type
varchar(50)
auth_code
varchar(50)
chip_read
varchar(50)
aid
varchar(50)
issuer
varchar(255)
policy_id
varchar(50)
expiration_date
date
survey_message
text
survey_website
varchar(255)
user_id
varchar(255)
password
varchar(255)
eligibility_note
text
```

```
Table: line_items

Columns:
line_item_id
int AI PK
receipt_id
int
sku
varchar(255)
description
varchar(255)
details
text
price
decimal(10,2)
```

App Steps:

1. User Input:
— The user accesses the Streamlit app through a web browser.
— The user interacts with the app's user interface to upload a receipt image file.

2. Image Upload:
— The uploaded receipt image is sent from the user's browser to the Streamlit app server.
— The Streamlit app receives the image file and stores it temporarily for processing.

3. Image Preprocessing:
— The uploaded receipt image undergoes preprocessing steps to prepare it for text extraction.
— Preprocessing steps may include resizing, cropping, noise reduction, or image enhancement techniques.
— The preprocessed image is ready to be sent to the OpenAI API for text extraction.

4. OpenAI API Integration:
— The Streamlit app sends a request to the OpenAI API, including the preprocessed receipt image and any necessary parameters or instructions.
— The request is authenticated using the provided API credentials.
— The OpenAI API receives the request and processes the image using its language model (LLM) for text extraction.

5. Text Extraction:
— The OpenAI API applies its LLM to analyze the receipt image and extract the relevant text information.
— The LLM uses techniques like optical character recognition (OCR), natural

language processing (NLP), and machine learning to identify and extract key data points from the receipt.
— The extracted text may include fields such as merchant name, transaction date, item descriptions, quantities, prices, and total amount.

6. API Response:
— The OpenAI API generates a structured response containing the extracted text information from the receipt.
— The response is typically in a machine-readable format, such as JSON.
— The API sends the response back to the Streamlit app server.

7. Data Processing:
— The Streamlit app receives the API response containing the extracted text data.
— The app processes the received data and performs any necessary formatting, parsing, or data transformation steps.
— The processed data is ready to be displayed to the user and stored in a database.

8. Data Storage:
— The Streamlit app connects to a database (e.g., MySQL, PostgreSQL, or MongoDB) to store the extracted receipt information.
— The app creates a new record in the database or updates an existing record with the extracted data.
— The stored data may include fields such as receipt ID, merchant name, transaction date, item details, and total amount.

9. User Display:
— The Streamlit app updates its user interface to display the extracted receipt information to the user.

— The displayed information may include the original receipt image, extracted text fields, and any additional insights or visualizations.
— The user can view, verify, and interact with the extracted receipt data within the app's interface.

10. Data Retrieval:
— If the user wants to view previously extracted receipt information, the Streamlit app retrieves the relevant data from the database.
— The app queries the database based on user-specified criteria (e.g., receipt ID, date range) and fetches the corresponding records.
— The retrieved data is processed and displayed to the user in a user-friendly format.

Open AI Vision Documentation:

https://platform.openai.com/docs/guides/vision

Streamlit Cheatsheet:

```
import streamlit as st

# Basics

# Title and header
st.title('My Streamlit App')
st.header('Welcome to my app!')

# Text and markdown
st.text('This is plain text.')
st.markdown('This is **markdown** text.')

# Data display
st.dataframe(my_dataframe)  # Display a DataFrame
st.table(my_dataframe)  # Display a static table

# Widgets
```

```python
# Button
if st.button('Click me'):
    st.write('Button clicked!')

# Checkbox
if st.checkbox('Show/Hide'):
    st.write('Content is visible.')

# Radio buttons
option = st.radio('Select an option', ('Option 1', 'Option 2', 'Option 3'))
st.write(f'You selected: {option}')

# Selectbox
option = st.selectbox('Select an option', ('Option 1', 'Option 2', 'Option 3'))
st.write(f'You selected: {option}')

# Multiselect
options = st.multiselect('Select multiple options', ('Option 1', 'Option 2', 'Op
st.write(f'You selected: {options}')

# Slider
value = st.slider('Select a value', min_value=0, max_value=100, value=50, step=1
st.write(f'You selected: {value}')

# Text input
text = st.text_input('Enter your name')
st.write(f'Hello, {text}!')

# Number input
number = st.number_input('Enter a number', min_value=0, max_value=100, value=50,
st.write(f'You entered: {number}')

# Date input
date = st.date_input('Select a date')
st.write(f'You selected: {date}')

# Time input
time = st.time_input('Select a time')
st.write(f'You selected: {time}')

# File uploader
uploaded_file = st.file_uploader('Choose a file')
if uploaded_file is not None:
    # Process the uploaded file
    pass

# Color picker
color = st.color_picker('Pick a color')
st.write(f'You selected: {color}')
```

```python
# Layout

# Sidebar
st.sidebar.title('Sidebar Title')
st.sidebar.button('Sidebar Button')

# Columns
col1, col2, col3 = st.columns(3)
with col1:
    st.header('Column 1')
    st.write('Content for column 1')
with col2:
    st.header('Column 2')
    st.write('Content for column 2')
with col3:
    st.header('Column 3')
    st.write('Content for column 3')

# Expander
with st.expander('Click to expand'):
    st.write('This content is hidden by default.')

# Container
with st.container():
    st.write('This content is inside a container.')

# Empty
st.empty()  # Create an empty placeholder

# Progress and Status

# Progress bar
progress_bar = st.progress(0)
for i in range(100):
    # Simulate progress
    progress_bar.progress(i + 1)

# Spinner
with st.spinner('Loading...'):
    # Simulate a long-running operation
    pass

# Success message
st.success('Operation successful!')

# Info message
st.info('This is an informational message.')

# Warning message
```

```
st.warning('This is a warning message.')

# Error message
st.error('This is an error message.')

# Exception
try:
    # Some code that may raise an exception
    pass
except Exception as e:
    st.exception(e)

# Plotting

# Line chart
st.line_chart(data)

# Area chart
st.area_chart(data)

# Bar chart
st.bar_chart(data)

# Pyplot
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.plot(data)
st.pyplot(fig)

# Altair
import altair as alt
chart = alt.Chart(data).mark_line().encode(x='date', y='value')
st.altair_chart(chart, use_container_width=True)

# Plotly
import plotly.express as px
fig = px.line(data, x='date', y='value')
st.plotly_chart(fig)

# Maps

# Map
st.map(data)

# Pydeck
import pydeck as pdk
st.pydeck_chart(pdk.Deck(
    initial_view_state=pdk.ViewState(
        latitude=37.76,
        longitude=-122.4,
```

```
        zoom=11,
        pitch=50,
    ),
    layers=[
        pdk.Layer(
            'HexagonLayer',
            data=data,
            get_position='[lon, lat]',
            radius=200,
            elevation_scale=4,
            elevation_range=[0, 1000],
            pickable=True,
            extruded=True,
        ),
    ],
))

# Media

# Image
st.image('image.jpg', caption='Image caption', use_column_width=True)

# Audio
st.audio('audio.mp3')

# Video
st.video('video.mp4')

# Caching

# Cache a function
@st.cache
def expensive_computation(a, b):
    # Perform expensive computation
    return result

# Cache a DataFrame
@st.cache
def load_data():
    data = pd.read_csv('data.csv')
    return data

# Callbacks

# Session state
if 'count' not in st.session_state:
    st.session_state.count = 0

def increment_counter():
    st.session_state.count += 1
```

```
st.button('Increment', on_click=increment_counter)
st.write(f'Count: {st.session_state.count}')

# Tips and Tricks

# Use st.write for simple output
st.write('Hello, World!')

# Use st.echo to display code and output
with st.echo():
    st.write('This code will be displayed along with its output.')

# Use st.stop to stop execution
if some_condition:
    st.write('Stopping execution.')
    st.stop()

# Use st.experimental_rerun to rerun the app
if st.button('Rerun'):
    st.experimental_rerun()

# Use st.experimental_singleton to create a singleton
@st.experimental_singleton
def get_database_connection():
    return db_connection

# Use st.experimental_memo to memoize a function
@st.experimental_memo
def expensive_computation(a, b):
    return result

# Do's and Don'ts

# Do: Use meaningful variable and function names
# Don't: Use single-letter or cryptic variable and function names

# Do: Break down your code into smaller, reusable functions
# Don't: Write long, monolithic functions

# Do: Use caching when dealing with expensive computations or loading data
# Don't: Recompute expensive operations unnecessarily

# Do: Handle exceptions and display informative error messages
# Don't: Let exceptions go unhandled, leading to a broken app

# Do: Use layout components to organize your app's UI
# Don't: Clutter your app with unorganized widgets

# Do: Provide clear instructions and guidance to users
```

```
# Don't: Assume users will know how to use your app without any guidance

# Do: Test your app thoroughly before deploying
# Don't: Deploy an untested app with potential bugs or errors

# Do: Keep your app's design simple and intuitive
# Don't: Overcomplicate the UI with unnecessary elements or confusing layouts

# Do: Optimize your app's performance by minimizing the use of expensive operati
# Don't: Ignore performance considerations, leading to a slow and unresponsive a

