# app.py (Streamlit Frontend) - COMPLETE UPDATED VERSION

import streamlit as st

import pandas as pd

import requests

import io

import time

import base64

import os

from datetime import datetime

import json

import tempfile

import shutil

# FastAPI backend URL

API\_BASE\_URL = "http://localhost:8000"

# Helper functions for deep config

def validate\_and\_normalize\_headers(columns):

"""Validate and normalize column headers"""

new\_columns = []

for i, col in enumerate(columns):

if col is None or str(col).strip() == "":

new\_col = f"column\_{i+1}"

else:

new\_col = str(col).strip().lower()

new\_columns.append(new\_col)

return new\_columns

# ---------- Minimalist Dark Theme ----------

st.markdown("""

<style>

:root {

--ev-colors-primary: #282828;

--ev-colors-secondary: #424242;

--ev-colors-tertiary: #4e332a;

--ev-colors-highlight: #e75f33;

--ev-colors-text: #fff;

--ev-colors-secondaryText: grey;

--ev-colors-tertiaryText: #a3a3a3;

--ev-colors-borderColor: #ffffff1f;

--ev-colors-background: #161616;

--ev-colors-success: #d8fc77;

--ev-colors-danger: #dc143c;

}

/\* Main background \*/

.stApp {

background: var(--ev-colors-background);

color: var(--ev-colors-text);

}

/\* Headers \*/

h1, h2, h3, h4, h5, h6 {

color: var(--ev-colors-text) !important;

border-left: 4px solid var(--ev-colors-secondary) !important;

padding-left: 10px !important;

}

/\* Cards \*/

.custom-card {

background: var(--ev-colors-primary);

border: 1px solid var(--ev-colors-borderColor);

border-radius: 8px;

padding: 20px;

margin: 10px 0;

transition: all 0.3s ease;

}

.custom-card:hover {

background: var(--ev-colors-secondary);

}

.card-title {

color: var(--ev-colors-text);

font-size: 1.2em;

font-weight: 600;

margin-bottom: 15px;

}

.card-content {

color: var(--ev-colors-tertiaryText);

font-size: 0.95em;

line-height: 1.5;

}

/\* Buttons - Only primary buttons use highlight color \*/

.stButton > button {

background: var(--ev-colors-secondary) !important;

color: var(--ev-colors-text) !important;

border: 1px solid var(--ev-colors-borderColor) !important;

border-radius: 6px !important;

padding: 8px 16px !important;

font-weight: 500 !important;

transition: all 0.2s ease !important;

}

.stButton > button:hover {

background: var(--ev-colors-tertiary) !important;

border-color: var(--ev-colors-tertiaryText) !important;

}

/\* Primary/Important buttons use highlight color \*/

.primary-button > button {

background: var(--ev-colors-highlight) !important;

color: white !important;

border: none !important;

font-weight: 600 !important;

}

.primary-button > button:hover {

background: #f27024 !important;

transform: translateY(-1px) !important;

}

/\* Process steps \*/

.process-step {

background: var(--ev-colors-primary);

padding: 15px;

border-radius: 6px;

margin: 8px 0;

border-left: 4px solid var(--ev-colors-secondary);

transition: all 0.3s ease;

}

.process-step.running {

border-left-color: var(--ev-colors-highlight);

}

.process-step.completed {

border-left-color: var(--ev-colors-success);

}

.process-step.pending {

border-left-color: var(--ev-colors-secondary);

}

/\* Dataframes \*/

.dataframe {

background: var(--ev-colors-primary) !important;

color: var(--ev-colors-text) !important;

}

/\* Input fields \*/

.stTextInput > div > div > input {

background: var(--ev-colors-primary);

color: var(--ev-colors-text);

border: 1px solid var(--ev-colors-borderColor);

}

.stTextInput > div > div > input:focus {

border-color: var(--ev-colors-highlight);

}

/\* Select boxes \*/

.stSelectbox > div > div {

background: var(--ev-colors-primary);

color: var(--ev-colors-text);

border: 1px solid var(--ev-colors-borderColor);

}

/\* Number inputs \*/

.stNumberInput > div > div > input {

background: var(--ev-colors-primary);

color: var(--ev-colors-text);

border: 1px solid var(--ev-colors-borderColor);

}

/\* Checkboxes & Radio buttons \*/

.stCheckbox > label, .stRadio > label {

color: var(--ev-colors-text) !important;

}

/\* Sidebar \*/

.css-1d391kg {

background: var(--ev-colors-primary) !important;

}

/\* Messages \*/

.stSuccess {

background: var(--ev-colors-primary) !important;

color: var(--ev-colors-success) !important;

border-left: 4px solid var(--ev-colors-success) !important;

}

.stError {

background: var(--ev-colors-primary) !important;

color: var(--ev-colors-danger) !important;

border-left: 4px solid var(--ev-colors-danger) !important;

}

.stWarning {

background: var(--ev-colors-primary) !important;

color: var(--ev-colors-highlight) !important;

border-left: 4px solid var(--ev-colors-highlight) !important;

}

.stInfo {

background: var(--ev-colors-primary) !important;

color: var(--ev-colors-text) !important;

border-left: 4px solid var(--ev-colors-secondary) !important;

}

/\* Text areas \*/

.stTextArea > div > div > textarea {

background: var(--ev-colors-primary);

color: var(--ev-colors-text);

border: 1px solid var(--ev-colors-borderColor);

}

/\* Preview table \*/

.preview-table {

background: var(--ev-colors-primary);

border: 1px solid var(--ev-colors-borderColor);

border-radius: 6px;

padding: 15px;

margin: 10px 0;

}

/\* File upload \*/

.uploadedFile {

background: var(--ev-colors-primary);

border: 2px dashed var(--ev-colors-borderColor);

border-radius: 8px;

padding: 20px;

text-align: center;

margin: 10px 0;

}

/\* Progress bar \*/

.stProgress > div > div > div {

background-color: var(--ev-colors-highlight);

}

/\* Scrollable chunk display \*/

.scrollable-chunk {

background: var(--ev-colors-primary);

border: 1px solid var(--ev-colors-borderColor);

border-radius: 4px;

padding: 10px;

margin: 5px 0;

max-height: 300px;

overflow-y: auto;

font-family: monospace;

font-size: 0.85em;

line-height: 1.4;

white-space: pre-wrap;

word-wrap: break-word;

}

.chunk-header {

background: var(--ev-colors-secondary);

padding: 8px 12px;

border-radius: 4px;

margin-bottom: 8px;

font-weight: bold;

color: var(--ev-colors-text);

}

/\* Scrollbar \*/

.scrollable-chunk::-webkit-scrollbar {

width: 6px;

}

.scrollable-chunk::-webkit-scrollbar-track {

background: var(--ev-colors-primary);

}

.scrollable-chunk::-webkit-scrollbar-thumb {

background: var(--ev-colors-secondary);

border-radius: 3px;

}

.scrollable-chunk::-webkit-scrollbar-thumb:hover {

background: var(--ev-colors-tertiaryText);

}

/\* Minimal highlight usage \*/

.highlight-text {

color: var(--ev-colors-highlight);

font-weight: 600;

}

/\* Section headers \*/

.section-header {

color: var(--ev-colors-text);

border-bottom: 1px solid var(--ev-colors-borderColor);

padding-bottom: 10px;

margin-bottom: 20px;

}

/\* Remove default Streamlit colors \*/

.st-bb {

border-bottom-color: var(--ev-colors-borderColor) !important;

}

.st-at {

background-color: var(--ev-colors-primary) !important;

}

.st-ae {

background-color: var(--ev-colors-secondary) !important;

}

.st-af {

background-color: var(--ev-colors-primary) !important;

}

.st-ag {

color: var(--ev-colors-text) !important;

}

.st-ah {

color: var(--ev-colors-tertiaryText) !important;

}

.st-ai {

color: var(--ev-colors-text) !important;

}

/\* Toggle and slider colors \*/

.st-cb {

background-color: var(--ev-colors-secondary) !important;

}

.st-cc {

background-color: var(--ev-colors-highlight) !important;

}

/\* Selectbox dropdown \*/

.st-cd {

background-color: var(--ev-colors-primary) !important;

border: 1px solid var(--ev-colors-borderColor) !important;

}

.st-ce {

color: var(--ev-colors-text) !important;

}

.st-cf:hover {

background-color: var(--ev-colors-secondary) !important;

}

/\* Expander \*/

.streamlit-expanderHeader {

background-color: var(--ev-colors-primary) !important;

color: var(--ev-colors-text) !important;

border: 1px solid var(--ev-colors-borderColor) !important;

}

/\* Radio buttons \*/

.st-bh {

background-color: var(--ev-colors-primary) !important;

}

.st-bi {

background-color: var(--ev-colors-highlight) !important;

}

/\* Checkbox \*/

.st-bg {

background-color: var(--ev-colors-primary) !important;

border: 1px solid var(--ev-colors-borderColor) !important;

}

</style>

""", unsafe\_allow\_html=True)

# ---------- SVG Logo Integration ----------

logo\_svg = """<svg id="Layer\_2" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 1703.31 535.6"><defs><style>

.cls-1 {

fill: #fff;

}

.cls-2 {

fill: #fbb03b;

}

.cls-3 {

fill: #f27024;

}

</style></defs><g id="Layer\_10"><g><path class="cls-1" d="M125.67,428.34c-39.15,0-70.27-13.09-92.48-38.91C11.17,363.84,0,334.47,0,302.15c0-30.4,9.47-57.88,28.14-81.68,23.77-30.39,56.01-45.8,95.83-45.8s74.1,15.76,98.58,46.85c17.39,21.95,26.36,49.63,26.66,82.28l.05,5.23H41.22c1.5,23.04,9.58,42.3,24.08,57.31,15.74,16.28,34.65,24.2,57.81,24.2,11.12,0,22.08-1.96,32.6-5.83,10.49-3.85,19.51-9.02,26.82-15.36,7.36-6.39,8.83-7.95,14.56-15.39l2.6-4.32c5.42-9.02,16.94-12.25,26.26-7.35h0c9.62,5.06,13.39,16.91,8.46,26.6l-1.53,3c-8.02,11.54-10.34,14.39-21.53,24.68-11.22,10.32-24.02,18.29-38.05,23.68-14.02,5.38-30.04,8.1-47.63,8.1ZM204.47,272.93c-3.65-12.13-8.55-22.08-14.6-29.64-7.06-8.82-16.57-16.06-28.27-21.51-11.75-5.46-24.27-8.23-37.2-8.23-21.29,0-39.83,6.92-55.1,20.58-9.88,8.81-17.76,21.84-23.46,38.8h158.64Z"></path><rect class="cls-1" x="288.28" y="97.26" width="40.15" height="331.08" rx="20.07" ry="20.07"></rect><path class="cls-1" d="M490.58,428.34c-39.15,0-70.27-13.09-92.48-38.91-22.02-25.59-33.19-54.96-33.19-87.28,0-30.4,9.47-57.88,28.14-81.68,23.77-30.39,56.01-45.8,95.83-45.8s74.1,15.76,98.58,46.85c17.39,21.95,26.36,49.63,26.66,82.28l.05,5.23h-208.03c1.5,23.04,9.58,42.3,24.08,57.31,15.74,16.28,34.65,24.2,57.81,24.2,11.12,0,22.08-1.96,32.6-5.83,10.49-3.85,19.51-9.02,26.82-15.36,7.36-6.39,8.83-7.95,14.56-15.39l2.6-4.32c5.42-9.02,16.94-12.25,26.26-7.35h0c9.62,5.06,13.39,16.91,8.46,26.6l-1.53,3c-8.02,11.54-10.34,14.39-21.53,24.68-11.22,10.32-24.02,18.29-38.05,23.68-14.02,5.38-30.04,8.1-47.63,8.1ZM569.37,272.93c-3.65-12.13-8.55-22.08-14.6-29.64-7.06-8.82-16.57-16.06-28.27-21.51-11.75-5.46-24.27-8.23-37.2-8.23-21.29,0-39.83,6.92-55.1,20.58-9.88,8.81-17.76,21.84-23.46,38.8h158.64Z"></path><path class="cls-1" d="M751.92,422.82l-96-208.47c-5.97-12.97,3.5-27.77,17.78-27.77h0c7.64,0,14.59,4.45,17.78,11.39l69.08,150.01,68.21-149.93c3.18-6.99,10.15-11.47,17.82-11.47h.22c14.26,0,23.74,14.76,17.8,27.73l-95.43,208.49c-1.55,3.38-4.92,5.54-8.63,5.54h0c-3.71,0-7.08-2.16-8.63-5.52Z"></path><g><path class="cls-2" d="M1052.79,311.55c-30.67,0-56.25,33.01-62.14,66.95,5.07-11.19,11.63-17.94,18.79-17.94,15.94,0,23.38,33.67,28.84,74.37,1.51,11.28,12.67,86.53,13.56,100.67.05,0,.11,0,.16,0,1.04-16.27,10.83-87.61,12.64-100.66,5.78-41.56,12.93-74.37,28.87-74.37,9.09,0,17.21,10.84,22.5,27.76-2.22-38.69-29.66-76.77-63.22-76.77Z"></path><path class="cls-3" d="M1053.33,46.78c60,50.38,96.73,131.67,97.74,218.86-26.55-32.52-60.86-50.27-97.76-50.27s-71.19,17.74-97.74,50.24c1.01-87.19,37.75-168.47,97.75-218.83M1053.33,0c-80.86,53.76-135.27,154.25-135.27,269.32,0,28.59,3.36,56.29,9.66,82.6,4.47,18.64,10.39,36.6,17.66,53.67,2.54-84.98,49.89-152.72,107.94-152.72s105.41,67.76,107.94,152.76c10.02-23.52,17.51-48.73,22.09-75.13,3.46-19.78,5.25-40.25,5.25-61.19C1188.59,154.25,1134.19,53.78,1053.33,0h0Z"></path></g><path class="cls-3" d="M1246.12,390.85l-15.96-370.06C1229.55,9.49,1238.55,0,1249.87,0h0c11.31,0,20.31,9.49,19.71,20.79l-15.96,370.06h-7.5Z"></path><path class="cls-1" d="M1333.96,408.27v-185.58h-40.62v-36.1h40.62v-69.25c0-11.09,8.99-20.07,20.07-20.07h0c11.09,0,20.07,8.99,20.07,20.07v69.25h62.21v36.1h-62.21v185.58c0,11.09-8.99,20.07-20.07,20.07h0c-11.09,0-20.07-8.99-20.07-20.07Z"></path><path class="cls-1" d="M1579.72,428.34c-39.15,0-70.26-13.09-92.48-38.91-22.02-25.59-33.18-54.95-33.18-87.28,0-30.4,9.47-57.88,28.14-81.68,23.77-30.39,56.01-45.8,95.83-45.8s74.1,15.76,98.59,46.85c17.39,21.94,26.36,49.63,26.66,82.28l.05,5.23h-208.03c1.5,23.04,9.59,42.3,24.08,57.31,15.74,16.28,34.64,24.2,57.81,24.2,11.12,0,22.09-1.96,32.6-5.83,10.49-3.85,19.51-9.02,26.82-15.36,7.36-6.39,9.22-7.53,15.54-17.02l1.62-2.69c5.42-9.02,16.94-12.25,26.26-7.35h0c9.62,5.06,13.39,16.91,8.46,26.6l-1.36,2.67c-6.09,8.44-10.51,14.72-21.7,25.01-11.22,10.32-24.02,18.29-38.06,23.68-14.02,5.38-30.04,8.1-47.63,8.1ZM1658.52,272.93c-3.65-12.13-8.55-22.08-14.6-29.64-7.06-8.82-16.57-16.06-28.27-21.51-11.76-5.46-24.27-8.23-37.2-8.23-21.29,0-39.83,6.92-55.1,20.58-9.89,8.81-17.76,21.85-23.46,38.8h158.64Z"></path></g></g></svg>"""