# Do: Regularly update and maintain your app
# Don't: Neglect your app after deployment, leading to outdated or broken functi
```

Python Web Frameworks:

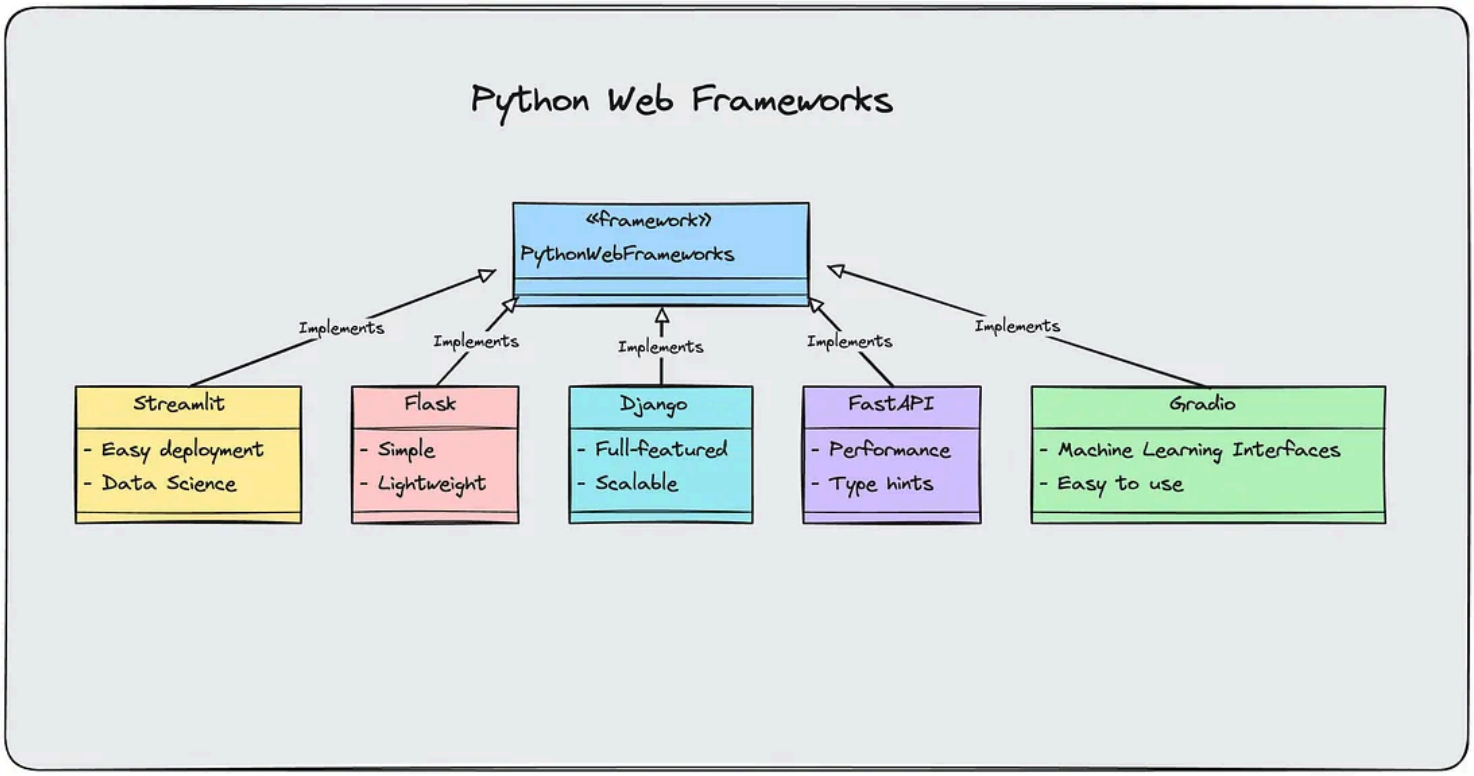


Image by the Author

Image by the Author

Streamlit:

— Streamlit is an open-source Python library that allows you to create interactive web applications quickly and easily.
— It is designed to simplify the process of building data-driven applications and enables you to create beautiful and responsive user interfaces with minimal coding.
— Streamlit provides a simple and intuitive API for building interactive widgets, charts, tables, and more.
— Documentation: https://docs.streamlit.io/
— GitHub Repository: https://github.com/streamlit/streamlit

Python Flask:

— Flask is a lightweight web framework for Python that allows you to build web applications quickly and with minimal overhead.

— It provides a simple and intuitive API for handling routes, requests, and responses, making it easy to create RESTful APIs and web applications.
— Flask follows a modular design, allowing you to choose and integrate various extensions based on your project's requirements.
— Documentation: https://flask.palletsprojects.com/
— GitHub Repository: https://github.com/pallets/flask

FastAPI:

— FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.
— It leverages the power of asynchronous programming and provides automatic API documentation using OpenAPI (Swagger) and JSON Schema.
— FastAPI is designed to be easy to use, fast to code, and suitable for production environments.
— Documentation: https://fastapi.tiangolo.com/
— GitHub Repository: https://github.com/tiangolo/fastapi

Django:

— Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.
— It follows the model-template-view (MTV) architectural pattern and provides a rich set of features out of the box, such as an ORM, authentication, admin interface, and more.
— Django emphasizes reusability and "don't repeat yourself" (DRY) principles, making it easier to build scalable and maintainable web applications.
— Documentation: https://docs.djangoproject.com/
— GitHub Repository: https://github.com/django/django

Gradio:

— Gradio is an open-source Python library that allows you to quickly create web interfaces for your machine learning models, APIs, or any Python function.
— It provides a simple and intuitive way to create interactive demos, visualizations, and GUI components for your projects.
— Gradio supports various input and output types, including text, image, audio, video, and more, making it suitable for a wide range of applications.
— Documentation: https://gradio.app/docs/
— GitHub Repository: https://github.com/gradio-app/gradio

Other Python libraries similar to FastAPI or Flask:

Falcon:

A lightweight and fast web framework for building APIs and microservices.
— GitHub Repository: https://github.com/falconry/falcon

Bottle:

A simple and lightweight WSGI web framework for Python.
— GitHub Repository: https://github.com/bottlepy/bottle

Hug:

A Python framework for building APIs and command-line tools quickly and easily.
— GitHub Repository: https://github.com/hugapi/hug

Sanic:

A fast web framework and server for Python, built on top of uvloop and designed for high performance.
— GitHub Repository: https://github.com/sanic-org/sanic

Lets convert the streamlit code into flask, fast API, etc.

Image by the Author

Python Flask:

The streamlit code is converted to Python Flask. I tested only the Streamlit. This code should work.Just a migration from Streamlit to Python Flask.

```
from flask import Flask, render_template, request
import base64
import requests
import mysql.connector
import json
from datetime import datetime

app = Flask(__name__)

# OpenAI API Key
api_key = "Your password"

# Database connection parameters
db_config = {
```

```
    'host': 'localhost',
    'user': 'root',
    'password': 'Your Password',
    'database': 'receipts'
}

# Function to encode the image
def encode_image(image_file):
    return base64.b64encode(image_file.read()).decode('utf-8')

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        # Get the uploaded image file
        image_file = request.files['image']

        # Encode the image
        base64_image = encode_image(image_file)

        headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {api_key}"
        }

        var_for = """Given the receipt image provided, extract all relevant info

        {
          "receipt_headers": {
            "store_name": "",
            "slogan": "",
            "address": "",
            "store_manager": "",
            "phone_number": "",
            "transaction_id": "",
            "date": "",
            "time": "",
            "cashier": "",
            "subtotal": 0,
            "sales_tax": 0,
            "total": 0,
            "gift_card": 0,
            "charged_amount": 0,
            "card_type": "",
            "auth_code": "",
            "chip_read": "",
            "aid": "",
            "issuer": "",
            "policy_id": "",
            "expiration_date": "",
            "survey_message": "",
```

```
    "survey_website": "",
    "user_id": "",
    "password": "",
    "eligibility_note": ""
  },
  "line_items": [
    {
      "sku": "",
      "description": "",
      "details": "",
      "price": 0
    }
  ]
}"""

payload = {
    "model": "gpt-4-vision-preview",
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": var_for
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{base64_image}"
                    }
                }
            ]
        }
    ],
    "max_tokens": 2048
}

response = requests.post("https://api.openai.com/v1/chat/completions", h

response_json = response.json()
receipt_data_str = response_json['choices'][0]['message']['content']

# Find the JSON string within the extracted content
receipt_data_json_str = receipt_data_str.split('```json')[1].split('```'

# Convert the JSON string to a Python dictionary
receipt_data = json.loads(receipt_data_json_str)

# Connect to the database
conn = mysql.connector.connect(**db_config)
```

```
cursor = conn.cursor()

# Insert into receipt_headers
header_insert_query = """
INSERT INTO receipt_headers (store_name, slogan, address, store_manager,
VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
"""

header_info = receipt_data['receipt_headers']
line_items = receipt_data['line_items']

# Convert date to 'YYYY-MM-DD' format
date_str = header_info['date']
date_obj = datetime.strptime(date_str, '%m/%d/%y')
formatted_date = date_obj.strftime('%Y-%m-%d')

# Convert time to 24-hour format
time_str = header_info['time']
time_obj = datetime.strptime(time_str, '%I:%M %p')
formatted_time = time_obj.strftime('%H:%M:%S')

# Convert expiration_date to 'YYYY-MM-DD' format
expiration_date_str = header_info['expiration_date']
expiration_date_obj = datetime.strptime(expiration_date_str, '%m/%d/%Y')
formatted_expiration_date = expiration_date_obj.strftime('%Y-%m-%d')