# Convert SVG to base64 and display

b64\_logo = base64.b64encode(logo\_svg.encode('utf-8')).decode("utf-8")

# Display logo and header

st.markdown(

f'''

<div style="text-align: center; margin-bottom: 20px;">

<img src="data:image/svg+xml;base64,{b64\_logo}" width="300" alt="I Chunk Optimizer Logo">

</div>

<div style="background: var(--ev-colors-primary); border: 1px solid var(--ev-colors-borderColor); border-radius: 8px; padding: 20px; margin-bottom: 30px;">

<h1 style="color: var(--ev-colors-text); text-align: center; margin: 0; font-size: 2.2em;">I Chunk Optimizer</h1>

<p style="color: var(--ev-colors-tertiaryText); text-align: center; margin: 10px 0 0 0; font-size: 1.1em;">Advanced Text Processing + 3GB File Support + Performance Optimized</p>

</div>

''',

unsafe\_allow\_html=True

)

# ---------- API Client Functions ----------

def call\_fast\_api(file\_path: str, filename: str, db\_type: str, db\_config: dict = None,

use\_openai: bool = False, openai\_api\_key: str = None, openai\_base\_url: str = None,

process\_large\_files: bool = True, use\_turbo: bool = False, batch\_size: int = 256):

"""Send CSV upload or trigger DB import for Fast mode"""

try:

# DB import path: send only form data (no file open)

if db\_config and db\_config.get('use\_db'):

data = {

"db\_type": db\_config.get("db\_type"),

"host": db\_config.get("host"),

"port": db\_config.get("port"),

"username": db\_config.get("username"),

"password": db\_config.get("password"),

"database": db\_config.get("database"),

"table\_name": db\_config.get("table\_name"),

"use\_openai": use\_openai,

"openai\_api\_key": openai\_api\_key,

"openai\_base\_url": openai\_base\_url,

"process\_large\_files": process\_large\_files,

"use\_turbo": use\_turbo,

"batch\_size": batch\_size

}

response = requests.post(f"{API\_BASE\_URL}/run\_fast", data=data)

return response.json()

# CSV upload path: open and send file

with open(file\_path, 'rb') as f:

files = {"file": (filename, f, "text/csv")}

data = {

"db\_type": db\_type,

"use\_openai": use\_openai,

"openai\_api\_key": openai\_api\_key,

"openai\_base\_url": openai\_base\_url,

"process\_large\_files": process\_large\_files,

"use\_turbo": use\_turbo,

"batch\_size": batch\_size

}

response = requests.post(f"{API\_BASE\_URL}/run\_fast", files=files, data=data)

return response.json()

except Exception as e:

return {"error": f"API call failed: {str(e)}"}

# Deep Config Step-by-Step API Functions

def call\_deep\_config\_preprocess\_api(file\_path: str, filename: str, db\_config: dict = None):

"""Step 1: Preprocess data"""

try:

if db\_config and db\_config.get('use\_db'):

data = {

"db\_type": db\_config.get("db\_type"),

"host": db\_config.get("host"),

"port": db\_config.get("port"),

"username": db\_config.get("username"),

"password": db\_config.get("password"),

"database": db\_config.get("database"),

"table\_name": db\_config.get("table\_name")

}

response = requests.post(f"{API\_BASE\_URL}/deep\_config/preprocess", data=data)

else:

with open(file\_path, 'rb') as f:

files = {"file": (filename, f, "text/csv")}

response = requests.post(f"{API\_BASE\_URL}/deep\_config/preprocess", files=files)

return response.json()

except Exception as e:

return {"error": f"Preprocess API call failed: {str(e)}"}

def call\_deep\_config\_type\_convert\_api(type\_conversions: dict):

"""Step 2: Convert data types"""

try:

data = {"type\_conversions": json.dumps(type\_conversions)}

response = requests.post(f"{API\_BASE\_URL}/deep\_config/type\_convert", data=data)

return response.json()

except Exception as e:

return {"error": f"Type convert API call failed: {str(e)}"}

def call\_deep\_config\_null\_handle\_api(null\_strategies: dict):

"""Step 3: Handle null values"""

try:

data = {"null\_strategies": json.dumps(null\_strategies)}

response = requests.post(f"{API\_BASE\_URL}/deep\_config/null\_handle", data=data)

return response.json()

except Exception as e:

return {"error": f"Null handle API call failed: {str(e)}"}

def call\_deep\_config\_stopwords\_api(remove\_stopwords: bool):

"""Step 4: Remove stop words"""

try:

data = {"remove\_stopwords": remove\_stopwords}

response = requests.post(f"{API\_BASE\_URL}/deep\_config/stopwords", data=data)

return response.json()

except Exception as e:

return {"error": f"Stopwords API call failed: {str(e)}"}

def call\_deep\_config\_normalize\_api(text\_processing: str):

"""Step 5: Text normalization"""

try:

data = {"text\_processing": text\_processing}

response = requests.post(f"{API\_BASE\_URL}/deep\_config/normalize", data=data)

return response.json()

except Exception as e:

return {"error": f"Normalize API call failed: {str(e)}"}

def call\_deep\_config\_chunk\_api(chunk\_params: dict):

"""Step 6: Chunk data"""

try:

# Extract parameters from the dictionary

chunk\_method = chunk\_params.get("method", "fixed")

chunk\_size = chunk\_params.get("chunk\_size", 400)

overlap = chunk\_params.get("overlap", 50)

key\_column = chunk\_params.get("key\_column")

token\_limit = chunk\_params.get("token\_limit", 2000)

preserve\_headers = chunk\_params.get("preserve\_headers", True)

data = {

"chunk\_method": chunk\_method,

"chunk\_size": chunk\_size,

"overlap": overlap,

"token\_limit": token\_limit,

"preserve\_headers": preserve\_headers

}

if key\_column:

data["key\_column"] = key\_column

if chunk\_method == "semantic":

data["n\_clusters"] = chunk\_params.get("n\_clusters", 10)

response = requests.post(f"{API\_BASE\_URL}/deep\_config/chunk", data=data)

return response.json()

except Exception as e:

return {"error": f"Chunk API call failed: {str(e)}"}

def call\_deep\_config\_embed\_api(embed\_params: dict):

"""Step 7: Generate embeddings"""

try:

# Extract parameters from the dictionary

model\_name = embed\_params.get("model\_name", "paraphrase-MiniLM-L6-v2")

use\_openai = embed\_params.get("use\_openai", False)

openai\_api\_key = embed\_params.get("openai\_api\_key")

openai\_base\_url = embed\_params.get("openai\_base\_url")

batch\_size = embed\_params.get("batch\_size", 64)

use\_parallel = embed\_params.get("use\_parallel", True)

data = {

"model\_name": model\_name,

"use\_openai": use\_openai,

"batch\_size": batch\_size

}

if openai\_api\_key:

data["openai\_api\_key"] = openai\_api\_key

if openai\_base\_url:

data["openai\_base\_url"] = openai\_base\_url

response = requests.post(f"{API\_BASE\_URL}/deep\_config/embed", data=data)

return response.json()

except Exception as e:

return {"error": f"Embed API call failed: {str(e)}"}

def call\_deep\_config\_store\_api(store\_params: dict):

"""Step 8: Store embeddings"""

try:

# Extract parameters from the dictionary

storage\_type = store\_params.get("storage\_type", "chroma")

collection\_name = store\_params.get("collection\_name", "deep\_config\_collection")

retrieval\_metric = store\_params.get("retrieval\_metric", "cosine")

data = {

"storage\_type": storage\_type,

"collection\_name": collection\_name

}

response = requests.post(f"{API\_BASE\_URL}/deep\_config/store", data=data)

return response.json()

except Exception as e:

return {"error": f"Store API call failed: {str(e)}"}

# Download functions for Deep Config

def download\_deep\_config\_preprocessed():

"""Download preprocessed data"""

response = requests.get(f"{API\_BASE\_URL}/deep\_config/export/preprocessed")

return response.content

def download\_deep\_config\_chunks():

"""Download chunks"""

response = requests.get(f"{API\_BASE\_URL}/deep\_config/export/chunks")

return response.content

def download\_deep\_config\_embeddings():

"""Download embeddings"""

response = requests.get(f"{API\_BASE\_URL}/deep\_config/export/embeddings")

return response.content

def call\_config1\_api(file\_path: str, filename: str, config: dict, db\_config: dict = None,

use\_openai: bool = False, openai\_api\_key: str = None, openai\_base\_url: str = None,

process\_large\_files: bool = True, use\_turbo: bool = False, batch\_size: int = 256):

"""Send CSV upload or trigger DB import for Config-1"""

try:

# DB import path: send only form data

if db\_config and db\_config.get('use\_db'):

data = {k: str(v).lower() if isinstance(v, bool) else v for k, v in config.items()}

data.update({

"db\_type": db\_config.get("db\_type"),

"host": db\_config.get("host"),

"port": db\_config.get("port"),

"username": db\_config.get("username"),

"password": db\_config.get("password"),

"database": db\_config.get("database"),

"table\_name": db\_config.get("table\_name"),

"use\_openai": use\_openai,

"openai\_api\_key": openai\_api\_key,

"openai\_base\_url": openai\_base\_url,

"process\_large\_files": process\_large\_files,

"use\_turbo": use\_turbo,

"batch\_size": batch\_size

})

response = requests.post(f"{API\_BASE\_URL}/run\_config1", data=data)

return response.json()

# CSV upload path: open and send file

with open(file\_path, 'rb') as f:

files = {"file": (filename, f, "text/csv")}

data = {k: str(v).lower() if isinstance(v, bool) else v for k, v in config.items()}

data.update({

"use\_openai": use\_openai,

"openai\_api\_key": openai\_api\_key,

"openai\_base\_url": openai\_base\_url,

"process\_large\_files": process\_large\_files,

"use\_turbo": use\_turbo,

"batch\_size": batch\_size

})

response = requests.post(f"{API\_BASE\_URL}/run\_config1", files=files, data=data)

return response.json()

except Exception as e:

return {"error": f"API call failed: {str(e)}"}

def call\_retrieve\_api(query: str, k: int = 5):

data = {"query": query, "k": k}

response = requests.post(f"{API\_BASE\_URL}/retrieve", data=data)

return response.json()

def call\_openai\_retrieve\_api(query: str, model: str = "all-MiniLM-L6-v2", n\_results: int = 5):

data = {"query": query, "model": model, "n\_results": n\_results}

response = requests.post(f"{API\_BASE\_URL}/v1/retrieve", data=data)

return response.json()

def call\_openai\_embeddings\_api(text: str, model: str = "text-embedding-ada-002",

openai\_api\_key: str = None, openai\_base\_url: str = None):

data = {

"model": model,

"input": text,

"openai\_api\_key": openai\_api\_key,

"openai\_base\_url": openai\_base\_url

}

response = requests.post(f"{API\_BASE\_URL}/v1/embeddings", data=data)

return response.json()

def get\_system\_info\_api():

response = requests.get(f"{API\_BASE\_URL}/system\_info")

return response.json()

def get\_file\_info\_api():

response = requests.get(f"{API\_BASE\_URL}/file\_info")

return response.json()

def get\_capabilities\_api():

response = requests.get(f"{API\_BASE\_URL}/capabilities")

return response.json()

def download\_file(url: str, filename: str):

response = requests.get(f"{API\_BASE\_URL}{url}")

return response.content

def download\_embeddings\_text():

"""Download embeddings in text format"""

response = requests.get(f"{API\_BASE\_URL}/export/embeddings\_text")

return response.content

# Database helper functions

def db\_test\_connection\_api(payload: dict):

return requests.post(f"{API\_BASE\_URL}/db/test\_connection", data=payload).json()

def db\_list\_tables\_api(payload: dict):

return requests.post(f"{API\_BASE\_URL}/db/list\_tables", data=payload).json()

# ---------- Large File Helper Functions ----------

def is\_large\_file(file\_size: int, threshold\_mb: int = 100) -> bool:

"""Check if file is considered large"""

return file\_size > threshold\_mb \* 1024 \* 1024

def format\_file\_size(size\_bytes: int) -> str:

"""Format file size in human readable format"""

for unit in ['B', 'KB', 'MB', 'GB']:

if size\_bytes < 1024.0:

return f"{size\_bytes:.2f} {unit}"

size\_bytes /= 1024.0

return f"{size\_bytes:.2f} TB"

def handle\_file\_upload(uploaded\_file):

"""

Safely handle file uploads by streaming to disk (no memory loading)

Returns temporary file path and file info

"""