# Prepare header values
header_values = (
    header_info['store_name'],
    header_info['slogan'],
    header_info['address'],
    header_info['store_manager'],
    header_info['phone_number'],
    header_info['transaction_id'],
    formatted_date,
    formatted_time,
    header_info['cashier'],
    header_info['subtotal'],
    header_info['sales_tax'],
    header_info['total'],
    header_info['gift_card'],
    header_info['charged_amount'],
    header_info['card_type'],
    header_info['auth_code'],
    header_info['chip_read'],
    header_info['aid'],
    header_info['issuer'],
    header_info['policy_id'],
    formatted_expiration_date,
    header_info['survey_message'],
```

```
            header_info['survey_website'],
            header_info['user_id'],
            header_info['password'],
            header_info['eligibility_note']
        )

        # Insert header values
        cursor.execute(header_insert_query, header_values)
        receipt_id = cursor.lastrowid  # Get the last inserted id to use for lin

        # Prepare and insert line items
        line_item_insert_query = """
        INSERT INTO line_items (receipt_id, sku, description, details, price)
        VALUES (%s, %s, %s, %s, %s)
        """

        for item in line_items:
            price = float(item['price'])
            line_item_values = (
                receipt_id,
                item['sku'],
                item['description'],
                item.get('details', ''),
                price
            )

            cursor.execute(line_item_insert_query, line_item_values)

        # Commit and close the connection
        conn.commit()
        cursor.close()
        conn.close()

        return "Receipt processed successfully!"

    return render_template('index.html')

if __name__ == '__main__':
    app.run(debug=True)
```

In this Flask app:

1. We import the necessary modules, including Flask, render_template, and request from the flask package.
2. We create a Flask app instance using app = Flask(__name__).
3. We define the OpenAI API key and the database connection parameters.
4. We define the encode_image function to encode the uploaded image file.
5. We define the index route that handles both GET and POST requests.
6. In the POST request handling:
   * We retrieve the uploaded image file using request.files['image'].
   * We encode the image using the encode_image function.
   * We set the headers and payload for the API request.
   * We send a POST request to the OpenAI API to extract receipt information.
   * We extract the JSON string from the API response and convert it to a Python dictionary.
   * We connect to the MySQL database using the provided configuration.
   * We insert the receipt headers and line items into the respective tables in the database.
   * We commit the changes and close the database connection.
   * We return a success message.
7. In the GET request handling, we render the index.html template.
8. Finally, we run the Flask app using app.run(debug=True).
   Note: Make sure to create an index.html template file in the templates directory to handle the rendering of the HTML page.

Also, ensure that you have the necessary dependencies installed, such as flask, requests, and mysql-connector-python.

This Flask app provides a web interface for uploading a receipt image, processing it using the OpenAI API, and storing the extracted information in a MySQL database.

Python Flask Cheatsheet:

```
from flask import Flask, render_template, request, redirect, url_for, session, j

app = Flask(__name__)
app.secret_key = 'your_secret_key'

# Basics

# Hello World
@app.route('/')
def hello():
    return 'Hello, World!'

# Routing
@app.route('/home')
def home():
    return 'Welcome to the home page!'

# Dynamic Routing
@app.route('/user/<username>')
def show_user_profile(username):
    return f'User {username}'

# HTTP Methods
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # Handle POST request
        pass
    else:
        # Handle GET request
        pass

# Templates
@app.route('/')
def index():
    return render_template('index.html')
```

```
# Template Inheritance
# base.html
'''
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
'''

# index.html
'''
{% extends 'base.html' %}

{% block title %}Home{% endblock %}

{% block content %}
    <h1>Welcome to the home page!</h1>
{% endblock %}
'''

# Static Files
# Include CSS, JavaScript, and image files in the 'static' folder

# Example usage in template:
'''
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<script src="{{ url_for('static', filename='script.js') }}"></script>
<img src="{{ url_for('static', filename='image.jpg') }}">
'''

# Request Object
@app.route('/submit', methods=['POST'])
def submit():
    name = request.form['name']
    email = request.form['email']
    # Process form data
    pass

# File Upload
from werkzeug.utils import secure_filename

@app.route('/upload', methods=['POST'])
def upload_file():
    file = request.files['file']
    filename = secure_filename(file.filename)
```

```
    file.save(filename)
    # Process uploaded file
    pass

# Redirection
@app.route('/redirect')
def redirect_example():
    return redirect(url_for('home'))

# Session
@app.route('/set_session')
def set_session():
    session['username'] = 'john_doe'
    return 'Session set'

@app.route('/get_session')
def get_session():
    username = session.get('username')
    return f'Username: {username}'

# Message Flashing
from flask import flash

@app.route('/flash_message')
def flash_message():
    flash('This is a flashed message', 'info')
    return redirect(url_for('index'))

# In the template:
'''
{% with messages = get_flashed_messages() %}
    {% if messages %}
        <ul>
            {% for message in messages %}
                <li>{{ message }}</li>
            {% endfor %}
        </ul>
    {% endif %}
{% endwith %}
'''

# Error Handling
@app.errorhandler(404)
def page_not_found(error):
    return render_template('404.html'), 404

# JSON Response
@app.route('/json')
def json_response():
    data = {'name': 'John', 'age': 30}
```

```
    return jsonify(data)

# Database Integration (SQLAlchemy)
from flask_sqlalchemy import SQLAlchemy

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///example.db'
db = SQLAlchemy(app)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

# Form Handling (Flask-WTF)
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired

class MyForm(FlaskForm):
    name = StringField('Name', validators=[DataRequired()])
    submit = SubmitField('Submit')

@app.route('/form', methods=['GET', 'POST'])
def form():
    form = MyForm()
    if form.validate_on_submit():
        # Process form data
        pass
    return render_template('form.html', form=form)

# Authentication (Flask-Login)
from flask_login import LoginManager, UserMixin, login_user, login_required, log

login_manager = LoginManager()
login_manager.init_app(app)

class User(UserMixin, db.Model):
    # User model definition
    pass

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

@app.route('/login', methods=['GET', 'POST'])
def login():
    # Login logic
    pass

@app.route('/logout')
```

```
@login_required
def logout():
    logout_user()
    return redirect(url_for('index'))

# Tips and Tricks

# Use meaningful variable and function names
# Break down your code into smaller, reusable functions
# Use template inheritance to avoid code duplication
# Handle errors gracefully and provide informative error pages
# Use Flask extensions for common functionalities (e.g., forms, authentication)
# Protect sensitive information (e.g., secret keys, database credentials)
# Validate and sanitize user input to prevent security vulnerabilities
# Use caching for expensive operations to improve performance
# Implement proper authentication and authorization mechanisms
# Follow PEP 8 guidelines for code style and consistency

# Do's and Don'ts

# Do: Use version control (e.g., Git) to manage your code
# Don't: Commit sensitive information (e.g., secrets, passwords) to version cont

# Do: Use a virtual environment to isolate project dependencies
# Don't: Install packages globally that are specific to a project

# Do: Handle exceptions and log errors for better debugging
# Don't: Ignore exceptions and let them propagate to the user

# Do: Use HTTPS for secure communication
# Don't: Transmit sensitive data over unencrypted connections

# Do: Implement proper authentication and authorization
# Don't: Trust user input without validation and sanitization

# Do: Use responsive design for mobile-friendly views
# Don't: Neglect the user experience on different devices

# Do: Optimize your application for performance
# Don't: Introduce unnecessary complexity or inefficiencies

# Do: Write unit tests to ensure code quality and prevent regressions
# Don't: Deploy untested code to production

# Do: Keep your dependencies up to date and address security vulnerabilities
# Don't: Use outdated or unsupported versions of libraries and frameworks

# Do: Provide clear and concise documentation for your application
# Don't: Assume users will understand how to use your application without guidan
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

Image by the Author

Fast API App:

The Streamlit code is converted into Fast API . Again I haven't tested this fast API code.

```
from fastapi import FastAPI, File, UploadFile
from fastapi.responses import JSONResponse
import base64
import requests
import mysql.connector
import json
from datetime import datetime

app = FastAPI()

# OpenAI API Key
api_key = "Your password"

# Database connection parameters
db_config = {
    'host': 'localhost',
    'user': 'root',
    'password': 'Your Password',
    'database': 'receipts'
```

```
}

# Function to encode the image
def encode_image(image_file):
    return base64.b64encode(image_file.read()).decode('utf-8')

@app.post("/process_receipt")
async def process_receipt(image: UploadFile = File(...)):
    # Read the uploaded image file
    image_file = await image.read()

    # Encode the image
    base64_image = encode_image(image_file)

    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {api_key}"
    }

    var_for = """Given the receipt image provided, extract all relevant informat

    {
      "receipt_headers": {
        "store_name": "",
        "slogan": "",
        "address": "",
        "store_manager": "",
        "phone_number": "",
        "transaction_id": "",
        "date": "",
        "time": "",
        "cashier": "",
        "subtotal": 0,
        "sales_tax": 0,
        "total": 0,
        "gift_card": 0,
        "charged_amount": 0,
        "card_type": "",
        "auth_code": "",
        "chip_read": "",
        "aid": "",
        "issuer": "",
        "policy_id": "",
        "expiration_date": "",
        "survey_message": "",
        "survey_website": "",
        "user_id": "",
        "password": "",
        "eligibility_note": ""
      },
```

```
  "line_items": [
    {
      "sku": "",
      "description": "",
      "details": "",
      "price": 0
    }
  ]
}"""

payload = {
    "model": "gpt-4-vision-preview",
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": var_for
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{base64_image}"
                    }
                }
            ]
        }
    ],
    "max_tokens": 2048
}

response = requests.post("https://api.openai.com/v1/chat/completions", heade

response_json = response.json()
receipt_data_str = response_json['choices'][0]['message']['content']

# Find the JSON string within the extracted content
receipt_data_json_str = receipt_data_str.split('```json')[1].split('```')[0]

# Convert the JSON string to a Python dictionary
receipt_data = json.loads(receipt_data_json_str)

# Connect to the database
conn = mysql.connector.connect(**db_config)
cursor = conn.cursor()

# Insert into receipt_headers
header_insert_query = """
INSERT INTO receipt_headers (store_name, slogan, address, store_manager, pho
```

```
    VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
    """

    header_info = receipt_data['receipt_headers']
    line_items = receipt_data['line_items']

    # Convert date to 'YYYY-MM-DD' format
    date_str = header_info['date']
    date_obj = datetime.strptime(date_str, '%m/%d/%y')
    formatted_date = date_obj.strftime('%Y-%m-%d')

    # Convert time to 24-hour format
    time_str = header_info['time']
    time_obj = datetime.strptime(time_str, '%I:%M %p')
    formatted_time = time_obj.strftime('%H:%M:%S')

    # Convert expiration_date to 'YYYY-MM-DD' format
    expiration_date_str = header_info['expiration_date']
    expiration_date_obj = datetime.strptime(expiration_date_str, '%m/%d/%Y')
    formatted_expiration_date = expiration_date_obj.strftime('%Y-%m-%d')

    # Prepare header values
    header_values = (
        header_info['store_name'],
        header_info['slogan'],
        header_info['address'],
        header_info['store_manager'],
        header_info['phone_number'],
        header_info['transaction_id'],
        formatted_date,
        formatted_time,
        header_info['cashier'],
        header_info['subtotal'],
        header_info['sales_tax'],
        header_info['total'],
        header_info['gift_card'],
        header_info['charged_amount'],
        header_info['card_type'],
        header_info['auth_code'],
        header_info['chip_read'],
        header_info['aid'],
        header_info['issuer'],
        header_info['policy_id'],
        formatted_expiration_date,
        header_info['survey_message'],
        header_info['survey_website'],
        header_info['user_id'],
        header_info['password'],
        header_info['eligibility_note']
    )
```

```
        # Insert header values
        cursor.execute(header_insert_query, header_values)
        receipt_id = cursor.lastrowid  # Get the last inserted id to use for line_it

        # Prepare and insert line items
        line_item_insert_query = """
        INSERT INTO line_items (receipt_id, sku, description, details, price)
        VALUES (%s, %s, %s, %s, %s)
        """

        for item in line_items:
            price = float(item['price'])
            line_item_values = (
                receipt_id,
                item['sku'],
                item['description'],
                item.get('details', ''),
                price
            )

            cursor.execute(line_item_insert_query, line_item_values)

        # Commit and close the connection
        conn.commit()
        cursor.close()
        conn.close()

        return JSONResponse(content={"message": "Receipt processed successfully!"})

    if __name__ == '__main__':
        import uvicorn
        uvicorn.run(app, host="0.0.0.0", port=8000)
```

In this FastAPI app:

1. We import the necessary modules, including FastAPI, File, UploadFile, and JSONResponse from the fastapi package.
2. We create a FastAPI app instance using app = FastAPI().
3. We define the OpenAI API key and the database connection parameters.
4. We define the encode_image function to encode the uploaded image file.

5. We define the /process_receipt endpoint using the @app.post() decorator, which accepts a POST request with an uploaded image file.

6. Inside the /process_receipt endpoint:
   * We read the uploaded image file using await image.read().
   * We encode the image using the encode_image function.
   * We set the headers and payload for the API request.
   * We send a POST request to the OpenAI API to extract receipt information.
   * We extract the JSON string from the API response and convert it to a Python dictionary.
   * We connect to the MySQL database using the provided configuration.
   * We insert the receipt headers and line items into the respective tables in the database.
   * We commit the changes and close the database connection.
   * We return a JSON response indicating the success of the receipt processing.

7. Finally, we run the FastAPI app using uvicorn.run(app, host="0.0.0.0", port=8000).
   Note: Make sure to have the necessary dependencies installed, such as fastapi, uvicorn, requests, and mysql-connector-python.

To use this FastAPI app, you can send a POST request to the /process_receipt endpoint with the receipt image file as the request body. The app will process the image using the OpenAI API, extract the receipt information, store it in the MySQL database, and return a success message in the JSON response.

Fast API Cheatsheet:

```python
from fastapi import FastAPI, Path, Query, Body, Header, Cookie, UploadFile, File
from fastapi.responses import JSONResponse, HTMLResponse, PlainTextResponse, Fil
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel, Field, EmailStr
from typing import List, Optional
from datetime import datetime

app = FastAPI()

# Basics

# Hello World
@app.get("/")
def hello():
    return {"message": "Hello, World!"}

# Path Parameters
@app.get("/items/{item_id}")
def read_item(item_id: int):
    return {"item_id": item_id}

# Query Parameters
@app.get("/items/")
def read_items(skip: int = 0, limit: int = 10):
    return {"skip": skip, "limit": limit}

# Request Body
class Item(BaseModel):
    name: str
    description: Optional[str] = None
    price: float
    tax: Optional[float] = None

@app.post("/items/")
def create_item(item: Item):
    return item

# Form Data
@app.post("/login/")
def login(username: str = Form(...), password: str = Form(...)):
    return {"username": username}

# File Upload
@app.post("/upload/")
def upload_file(file: UploadFile = File(...)):
    return {"filename": file.filename}
```

```
# Responses
@app.get("/text/")
def get_text():
    return "Hello, World!"

@app.get("/html/")
def get_html():
    return HTMLResponse("<h1>Hello, World!</h1>")

@app.get("/json/")
def get_json():
    return JSONResponse({"message": "Hello, World!"})

@app.get("/file/")
def get_file():
    return FileResponse("example.txt")

# Middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)

# Dependency Injection
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

def get_current_user(token: str = Depends(oauth2_scheme)):
    user = fake_decode_token(token)
    return user

@app.get("/users/me")
def read_users_me(current_user: User = Depends(get_current_user)):
    return current_user

# Authentication and Authorization
class User(BaseModel):
    username: str
    email: EmailStr
    full_name: Optional[str] = None
    disabled: Optional[bool] = None

def fake_decode_token(token):
    return User(
        username=token + "fakedecoded",
        email="john@example.com",
        full_name="John Doe"
    )
```

```python
@app.post("/token")
def login(form_data: OAuth2PasswordRequestForm = Depends()):
    user_dict = fake_users_db.get(form_data.username)
    if not user_dict:
        raise HTTPException(status_code=400, detail="Incorrect username or passw
    user = UserInDB(**user_dict)
    hashed_password = fake_hash_password(form_data.password)
    if not hashed_password == user.hashed_password:
        raise HTTPException(status_code=400, detail="Incorrect username or passw
    return {"access_token": user.username, "token_type": "bearer"}

# Background Tasks
from fastapi import BackgroundTasks

def write_notification(email: str, message=""):
    with open("log.txt", mode="w") as email_file:
        content = f"notification for {email}: {message}"
        email_file.write(content)

@app.post("/send-notification/{email}")
def send_notification(email: str, background_tasks: BackgroundTasks):
    background_tasks.add_task(write_notification, email, message="some notificat
    return {"message": "Notification sent in the background"}

# WebSocket
from fastapi import WebSocket

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
        await websocket.send_text(f"Message text was: {data}")

# GraphQL
from fastapi import Depends
from strawberry.asgi import GraphQL

@strawberry.type
class User:
    name: str
    age: int

@strawberry.type
class Query:
    @strawberry.field
    def user(self) -> User:
        return User(name="Patrick", age=100)

schema = strawberry.Schema(query=Query)
```

```
graphql_app = GraphQL(schema)

app.add_route("/graphql", graphql_app)
app.add_websocket_route("/graphql", graphql_app)

# Tips and Tricks

# Use meaningful variable and function names
# Leverage Python type hints for better code readability and error detection
# Use Pydantic models for data validation and serialization
# Utilize dependency injection for reusable and modular code
# Implement authentication and authorization for secure endpoints
# Use background tasks for long-running operations
# Implement WebSocket endpoints for real-time communication
# Explore GraphQL integration for flexible and efficient data fetching
# Use Docker for easy deployment and scalability
# Write tests to ensure code quality and prevent regressions

# Do's and Don'ts

# Do: Use ASGI server for production (e.g., Uvicorn, Hypercorn)
# Don't: Use the built-in development server for production

# Do: Use HTTPS for secure communication
# Don't: Expose sensitive data over unencrypted connections

# Do: Validate and sanitize user input to prevent security vulnerabilities
# Don't: Trust user input without proper validation

# Do: Use Pydantic models for data validation and serialization
# Don't: Manually parse and validate request data

# Do: Use dependency injection for reusable and modular code
# Don't: Repeat code or hard-code dependencies

# Do: Implement proper authentication and authorization mechanisms
# Don't: Leave endpoints unprotected or accessible to unauthorized users

# Do: Handle errors gracefully and return meaningful error responses
# Don't: Let exceptions propagate to the client or reveal sensitive information

# Do: Use background tasks for long-running operations
# Don't: Block the main thread with time-consuming tasks

# Do: Leverage asynchronous programming for better performance
# Don't: Use blocking I/O operations in async functions

# Do: Document your API endpoints and provide clear examples
# Don't: Assume users will understand how to use your API without documentation
```