# Create temporary file on disk

with tempfile.NamedTemporaryFile(delete=False, suffix='.csv') as tmp\_file:

# Stream the uploaded file directly to disk

shutil.copyfileobj(uploaded\_file, tmp\_file)

temp\_path = tmp\_file.name

# Get file size from disk

file\_size = os.path.getsize(temp\_path)

file\_size\_str = format\_file\_size(file\_size)

file\_info = {

"name": uploaded\_file.name,

"size": file\_size\_str,

"upload\_time": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),

"location": "Temporary storage",

"temp\_path": temp\_path

}

return temp\_path, file\_info

# ---------- Scrollable Chunk Display Function ----------

def display\_scrollable\_chunk(result, chunk\_index):

"""Display chunk content in a scrollable container"""

similarity\_color = "#28a745" if result['similarity'] > 0.7 else "#ffc107" if result['similarity'] > 0.4 else "#dc3545"

# Create a unique key for the expander

expander\_key = f"chunk\_{chunk\_index}\_{result['rank']}"

with st.expander(f"📄 Rank #{result['rank']} (Similarity: {result['similarity']:.3f})", expanded=False):

# Header with similarity score

st.markdown(f"""

<div style="background: #2d2d2d; padding: 10px; border-radius: 5px; margin-bottom: 10px; border-left: 4px solid {similarity\_color};">

<strong>Rank:</strong> {result['rank']} |

<strong>Similarity:</strong> {result['similarity']:.3f} |

<strong>Distance:</strong> {result.get('distance', 'N/A')}

</div>

""", unsafe\_allow\_html=True)

# Scrollable content area

st.markdown("""

<div class="chunk-header">

📋 Chunk Content (Scrollable)

</div>

""", unsafe\_allow\_html=True)

# Use text\_area for scrollable content but make it read-only

content = result['content']

# Create a scrollable text area

st.text\_area(

"Chunk Content",

value=content,

height=300,

key=f"chunk\_content\_{chunk\_index}",

disabled=True,

label\_visibility="collapsed"

)

# Additional metadata if available

if 'metadata' in result:

st.markdown("""

<div class="chunk-header">

ℹ️ Metadata

</div>

""", unsafe\_allow\_html=True)

st.json(result['metadata'])

# ---------- Streamlit App ----------

st.set\_page\_config(page\_title="I Chunk Optimizer", layout="wide", page\_icon="")

# Session state

if "api\_results" not in st.session\_state:

st.session\_state.api\_results = None

if "current\_mode" not in st.session\_state:

st.session\_state.current\_mode = None

if "uploaded\_file" not in st.session\_state:

st.session\_state.uploaded\_file = None

if "retrieval\_results" not in st.session\_state:

st.session\_state.retrieval\_results = None

if "process\_status" not in st.session\_state:

st.session\_state.process\_status = {

"preprocessing": "pending",

"chunking": "pending",

"embedding": "pending",

"storage": "pending",

"retrieval": "pending"

}

if "process\_timings" not in st.session\_state:

st.session\_state.process\_timings = {}

if "file\_info" not in st.session\_state:

st.session\_state.file\_info = {}

if "current\_df" not in st.session\_state:

st.session\_state.current\_df = None

if "column\_types" not in st.session\_state:

st.session\_state.column\_types = {}

if "preview\_df" not in st.session\_state:

st.session\_state.preview\_df = None

if "text\_processing\_option" not in st.session\_state:

st.session\_state.text\_processing\_option = "none"

if "preview\_updated" not in st.session\_state:

st.session\_state.preview\_updated = False

if "use\_openai" not in st.session\_state:

st.session\_state.use\_openai = False

if "openai\_api\_key" not in st.session\_state:

st.session\_state.openai\_api\_key = ""

if "openai\_base\_url" not in st.session\_state:

st.session\_state.openai\_base\_url = ""

if "process\_large\_files" not in st.session\_state:

st.session\_state.process\_large\_files = True

if "temp\_file\_path" not in st.session\_state:

st.session\_state.temp\_file\_path = None

if "use\_turbo" not in st.session\_state:

st.session\_state.use\_turbo = True

if "batch\_size" not in st.session\_state:

st.session\_state.batch\_size = 256

# Sidebar with process tracking and system info

with st.sidebar:

st.markdown("""

<div style="background: linear-gradient(45deg, var(--ev-colors-highlight), #FFA500); padding: 20px; border-radius: 10px; margin-bottom: 20px;">

<h2 style="color: white; text-align: center; margin: 0;">Process Tracker</h2>

</div>

""", unsafe\_allow\_html=True)

# API connection test

try:

response = requests.get(f"{API\_BASE\_URL}/health", timeout=5)

st.success("✅ API Connected")

# Show capabilities

capabilities = get\_capabilities\_api()

if capabilities.get('large\_file\_support'):

st.info("🚀 3GB+ File Support")

if capabilities.get('performance\_features', {}).get('turbo\_mode'):

st.info("⚡ Turbo Mode Available")

except:

st.error("❌ API Not Connected")

st.markdown("---")

# OpenAI Configuration

with st.expander("🤖 OpenAI Configuration"):

st.session\_state.use\_openai = st.checkbox("Use OpenAI API", value=st.session\_state.use\_openai)

if st.session\_state.use\_openai:

st.session\_state.openai\_api\_key = st.text\_input("OpenAI API Key",

value=st.session\_state.openai\_api\_key,

type="password",

help="Your OpenAI API key")

st.session\_state.openai\_base\_url = st.text\_input("OpenAI Base URL (optional)",

value=st.session\_state.openai\_base\_url,

placeholder="https://api.openai.com/v1",

help="Custom OpenAI-compatible API endpoint")

if st.session\_state.openai\_api\_key:

st.success("✅ OpenAI API Configured")

else:

st.warning("⚠️ Please enter OpenAI API Key")

# Large File Configuration

with st.expander("💾 Large File Settings"):

st.session\_state.process\_large\_files = st.checkbox(

"Enable Large File Processing",

value=st.session\_state.process\_large\_files,

help="Process files larger than 100MB in batches to avoid memory issues"

)

if st.session\_state.process\_large\_files:

st.info("""\*\*Large File Features:\*\*

- Direct disk streaming (no memory overload)

- Batch processing for memory efficiency

- Automatic chunking for files >100MB

- Progress tracking for large datasets

- Support for 3GB+ files

""")

# Process steps display

st.markdown("### ⚙️ Processing Steps")

steps = [

("preprocessing", "🧹 Preprocessing"),

("chunking", "📦 Chunking"),

("embedding", "🤖 Embedding"),

("storage", "💾 Vector DB"),

("retrieval", "🔍 Retrieval")

]

for step\_key, step\_name in steps:

status = st.session\_state.process\_status.get(step\_key, "pending")

timing = st.session\_state.process\_timings.get(step\_key, "")

if status == "completed":

icon = "✅"

color = "completed"

timing\_display = f"({timing})" if timing else ""

elif status == "running":

icon = "🟠"

color = "running"

timing\_display = ""

else:

icon = "⚪"

color = "pending"

timing\_display = ""

st.markdown(f"""

<div class="process-step {color}">

{icon} <strong>{step\_name}</strong> {timing\_display}

</div>

""", unsafe\_allow\_html=True)

st.markdown("---")