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Image by the Author

Django App:

```
# views.py

from django.shortcuts import render
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
import base64
import requests
import mysql.connector
import json
from datetime import datetime

# OpenAI API Key
api_key = "Your password"

# Database connection parameters
db_config = {
    'host': 'localhost',
    'user': 'root',
    'password': 'Your Password',
```

```
    'database': 'receipts'
}

# Function to encode the image
def encode_image(image_file):
    return base64.b64encode(image_file.read()).decode('utf-8')

@csrf_exempt
def process_receipt(request):
    if request.method == 'POST':
        # Get the uploaded image file
        image_file = request.FILES['image']

        # Encode the image
        base64_image = encode_image(image_file)

        headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {api_key}"
        }

        var_for = """Given the receipt image provided, extract all relevant info

        {
          "receipt_headers": {
            "store_name": "",
            "slogan": "",
            "address": "",
            "store_manager": "",
            "phone_number": "",
            "transaction_id": "",
            "date": "",
            "time": "",
            "cashier": "",
            "subtotal": 0,
            "sales_tax": 0,
            "total": 0,
            "gift_card": 0,
            "charged_amount": 0,
            "card_type": "",
            "auth_code": "",
            "chip_read": "",
            "aid": "",
            "issuer": "",
            "policy_id": "",
            "expiration_date": "",
            "survey_message": "",
            "survey_website": "",
            "user_id": "",
            "password": "",
```

```
    "eligibility_note": ""
  },
  "line_items": [
    {
      "sku": "",
      "description": "",
      "details": "",
      "price": 0
    }
  ]
}"""

payload = {
    "model": "gpt-4-vision-preview",
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": var_for
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{base64_image}"
                    }
                }
            ]
        }
    ],
    "max_tokens": 2048
}

response = requests.post("https://api.openai.com/v1/chat/completions", h

response_json = response.json()
receipt_data_str = response_json['choices'][0]['message']['content']

# Find the JSON string within the extracted content
receipt_data_json_str = receipt_data_str.split('```json')[1].split('```'

# Convert the JSON string to a Python dictionary
receipt_data = json.loads(receipt_data_json_str)

# Connect to the database
conn = mysql.connector.connect(**db_config)
cursor = conn.cursor()

# Insert into receipt_headers
```

```
            header_insert_query = """
            INSERT INTO receipt_headers (store_name, slogan, address, store_manager,
            VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
            """

            header_info = receipt_data['receipt_headers']
            line_items = receipt_data['line_items']

            # Convert date to 'YYYY-MM-DD' format
            date_str = header_info['date']
            date_obj = datetime.strptime(date_str, '%m/%d/%y')
            formatted_date = date_obj.strftime('%Y-%m-%d')

            # Convert time to 24-hour format
            time_str = header_info['time']
            time_obj = datetime.strptime(time_str, '%I:%M %p')
            formatted_time = time_obj.strftime('%H:%M:%S')

            # Convert expiration_date to 'YYYY-MM-DD' format
            expiration_date_str = header_info['expiration_date']
            expiration_date_obj = datetime.strptime(expiration_date_str, '%m/%d/%Y')
            formatted_expiration_date = expiration_date_obj.strftime('%Y-%m-%d')

            # Prepare header values
            header_values = (
                header_info['store_name'],
                header_info['slogan'],
                header_info['address'],
                header_info['store_manager'],
                header_info['phone_number'],
                header_info['transaction_id'],
                formatted_date,
                formatted_time,
                header_info['cashier'],
                header_info['subtotal'],
                header_info['sales_tax'],
                header_info['total'],
                header_info['gift_card'],
                header_info['charged_amount'],
                header_info['card_type'],
                header_info['auth_code'],
                header_info['chip_read'],
                header_info['aid'],
                header_info['issuer'],
                header_info['policy_id'],
                formatted_expiration_date,
                header_info['survey_message'],
                header_info['survey_website'],
                header_info['user_id'],
                header_info['password'],
```

```
            header_info['eligibility_note']
        )

        # Insert header values
        cursor.execute(header_insert_query, header_values)
        receipt_id = cursor.lastrowid  # Get the last inserted id to use for lin

        # Prepare and insert line items
        line_item_insert_query = """
        INSERT INTO line_items (receipt_id, sku, description, details, price)
        VALUES (%s, %s, %s, %s, %s)
        """

        for item in line_items:
            price = float(item['price'])
            line_item_values = (
                receipt_id,
                item['sku'],
                item['description'],
                item.get('details', ''),
                price
            )

            cursor.execute(line_item_insert_query, line_item_values)

        # Commit and close the connection
        conn.commit()
        cursor.close()
        conn.close()

        return JsonResponse({"message": "Receipt processed successfully!"})

    return JsonResponse({"error": "Invalid request method"}, status=400)

# urls.py

from django.urls import path
from . import views

urlpatterns = [
    path('process_receipt/', views.process_receipt, name='process_receipt'),
]
```

In this Django app:

1. In the views.py file:
   * We import the necessary modules, including render, JsonResponse, csrf_exempt, and other required libraries.
   * We define the OpenAI API key and the database connection parameters.
   * We define the encode_image function to encode the uploaded image file.
   * We define the process_receipt view function, which is decorated with @csrf_exempt to allow POST requests without CSRF token validation.
   * Inside the process_receipt view:
   We check if the request method is POST.
   We retrieve the uploaded image file using request.FILES['image'].
   We encode the image using the encode_image function.
   We set the headers and payload for the API request.
   We send a POST request to the OpenAI API to extract receipt information.
   We extract the JSON string from the API response and convert it to a Python dictionary.
   We connect to the MySQL database using the provided configuration.
   We insert the receipt headers and line items into the respective tables i in the database.
   We commit the changes and close the database connection.
   We return a JSON response indicating the success of the receipt.
   processing.
   If the request method is not POST, we return a JSON response with an error message and a 400 status code.

2. In the urls.py file:
   We import the path function from django.urls and the views module.
   We define the URL pattern for the process_receipt view, mapping it to the /process_receipt/ URL path.
   Note: Make sure to have the necessary dependencies installed, such as django, requests, and mysql-connector-python.

To use this Django app, you can send a POST request to the /process_receipt/ URL with the receipt image file as part of the request data. The app will process the image using the OpenAI API, extract the receipt information, store it in the MySQL database, and return a success message in the JSON response.

Remember to integrate this code into your Django project structure and configure the necessary settings, such as database connections and URL patterns.

Django Cheatsheet:

```
# Basics

# Create a new Django project
django-admin startproject myproject

# Create a new Django app
python manage.py startapp myapp

# Run the development server
python manage.py runserver

# Create database migrations
python manage.py makemigrations

# Apply database migrations
python manage.py migrate

# Create a superuser
python manage.py createsuperuser

# Collect static files
python manage.py collectstatic

# URL Configuration

# myproject/urls.py
from django.urls import path, include
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('myapp/', include('myapp.urls')),
]

# myapp/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
    path('detail/<int:pk>/', views.detail, name='detail'),
]

# Views

# myapp/views.py
from django.shortcuts import render, get_object_or_404
from .models import MyModel

def index(request):
    objects = MyModel.objects.all()
    return render(request, 'index.html', {'objects': objects})

def detail(request, pk):
    object = get_object_or_404(MyModel, pk=pk)
    return render(request, 'detail.html', {'object': object})

# Models

# myapp/models.py
from django.db import models

class MyModel(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title

# Templates

# myapp/templates/index.html
<!DOCTYPE html>
<html>
<head>
    <title>Index Page</title>
</head>
```

```
<body>
    <h1>Objects</h1>
    <ul>
        {% for object in objects %}
            <li><a href="{% url 'detail' object.pk %}">{{ object.title }}</a></l
        {% endfor %}
    </ul>
</body>
</html>

# myapp/templates/detail.html
<!DOCTYPE html>
<html>
<head>
    <title>Detail Page</title>
</head>
<body>
    <h1>{{ object.title }}</h1>
    <p>{{ object.description }}</p>
    <p>Created at: {{ object.created_at }}</p>
</body>
</html>

# Forms

# myapp/forms.py
from django import forms
from .models import MyModel

class MyForm(forms.ModelForm):
    class Meta:
        model = MyModel
        fields = ['title', 'description']

# myapp/views.py
from .forms import MyForm

def create(request):
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('index')
    else:
        form = MyForm()
    return render(request, 'create.html', {'form': form})

# Authentication and Authorization

# myproject/settings.py
```

```
INSTALLED_APPS = [
    ...
    'django.contrib.auth',
    'django.contrib.contenttypes',
    ...
]

# myapp/views.py
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    ...

# Admin Interface

# myapp/admin.py
from django.contrib import admin
from .models import MyModel

admin.site.register(MyModel)

# Django REST Framework

# myproject/settings.py
INSTALLED_APPS = [
    ...
    'rest_framework',
]

# myapp/serializers.py
from rest_framework import serializers
from .models import MyModel

class MyModelSerializer(serializers.ModelSerializer):
    class Meta:
        model = MyModel
        fields = ['id', 'title', 'description', 'created_at']

# myapp/views.py
from rest_framework import viewsets
from .models import MyModel
from .serializers import MyModelSerializer

class MyModelViewSet(viewsets.ModelViewSet):
    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer

# Caching
```

```
# myproject/settings.py
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}

# myapp/views.py
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)  # Cache for 15 minutes
def my_view(request):
    ...

# Testing

# myapp/tests.py
from django.test import TestCase
from .models import MyModel

class MyModelTest(TestCase):
    def setUp(self):
        MyModel.objects.create(title='Test Object', description='This is a test'

    def test_my_model(self):
        obj = MyModel.objects.get(title='Test Object')
        self.assertEqual(obj.description, 'This is a test')

# Tips and Tricks

# Use meaningful app and model names
# Follow the Django coding style guide (PEP 8)
# Use Django's built-in authentication and authorization system
# Utilize Django's ORM for database queries
# Use Django's form validation and handling
# Take advantage of Django's template inheritance and tags
# Use Django's caching framework for improved performance
# Write unit tests to ensure code quality and prevent regressions
# Leverage Django REST framework for building APIs
# Consider using Django's class-based views for cleaner code

# Do's and Don'ts

# Do: Use virtual environments to isolate project dependencies
# Don't: Install packages globally that are specific to a project

# Do: Use version control (e.g., Git) to manage your code
# Don't: Commit sensitive information (e.g., secrets, passwords) to version cont
```

```
# Do: Follow the DRY (Don't Repeat Yourself) principle
# Don't: Duplicate code across multiple places

# Do: Use Django's built-in features and conventions when possible
# Don't: Reinvent the wheel or go against Django's conventions unnecessarily

# Do: Validate and sanitize user input to prevent security vulnerabilities
# Don't: Trust user input without proper validation and sanitization

# Do: Use Django's forms for handling user input and validation
# Don't: Manually process form data without utilizing Django's form functionalit

# Do: Use Django's ORM for database queries and operations
# Don't: Write raw SQL queries unless absolutely necessary

# Do: Implement proper authentication and authorization mechanisms
# Don't: Leave views and URLs accessible to unauthenticated or unauthorized user

# Do: Optimize database queries and use caching for improved performance
# Don't: Neglect performance considerations and query efficiency

# Do: Write tests to ensure code quality and catch regressions
# Don't: Deploy untested code to production
```

Image by the Author

Gradio App:

```
import gradio as gr
import base64
import requests
import mysql.connector
import json
from datetime import datetime

# OpenAI API Key
api_key = "Your password"

# Database connection parameters
db_config = {
    'host': 'localhost',
    'user': 'root',
    'password': 'Your Password',
    'database': 'receipts'
}

# Function to encode the image
def encode_image(image_file):
    return base64.b64encode(image_file.read()).decode('utf-8')

def process_receipt(image_file):
    # Encode the image
    base64_image = encode_image(image_file)

    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {api_key}"
    }

    var_for = """Given the receipt image provided, extract all relevant informat

    {
      "receipt_headers": {
        "store_name": "",
        "slogan": "",
        "address": "",
        "store_manager": "",
        "phone_number": "",
        "transaction_id": "",
        "date": "",
        "time": "",
        "cashier": "",
        "subtotal": 0,
        "sales_tax": 0,
        "total": 0,
        "gift_card": 0,
```

```
    "charged_amount": 0,
    "card_type": "",
    "auth_code": "",
    "chip_read": "",
    "aid": "",
    "issuer": "",
    "policy_id": "",
    "expiration_date": "",
    "survey_message": "",
    "survey_website": "",
    "user_id": "",
    "password": "",
    "eligibility_note": ""
  },
  "line_items": [
    {
      "sku": "",
      "description": "",
      "details": "",
      "price": 0
    }
  ]
}"""

payload = {
    "model": "gpt-4-vision-preview",
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": var_for
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{base64_image}"
                    }
                }
            ]
        }
    ],
    "max_tokens": 2048
}

response = requests.post("https://api.openai.com/v1/chat/completions", heade

response_json = response.json()
receipt_data_str = response_json['choices'][0]['message']['content']
```

```
# Find the JSON string within the extracted content
receipt_data_json_str = receipt_data_str.split('```json')[1].split('```')[0]

# Convert the JSON string to a Python dictionary
receipt_data = json.loads(receipt_data_json_str)

# Connect to the database
conn = mysql.connector.connect(**db_config)
cursor = conn.cursor()

# Insert into receipt_headers
header_insert_query = """
INSERT INTO receipt_headers (store_name, slogan, address, store_manager, pho
VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
"""

header_info = receipt_data['receipt_headers']
line_items = receipt_data['line_items']

# Convert date to 'YYYY-MM-DD' format
date_str = header_info['date']
date_obj = datetime.strptime(date_str, '%m/%d/%y')
formatted_date = date_obj.strftime('%Y-%m-%d')

# Convert time to 24-hour format
time_str = header_info['time']
time_obj = datetime.strptime(time_str, '%I:%M %p')
formatted_time = time_obj.strftime('%H:%M:%S')

# Convert expiration_date to 'YYYY-MM-DD' format
expiration_date_str = header_info['expiration_date']
expiration_date_obj = datetime.strptime(expiration_date_str, '%m/%d/%Y')
formatted_expiration_date = expiration_date_obj.strftime('%Y-%m-%d')

# Prepare header values
header_values = (
    header_info['store_name'],
    header_info['slogan'],
    header_info['address'],
    header_info['store_manager'],
    header_info['phone_number'],
    header_info['transaction_id'],
    formatted_date,
    formatted_time,
    header_info['cashier'],
    header_info['subtotal'],
    header_info['sales_tax'],
    header_info['total'],
    header_info['gift_card'],
```

```
            header_info['charged_amount'],
            header_info['card_type'],
            header_info['auth_code'],
            header_info['chip_read'],
            header_info['aid'],
            header_info['issuer'],
            header_info['policy_id'],
            formatted_expiration_date,
            header_info['survey_message'],
            header_info['survey_website'],
            header_info['user_id'],
            header_info['password'],
            header_info['eligibility_note']
        )

        # Insert header values
        cursor.execute(header_insert_query, header_values)
        receipt_id = cursor.lastrowid  # Get the last inserted id to use for line_it

        # Prepare and insert line items
        line_item_insert_query = """
        INSERT INTO line_items (receipt_id, sku, description, details, price)
        VALUES (%s, %s, %s, %s, %s)
        """

        for item in line_items:
            price = float(item['price'])
            line_item_values = (
                receipt_id,
                item['sku'],
                item['description'],
                item.get('details', ''),
                price
            )

            cursor.execute(line_item_insert_query, line_item_values)

        # Commit and close the connection
        conn.commit()
        cursor.close()
        conn.close()

        return "Receipt processed successfully!"

    # Create the Gradio interface
    iface = gr.Interface(
        fn=process_receipt,
        inputs=gr.inputs.File(label="Upload Receipt Image"),
        outputs=gr.outputs.Textbox(label="Result"),
        title="Receipt Processing App",
```

```
    description="Upload a receipt image to extract and store the information in
)

# Launch the Gradio app
if __name__ == "__main__":
    iface.launch()
```

In this Gradio app:

1. We import the necessary modules, including gradio, base64, requests, mysql.connector, json, and datetime.
2. We define the OpenAI API key and the database connection parameters. We define the encode_image function to encode the uploaded image file.
3. We define the process_receipt function, which takes the uploaded image file as input:
4. We encode the image using the encode_image function.
5. We set the headers and payload for the API request.
6. We send a POST request to the OpenAI API to extract receipt information.
7. We extract the JSON string from the API response and convert it to a Python dictionary.
8. We connect to the MySQL database using the provided configuration.
9. We insert the receipt headers and line items into the respective tables in the database.
10. We commit the changes and close the database connection.
11. We return a success message.
12. We create the Gradio interface using gr.Interface:

13. We specify the process_receipt function as the main function (fn) to be called when the user uploads an image.
14. We define the input as a file upload using gr.inputs.File and provide a label for it.
15. We define the output as a text box using gr.outputs.Textbox to display the result.
16. We set the title and description of the app.
17. Finally, we launch the Gradio app using iface.launch() if the script is run directly.
    Note: Make sure to have the necessary dependencies installed, such as gradio, requests, and mysql-connector-python.

To use this Gradio app, run the script and access the provided URL in your web browser. Upload a receipt image using the file upload input, and the app will process the image using the OpenAI API, extract the receipt information, store it in the MySQL database, and display a success message in the output text box.

Remember to replace the placeholders for the OpenAI API key and database connection parameters with your actual values.