# System Information

st.markdown("### 💻 System Information")

try:

system\_info = get\_system\_info\_api()

st.write(f"\*\*Memory Usage:\*\* {system\_info.get('memory\_usage', 'N/A')}")

st.write(f"\*\*Available Memory:\*\* {system\_info.get('available\_memory', 'N/A')}")

st.write(f"\*\*Total Memory:\*\* {system\_info.get('total\_memory', 'N/A')}")

st.write(f"\*\*Batch Size:\*\* {system\_info.get('embedding\_batch\_size', 'N/A')}")

if system\_info.get('large\_file\_support'):

st.write(f"\*\*Max File Size:\*\* {system\_info.get('max\_recommended\_file\_size', 'N/A')}")

except:

st.write("\*\*Memory Usage:\*\* N/A")

st.write("\*\*Available Memory:\*\* N/A")

st.write("\*\*Total Memory:\*\* N/A")

# File Information

st.markdown("### 📁 File Information")

if st.session\_state.file\_info:

file\_info = st.session\_state.file\_info

st.write(f"\*\*File Name:\*\* {file\_info.get('name', 'N/A')}")

st.write(f"\*\*File Size:\*\* {file\_info.get('size', 'N/A')}")

st.write(f"\*\*Upload Time:\*\* {file\_info.get('upload\_time', 'N/A')}")

if file\_info.get('large\_file\_processed'):

st.success("✅ Large File Optimized")

if file\_info.get('turbo\_mode'):

st.success("⚡ Turbo Mode Enabled")

else:

try:

file\_info = get\_file\_info\_api()

if file\_info and 'filename' in file\_info:

st.write(f"\*\*File Name:\*\* {file\_info.get('filename', 'N/A')}")

st.write(f"\*\*File Size:\*\* {file\_info.get('file\_size', 0) / 1024:.2f} KB")

st.write(f"\*\*Upload Time:\*\* {file\_info.get('upload\_time', 'N/A')}")

st.write(f"\*\*File Location:\*\* Backend storage")

except:

st.write("\*\*File Info:\*\* Not available")

st.markdown("---")

if st.session\_state.api\_results:

st.markdown("### 📊 Last Results")

result = st.session\_state.api\_results

st.write(f"\*\*Mode:\*\* {result.get('mode', 'N/A')}")

if 'summary' in result:

st.write(f"\*\*Chunks:\*\* {result['summary'].get('chunks', 'N/A')}")

st.write(f"\*\*Storage:\*\* {result['summary'].get('stored', 'N/A')}")

st.write(f"\*\*Model:\*\* {result['summary'].get('embedding\_model', 'N/A')}")

if result['summary'].get('turbo\_mode'):

st.success("⚡ Turbo Mode Used")

if 'conversion\_results' in result['summary']:

conv\_results = result['summary']['conversion\_results']

if conv\_results:

st.write(f"\*\*Type Conversions:\*\* {len(conv\_results.get('successful', []))} successful")

if result['summary'].get('retrieval\_ready'):

st.success("🔍 Retrieval Ready")

if result['summary'].get('large\_file\_processed'):

st.success("🚀 Large File Optimized")

if st.button("🔄 Reset Session", use\_container\_width=True):

# Clean up temporary files

if st.session\_state.get('temp\_file\_path') and os.path.exists(st.session\_state.temp\_file\_path):

os.unlink(st.session\_state.temp\_file\_path)

for key in list(st.session\_state.keys()):

del st.session\_state[key]

st.rerun()

# Mode selection

st.markdown("## 🎯 Choose Processing Mode")

col1, col2, col3 = st.columns(3)

with col1:

if st.button("⚡ Fast Mode", use\_container\_width=True, type="primary"):

st.session\_state.current\_mode = "fast"

st.session\_state.process\_status = {k: "pending" for k in st.session\_state.process\_status}

with col2:

if st.button("⚙️ Config-1 Mode", use\_container\_width=True, type="primary"):

st.session\_state.current\_mode = "config1"

st.session\_state.process\_status = {k: "pending" for k in st.session\_state.process\_status}

with col3:

if st.button("🔬 Deep Config Mode", use\_container\_width=True, type="primary"):

st.session\_state.current\_mode = "deep"

st.session\_state.process\_status = {k: "pending" for k in st.session\_state.process\_status}

if st.session\_state.current\_mode:

pass

# Mode-specific processing

if st.session\_state.current\_mode:

if st.session\_state.current\_mode == "fast":

st.markdown("### ⚡ Fast Mode Configuration")

# Input source selection

input\_source = st.radio("Select Input Source:", ["📁 Upload CSV File", "🗄️ Database Import"], key="fast\_input\_source")

if input\_source == "📁 Upload CSV File":

st.markdown("#### 📤 Upload CSV File")

uploaded\_file = st.file\_uploader("Choose a CSV file", type=["csv"], key="fast\_file\_upload")

if uploaded\_file is not None:

# Use filesystem upload method

with st.spinner("🔄 Streaming file to disk..."):

temp\_path, file\_info = handle\_file\_upload(uploaded\_file)

st.session\_state.temp\_file\_path = temp\_path

st.session\_state.file\_info = file\_info

file\_size\_str = file\_info["size"]

file\_size\_bytes = os.path.getsize(temp\_path)

# Check if file is large

if is\_large\_file(file\_size\_bytes):

st.markdown(f"""

<div class="large-file-warning">

<strong>🚀 Large File Detected: {file\_size\_str}</strong><br>

Large file processing is {'ENABLED' if st.session\_state.process\_large\_files else 'DISABLED'}<br>

<em>File streamed to disk - no memory overload</em>

</div>

""", unsafe\_allow\_html=True)

# removed success banner per request

use\_db\_config = None

else: # Database Import

st.markdown("#### 🗄️ Database Configuration")

col1, col2 = st.columns(2)

with col1:

db\_type = st.selectbox("Database Type", ["mysql", "postgresql"], key="fast\_db\_type")

host = st.text\_input("Host", "localhost", key="fast\_host")

port = st.number\_input("Port", 1, 65535, 3306 if db\_type == "mysql" else 5432, key="fast\_port")

with col2:

username = st.text\_input("Username", key="fast\_username")

password = st.text\_input("Password", type="password", key="fast\_password")

database = st.text\_input("Database", key="fast\_database")

# Test connection and get tables

col1, col2 = st.columns(2)

with col1:

if st.button("🔌 Test Connection", key="fast\_test\_conn", help="Tests DB connectivity. Then click \"List Tables\"."):

res = db\_test\_connection\_api({

"db\_type": db\_type,

"host": host,

"port": port,

"username": username,

"password": password,

"database": database,

})

if res.get("status") == "success":

import time as \_t

st.session\_state["fast\_conn\_ok\_until"] = \_t.time() + 5

else:

st.error(f"❌ Connection failed: {res.get('message', 'Unknown error')}")