Gradio Cheatsheet:

```
import gradio as gr

# Basics

# Hello World
def hello(name):
```

```
    return f"Hello, {name}!"

iface = gr.Interface(fn=hello, inputs="text", outputs="text")
iface.launch()

# Image Classification
def classify_image(image):
    # Perform image classification logic here
    return "Classification Result"

iface = gr.Interface(fn=classify_image, inputs="image", outputs="text")
iface.launch()

# Multiple Inputs and Outputs
def process_data(name, age):
    output1 = f"Hello, {name}!"
    output2 = f"You are {age} years old."
    return output1, output2

iface = gr.Interface(fn=process_data, inputs=["text", "number"], outputs=["text"
iface.launch()

# Checkboxes and Dropdown
def process_options(option1, option2):
    return f"Selected options: {option1}, {option2}"

options = ["Option 1", "Option 2", "Option 3"]
iface = gr.Interface(
    fn=process_options,
    inputs=[
        gr.inputs.CheckboxGroup(options, label="Checkbox Options"),
        gr.inputs.Dropdown(options, label="Dropdown Options")
    ],
    outputs="text"
)
iface.launch()

# Radio Buttons and Slider
def process_inputs(radio_value, slider_value):
    return f"Radio value: {radio_value}, Slider value: {slider_value}"

iface = gr.Interface(
    fn=process_inputs,
    inputs=[
        gr.inputs.Radio(["Option 1", "Option 2"], label="Radio Buttons"),
        gr.inputs.Slider(0, 100, label="Slider")
    ],
    outputs="text"
)
iface.launch()
```

```python
# File Upload
def process_file(file):
    # Process the uploaded file here
    return "File processed successfully!"

iface = gr.Interface(fn=process_file, inputs="file", outputs="text")
iface.launch()

# Audio Recording
def process_audio(audio):
    # Process the recorded audio here
    return "Audio processed successfully!"

iface = gr.Interface(fn=process_audio, inputs="microphone", outputs="text")
iface.launch()

# Video Recording
def process_video(video):
    # Process the recorded video here
    return "Video processed successfully!"

iface = gr.Interface(fn=process_video, inputs="webcam", outputs="text")
iface.launch()

# Styling and Customization
iface = gr.Interface(
    fn=hello,
    inputs="text",
    outputs="text",
    title="My Gradio App",
    description="This is a sample Gradio app.",
    theme="default",
    layout="vertical",
    css=".output-text { color: blue; }"
)
iface.launch()

# Event Handling
def update_output(input_value):
    return f"You entered: {input_value}"

input_component = gr.inputs.Textbox(label="Enter text")
output_component = gr.outputs.Textbox(label="Output")

input_component.change(fn=update_output, inputs=input_component, outputs=output_

iface = gr.Interface(inputs=input_component, outputs=output_component)
iface.launch()
```

```python
# State Management
def accumulate_number(history, current_number):
    history = history or []
    history.append(current_number)
    return history, sum(history)

iface = gr.Interface(
    fn=accumulate_number,
    inputs=["state", "number"],
    outputs=["state", "number"],
    allow_screenshot=False,
    allow_flagging=False
)
iface.launch()

# Async Interfaces
import time

def long_running_task(duration):
    time.sleep(duration)
    return f"Task completed after {duration} seconds."

iface = gr.Interface(
    fn=long_running_task,
    inputs=gr.inputs.Slider(minimum=1, maximum=10, step=1, label="Task Duration
    outputs="text",
    title="Async Interface Example",
    description="Demonstrates a long-running task using an async interface."
)
iface.queue().launch()

# Tips and Tricks

# Use meaningful function and component names
# Leverage Gradio's built-in input and output components for various data types
# Customize the app's appearance with CSS and layout options
# Handle events and interactivity using the change() method
# Manage state for stateful interfaces using the "state" input type
# Use async interfaces for long-running tasks to avoid blocking the UI
# Provide clear instructions and examples for your app's usage
# Test your app thoroughly with different inputs and edge cases

# Do's and Don'ts

# Do: Keep your interface simple and intuitive for users
# Don't: Overcomplicate the UI with unnecessary components or unclear instructio

# Do: Use appropriate input and output types for your data
# Don't: Misuse input and output types or expect users to enter data in the wron
```

```
# Do: Provide helpful error messages and validation for user inputs
# Don't: Assume users will always provide valid or expected inputs

# Do: Use event handling and interactivity to enhance the user experience
# Don't: Make the interface unresponsive or confusing to interact with

# Do: Manage state properly for stateful interfaces
# Don't: Ignore state management or create inconsistencies in the app's behavior

# Do: Optimize the app's performance, especially for long-running tasks
# Don't: Block the UI or make users wait unnecessarily for tasks to complete

# Do: Style and customize your app to make it visually appealing and consistent
# Don't: Neglect the app's appearance or use inconsistent styling

# Do: Provide clear documentation and examples for your app's usage
# Don't: Assume users will understand how to use your app without guidance

# Do: Regularly update and maintain your app to fix bugs and add new features
# Don't: Neglect your app after deployment or ignore user feedback and issues
```

References:

Python Web Frameworks:

1. Streamlit:

— Documentation: https://docs.streamlit.io/

— GitHub Repository: https://github.com/streamlit/streamlit

2. Flask:

— Documentation: https://flask.palletsprojects.com/

— GitHub Repository: https://github.com/pallets/flask

3. FastAPI:

— Documentation: https://fastapi.tiangolo.com/

— GitHub Repository: https://github.com/tiangolo/fastapi

4. Django:

— Documentation: https://docs.djangoproject.com/

— GitHub Repository: https://github.com/django/django

Python Libraries for OCR:

1. Tesseract OCR (pytesseract):

— GitHub Repository: https://github.com/madmaze/pytesseract

2. OpenCV (cv2):

— Documentation: https://docs.opencv.org/

— GitHub Repository: https://github.com/opencv/opencv-python

3. EasyOCR:

— GitHub Repository: https://github.com/JaidedAI/EasyOCR

4. Google Cloud Vision API:

— Documentation: https://cloud.google.com/vision/docs

5. Amazon Textract:

— Documentation: https://docs.aws.amazon.com/textract/latest/dg/

Python Libraries for Rule-Based Text Extraction:

1. Regular Expressions (re):

— Documentation: https://docs.python.org/3/library/re.html

2. spaCy:

— Documentation: https://spacy.io/usage

— GitHub Repository: https://github.com/explosion/spaCy

3. NLTK (Natural Language Toolkit):
— Documentation: https://www.nltk.org/
— GitHub Repository: https://github.com/nltk/nltk

4. TextBlob:
— Documentation: https://textblob.readthedocs.io/
— GitHub Repository: https://github.com/sloria/TextBlob

5. fuzzywuzzy:
— GitHub Repository: https://github.com/seatgeek/fuzzywuzzy

Python Libraries for Traditional Machine Learning Approaches:
1. scikit-learn (sklearn):
— Documentation: https://scikit-learn.org/stable/documentation.html
— GitHub Repository: https://github.com/scikit-learn/scikit-learn

2. NLTK (Natural Language Toolkit):
— Documentation: https://www.nltk.org/
— GitHub Repository: https://github.com/nltk/nltk

3. spaCy:
— Documentation: https://spacy.io/usage
— GitHub Repository: https://github.com/explosion/spaCy

4. Pandas:
— Documentation: https://pandas.pydata.org/docs/
— GitHub Repository: https://github.com/pandas-dev/pandas

5. NumPy:
— Documentation: https://numpy.org/doc/

— GitHub Repository: https://github.com/numpy/numpy

Conclusion:

The receipt text extraction app showcased in this article has the potential to eliminate data entry processes across various industries, saving time, reducing errors, and unlocking valuable insights from receipts. By automating the extraction of key fields such as merchant names, transaction dates, item details, and total amounts, businesses can streamline their expense tracking, financial management, and analytics workflows.As we look to the future, the possibilities for enhancing and expanding the capabilities of receipt text extraction apps are endless. With advancements in machine learning techniques, integration with other systems, and support for multiple languages and receipt formats, these apps have the potential to become indispensable tools for organizations of all sizes.

Artificial Intelligence | Machine Learning | Data Science | Python | Programming

Written by Senthil E

2.7K Followers · Writer for Level Up Coding

Following

ML/DS - Certified GCP Professional Machine Learning Engineer, Certified AWS Professional Machine learning Speciality,Certified GCP Professional Data Engineer .

More from Senthil E and Level Up Coding

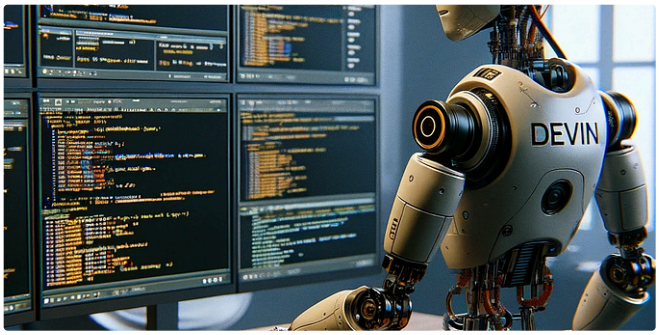
Senthil E in Level Up Coding

Unleashing the Potential of LLMs: How Enterprises are Leveraging ...

From Chatbots to Automation: Exploring the Versatile Use Cases of LLMs in Enterprises

58 min read · Mar 31, 2024

113



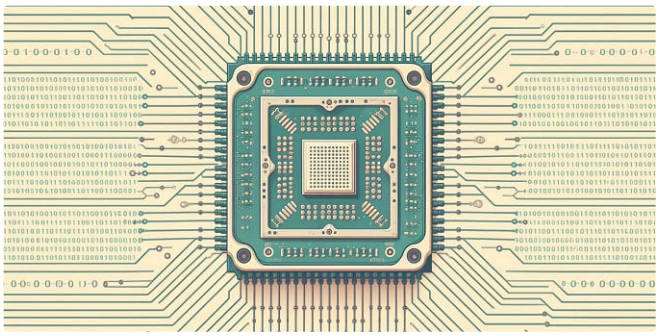
Gencay I. in Level Up Coding

Meet with DevinAI: Is This the Actual End of Coding?