# Ephemeral success message under the button

import time as \_t

if st.session\_state.get("fast\_conn\_ok\_until", 0) > \_t.time():

st.markdown(

'<span style="padding:6px 10px; border:1px solid #444; border-radius:6px; background:#2d2d2d; color:#ddd;">Connection successful</span>',

unsafe\_allow\_html=True

)

with col2:

if st.button("📋 List Tables", key="fast\_list\_tables"):

res = db\_list\_tables\_api({

"db\_type": db\_type,

"host": host,

"port": port,

"username": username,

"password": password,

"database": database,

})

st.session\_state["fast\_db\_tables"] = res.get("tables", [])

if st.session\_state["fast\_db\_tables"]:

st.success(f"✅ Found {len(st.session\_state['fast\_db\_tables'])} tables")

else:

st.warning("⚠️ No tables found")

tables = st.session\_state.get("fast\_db\_tables", [])

if tables:

table\_name = st.selectbox("Select Table", tables, key="fast\_table\_select")

use\_db\_config = {

"use\_db": True,

"db\_type": db\_type,

"host": host,

"port": port,

"username": username,

"password": password,

"database": database,

"table\_name": table\_name

}

else:

use\_db\_config = None

# FAST MODE DEFAULTS - No user configuration needed

# Auto-enable turbo mode and set batch size to 256

st.session\_state.use\_turbo = True

st.session\_state.batch\_size = 256

# Display Fast Mode pipeline with FIXED string formatting

processing\_type = "Parallel processing" if st.session\_state.use\_turbo else "Sequential processing"

st.markdown(f"""

<div class="custom-card">

<div class="card-title">Fast Mode Pipeline</div>

<div class="card-content">

• Optimized preprocessing for speed<br>

• Semantic clustering chunking<br>

• paraphrase-MiniLM-L6-v2 embedding model<br>

• Batch embedding with size {st.session\_state.batch\_size}<br>

• {processing\_type}<br>

• FAISS storage for fast retrieval<br>

• 3GB+ file support with disk streaming<br>

</div>

</div>

""", unsafe\_allow\_html=True)

run\_enabled = (

(input\_source == "📁 Upload CSV File" and st.session\_state.get('temp\_file\_path') is not None) or

(input\_source == "🗄️ Database Import" and use\_db\_config is not None)

)

if st.button("🚀 Run Fast Pipeline", type="primary", use\_container\_width=True, disabled=not run\_enabled):

with st.spinner("Running Fast Mode pipeline..."):

try:

if input\_source == "📁 Upload CSV File":

result = call\_fast\_api(

st.session\_state.temp\_file\_path,

st.session\_state.file\_info["name"],

"sqlite",

use\_db\_config,

st.session\_state.use\_openai,

st.session\_state.openai\_api\_key,

st.session\_state.openai\_base\_url,

st.session\_state.process\_large\_files,

st.session\_state.use\_turbo,

st.session\_state.batch\_size

)

else:

result = call\_fast\_api(

None, None, "sqlite", use\_db\_config,

st.session\_state.use\_openai,

st.session\_state.openai\_api\_key,

st.session\_state.openai\_base\_url,

st.session\_state.process\_large\_files,

st.session\_state.use\_turbo,

st.session\_state.batch\_size

)

# Update process status

for step in ["preprocessing", "chunking", "embedding", "storage"]:

st.session\_state.process\_status[step] = "completed"

st.session\_state.process\_timings[step] = "Completed"

st.session\_state.api\_results = result

# Show performance results

if 'summary' in result:

if result['summary'].get('large\_file\_processed'):

st.success("✅ Large file processed efficiently with disk streaming!")

elif result['summary'].get('turbo\_mode'):

st.success("⚡ Turbo mode completed successfully!")

else:

st.success("✅ Fast pipeline completed successfully!")

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

finally:

# Clean up temporary file

if st.session\_state.get('temp\_file\_path') and os.path.exists(st.session\_state.temp\_file\_path):

os.unlink(st.session\_state.temp\_file\_path)

st.session\_state.temp\_file\_path = None

elif st.session\_state.current\_mode == "config1":

st.markdown("### ⚙️ Config-1 Mode Configuration")

# Input source selection

input\_source = st.radio("Select Input Source:", ["📁 Upload CSV File", "🗄️ Database Import"], key="config1\_input\_source")

if input\_source == "📁 Upload CSV File":

st.markdown("#### 📤 Upload CSV File")

uploaded\_file = st.file\_uploader("Choose a CSV file", type=["csv"], key="config1\_file\_upload")

if uploaded\_file is not None:

# Use filesystem upload method

with st.spinner("🔄 Streaming file to disk..."):

temp\_path, file\_info = handle\_file\_upload(uploaded\_file)

st.session\_state.temp\_file\_path = temp\_path

st.session\_state.file\_info = file\_info

file\_size\_str = file\_info["size"]

file\_size\_bytes = os.path.getsize(temp\_path)

# Check if file is large

if is\_large\_file(file\_size\_bytes):

st.markdown(f"""

<div class="large-file-warning">

<strong>🚀 Large File Detected: {file\_size\_str}</strong><br>

Large file processing is {'ENABLED' if st.session\_state.process\_large\_files else 'DISABLED'}<br>

<em>File streamed to disk - no memory overload</em>

</div>

""", unsafe\_allow\_html=True)

st.success(f"✅ \*\*{uploaded\_file.name}\*\* loaded! ({file\_size\_str})")

use\_db\_config = None

else: # Database Import

st.markdown("#### 🗄️ Database Configuration")

col1, col2 = st.columns(2)

with col1:

db\_type = st.selectbox("Database Type", ["mysql", "postgresql"], key="config1\_db\_type")

host = st.text\_input("Host", "localhost", key="config1\_host")

port = st.number\_input("Port", 1, 65535, 3306 if db\_type == "mysql" else 5432, key="config1\_port")

with col2:

username = st.text\_input("Username", key="config1\_username")

password = st.text\_input("Password", type="password", key="config1\_password")

database = st.text\_input("Database", key="config1\_database")

# Test connection and get tables

col1, col2 = st.columns(2)

with col1:

if st.button("🔌 Test Connection", key="config1\_test\_conn", help="Tests DB connectivity. Then click \"List Tables\"."):

res = db\_test\_connection\_api({

"db\_type": db\_type,

"host": host,

"port": port,

"username": username,

"password": password,

"database": database,

})

if res.get("status") == "success":

import time as \_t

st.session\_state["config1\_conn\_ok\_until"] = \_t.time() + 5

else:

st.error(f"❌ Connection failed: {res.get('message', 'Unknown error')}")

# Ephemeral success message under the button

import time as \_t

if st.session\_state.get("config1\_conn\_ok\_until", 0) > \_t.time():

st.markdown(

'<span style="padding:6px 10px; border:1px solid #444; border-radius:6px; background:#2d2d2d; color:#ddd;">Connection successful</span>',

unsafe\_allow\_html=True

)

with col2:

if st.button("📋 List Tables", key="config1\_list\_tables"):

res = db\_list\_tables\_api({

"db\_type": db\_type,

"host": host,

"port": port,

"username": username,

"password": password,

"database": database,

})

st.session\_state["config1\_db\_tables"] = res.get("tables", [])

if st.session\_state["config1\_db\_tables"]:

st.success(f"✅ Found {len(st.session\_state['config1\_db\_tables'])} tables")

else:

st.warning("⚠️ No tables found")

tables = st.session\_state.get("config1\_db\_tables", [])

if tables:

table\_name = st.selectbox("Select Table", tables, key="config1\_table\_select")

use\_db\_config = {

"use\_db": True,

"db\_type": db\_type,

"host": host,

"port": port,

"username": username,

"password": password,

"database": database,

"table\_name": table\_name

}

else:

use\_db\_config = None

# Config-1 parameters (refactored into tabs)

st.markdown("#### ⚙️ Configuration Parameters")

tab\_chunk, tab\_embed, tab\_store = st.tabs(["Chunking", "Embedding", "Storage & Retrieval"])