Here's Why It Could Be the Final Curtain for Coding

· 6 min read · Mar 17, 2024

1K 18



Dr. Ashish Bamania in Level Up Coding

Ditch JSON! Here Are 5 (Better) Data Serialization Formats To Use...

Have you heard about "Cap'n Proto", the Infinity times faster protocol?

Senthil E in Analytics Vidhya

MLOps-Dockers and Kubernetes Essentials for a Data Scientist

Introduction

· 5 min read · Apr 1, 2024

934 23

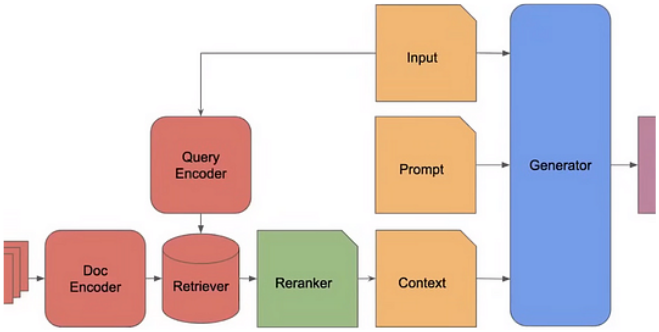
15 min read · Oct 4, 2021

335 6

See all from Senthil E

See all from Level Up Coding

Recommended from Medium



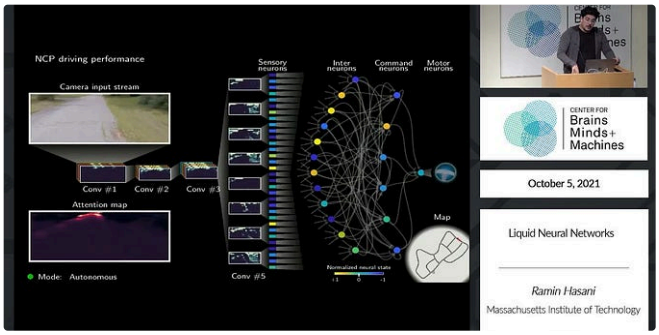
Vishal Rajput in AIGuys

RAG 2.0: Retrieval Augmented Language Models

RAG 2.0 shows the true capabilities of Retrieval Systems and LLMs

· 12 min read · 4 days ago

336



Tim Cvetko in AI Advances

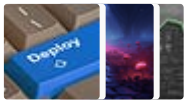
Build Your Own Liquid Neural Network with PyTorch

Why LNNs are so Fascinating—2024 Overview

· 6 min read · Apr 9, 2024

603 6

Lists



Predictive Modeling w/ Python

20 stories · 1113 saves

Practical Guides to Machine Learning

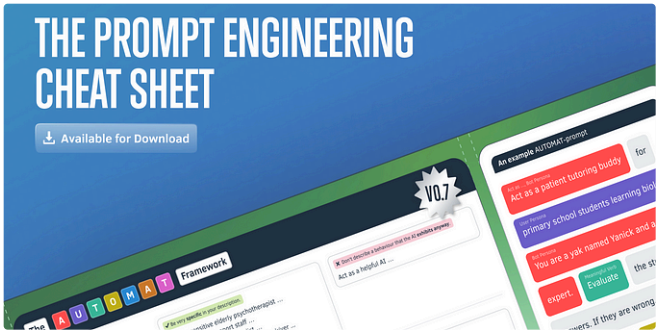
10 stories · 1330 saves

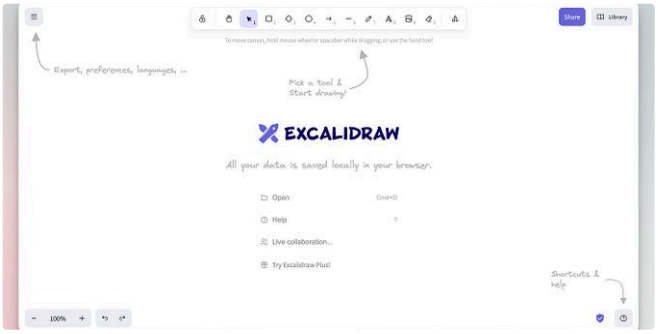
Coding & Development

11 stories · 569 saves

Natural Language Processing

1383 stories · 875 saves





Maximilian Vogel in The Generator

The Perfect Prompt: A Prompt Engineering Cheat Sheet

Large language models can produce any sequence of characters. Literally any. In any...

7 min read · Apr 8, 2024

1.5K 9

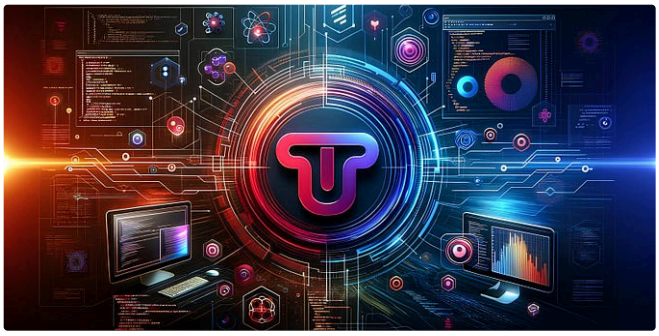
Xinran Ma in Bootcamp

How to create hand-drawn like diagrams with AI

Speed up the process to under a minute

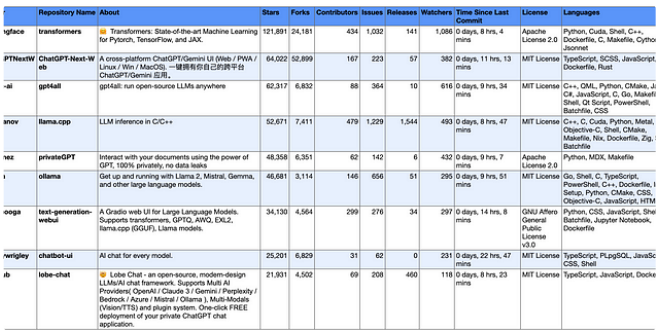
4 min read · Apr 10, 2024

460 3



Thomas Reid in Level Up Coding

Taipy: The best Python GUI tool you've never heard of?



	Repository Name	About	Stars	Forks	Contributors	Issues	Releases	Watchers	Time Since Last Commit	License	Languages
gface	transformers	Transformers: State-of-the-art Machine Learning for Pytorch, TensorFlow, and JAX.	121,891	24,181	434	1,032	141	1,086	0 days, 8 hrs, 4 mins	Apache License 2.0	Python, Cuda, Shell, C++, Dockerfile, C, Makefile, Cytho Jsonnet
PTNextW	ChatGPT-Next-W eb	A cross-platform ChatGPT/Gemini UI (Web / PWA / Linux / Win / MacOS). 一键拥有你自己的跨平台 ChatGPT/Gemini 应用。	64,022	52,899	167	223	57	382	0 days, 11 hrs, 13 mins	MIT License	TypeScript, SCSS, JavaScript, Dockerfile, Rust
-ai	gpt4all	gpt4all: run open-source LLMs anywhere	62,317	6,832	88	364	10	616	0 days, 9 hrs, 34 mins	MIT License	C++, QML, Python, CMake, J C#, JavaScript, C, Go, Makefi Shell, Qt Script, PowerShell, Batchfile, CSS
anov	llama.cpp	LLM inference in C/C++	52,671	7,411	479	1,229	1,544	493	0 days, 8 hrs, 47 mins	MIT License	C++, C, Cuda, Python, Metal, Objective-C, Shell, CMake, Makefile, Nix, Dockerfile, Zig, Batchfile
nez	privateGPT	Interact with your documents using the power of GPT, 100% privately, no data leaks	48,358	6,351	62	142	6	432	0 days, 9 hrs, 7 mins	Apache License 2.0	Python, MDX, Makefile
	ollama	Get up and running with Llama 2, Mistral, Gemma, and other large language models.	46,681	3,114	146	656	51	295	0 days, 9 hrs, 51 mins	MIT License	Go, Shell, C, TypeScript, PowerShell, C++, Dockerfile, I Setup, Python, CMake, CSS, Objective-C, JavaScript, HTM
ooga	text-generation-webui	A Gradio web UI for Large Language Models. Supports transformers, GPTQ, AWQ, EXL2, llama.cpp (GGUF), Llama models.	34,130	4,564	299	276	34	297	0 days, 14 hrs, 8 mins	GNU Affero General Public License v3.0	Python, CSS, JavaScript, She Batchfile, Jupyter Notebook, Dockerfile
wrigley	chatbot-ui	AI chat for every model.	25,201	6,829	31	62	0	231	0 days, 22 hrs, 47 mins	MIT License	TypeScript, PLpgSQL, JavaSc CSS, Shell
ib	lobe-chat	Lobe Chat - an open-source, modern-design LLMs/AI chat framework. Supports Multi AI Providers(OpenAI / Claude 3 / Gemini / Perplexity / Bedrock / Azure / Mistral / Ollama), Multi-Modals (Vision/TTS) and plugin system. One-click FREE deployment of your private ChatGPT chat application.	21,931	4,502	69	208	460	118	0 days, 8 hrs, 23 mins	MIT License	TypeScript, JavaScript, Docke

Vince Lam in The Deep Hub

50+ Open-Source Options for Running LLMs Locally

Could it be a streamlit killer?

✨ · 6 min read · Apr 11, 2024

464 1

In my previous post I discussed the benefits of using locally hosted open weights LLMs,...

8 min read · Mar 12, 2024

228 5

See more recommendations