# Defaults to ensure variables exist for payload

chunk\_method = st.session\_state.get("config1\_chunk", "recursive")

chunk\_size = st.session\_state.get("config1\_size", 800)

overlap = st.session\_state.get("config1\_overlap", 20)

document\_key\_column = st.session\_state.get("config1\_document\_key\_column", "")

token\_limit = st.session\_state.get("config1\_token\_limit", 2000)

model\_choice = st.session\_state.get("config1\_model", "paraphrase-MiniLM-L6-v2")

storage\_choice = st.session\_state.get("config1\_storage", "faiss")

config1\_retrieval\_metric = st.session\_state.get("config1\_retrieval\_metric", "cosine")

with tab\_chunk:

st.markdown("#### 📦 Chunking")

chunk\_method = st.selectbox("Chunking method", ["fixed", "recursive", "semantic", "document"], key="config1\_chunk")

if chunk\_method in ["fixed", "recursive"]:

chunk\_size = st.slider("Chunk size", 100, 2000, int(chunk\_size), key="config1\_size")

overlap = st.slider("Overlap", 0, 500, int(overlap), key="config1\_overlap")

elif chunk\_method == "document":

st.markdown("#### 📄 Document Chunking Options")

document\_key\_column = st.text\_input(

"Key column (leave blank to use first column)",

key="config1\_document\_key\_column",

value=str(document\_key\_column) if document\_key\_column else ""

)

token\_limit = st.number\_input(

"Token limit per chunk",

min\_value=200,

max\_value=10000,

value=int(token\_limit),

step=100,

key="config1\_token\_limit"

)

# removed explanatory info text per request

with tab\_embed:

st.markdown("#### 🤖 Embedding")

model\_choice = st.selectbox("Embedding model",

["all-MiniLM-L6-v2", "paraphrase-MiniLM-L6-v2", "text-embedding-ada-002"],

key="config1\_model")

st.markdown("#### ⚡ Performance")

st.session\_state.use\_turbo = st.checkbox(

"Enable Turbo Mode",

value=st.session\_state.use\_turbo,

help="Faster processing with parallel operations",

key="config1\_use\_turbo"

)

st.session\_state.batch\_size = st.slider(

"Embedding Batch Size",

min\_value=64,

max\_value=512,

value=st.session\_state.batch\_size,

step=64,

help="Larger batches = faster processing (requires more memory)",

key="config1\_batch\_size"

)

with tab\_store:

st.markdown("#### 💾 Storage")

storage\_choice = st.selectbox("Vector storage", ["faiss", "chromadb"], key="config1\_storage", index=["faiss","chromadb"].index(storage\_choice) if storage\_choice in ["faiss","chromadb"] else 0)

st.markdown("#### 🔎 Retrieval Metric")

config1\_retrieval\_metric = st.selectbox(

"Similarity metric",

["cosine", "dot", "euclidean"],

index=["cosine","dot","euclidean"].index(config1\_retrieval\_metric) if config1\_retrieval\_metric in ["cosine","dot","euclidean"] else 0,

key="config1\_retrieval\_metric"

)

# removed explanatory captions per request

# removed turbo mode success banner per request

run\_enabled = (

(input\_source == "📁 Upload CSV File" and st.session\_state.get('temp\_file\_path') is not None) or

(input\_source == "🗄️ Database Import" and use\_db\_config is not None)

)

if st.button("🚀 Run Config-1 Pipeline", type="primary", use\_container\_width=True, disabled=not run\_enabled):

with st.spinner("Running Config-1 pipeline..."):

try:

config = {

"chunk\_method": chunk\_method,

"chunk\_size": chunk\_size if 'chunk\_size' in locals() else 800,

"overlap": overlap if 'overlap' in locals() else 20,

"model\_choice": model\_choice,

"storage\_choice": storage\_choice,

}

if chunk\_method == "document":

if 'document\_key\_column' in locals() and document\_key\_column:

config["document\_key\_column"] = document\_key\_column

if 'token\_limit' in locals() and token\_limit:

config["token\_limit"] = int(token\_limit)

# include retrieval metric for storage compatibility

if 'config1\_retrieval\_metric' in locals() and config1\_retrieval\_metric:

config["retrieval\_metric"] = config1\_retrieval\_metric

if input\_source == "📁 Upload CSV File":

result = call\_config1\_api(

st.session\_state.temp\_file\_path,

st.session\_state.file\_info["name"],

config,

use\_db\_config,

st.session\_state.use\_openai,

st.session\_state.openai\_api\_key,

st.session\_state.openai\_base\_url,

st.session\_state.process\_large\_files,

st.session\_state.use\_turbo,

st.session\_state.batch\_size

)

else:

result = call\_config1\_api(

None, None, config, use\_db\_config,

st.session\_state.use\_openai,

st.session\_state.openai\_api\_key,

st.session\_state.openai\_base\_url,

st.session\_state.process\_large\_files,

st.session\_state.use\_turbo,

st.session\_state.batch\_size

)

# Mark all as completed

for step in ["preprocessing", "chunking", "embedding", "storage"]:

st.session\_state.process\_status[step] = "completed"

st.session\_state.process\_timings[step] = "Completed"

st.session\_state.api\_results = result

# Show performance results

if 'summary' in result:

if result['summary'].get('large\_file\_processed'):

st.success("✅ Large file processed efficiently with disk streaming!")

elif result['summary'].get('turbo\_mode'):

st.success("⚡ Turbo mode completed successfully!")

else:

st.success("✅ Config-1 pipeline completed successfully!")

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

finally:

# Clean up temporary file

if st.session\_state.get('temp\_file\_path') and os.path.exists(st.session\_state.temp\_file\_path):

os.unlink(st.session\_state.temp\_file\_path)

st.session\_state.temp\_file\_path = None

elif st.session\_state.current\_mode == "deep":

st.markdown("### 🔬 Deep Config Mode - Comprehensive Workflow")

# Import enhanced functions from backend

try:

from backend import (

preprocess\_csv\_enhanced,

profile\_nulls\_enhanced,

suggest\_null\_strategy\_enhanced,

apply\_null\_strategies\_enhanced,

remove\_stopwords\_from\_text\_column\_enhanced,

process\_text\_enhanced,

chunk\_fixed\_enhanced,

chunk\_semantic\_cluster\_enhanced,

document\_based\_chunking\_enhanced,

chunk\_recursive\_keyvalue\_enhanced,

embed\_texts\_enhanced,

store\_chroma\_enhanced,

store\_faiss\_enhanced

)

except ImportError as e:

st.error(f"Failed to import enhanced backend functions: {e}")

st.error("Please ensure backend.py contains the enhanced functions")

st.stop()

# Initialize deep config session state variables

if "deep\_config\_step" not in st.session\_state:

st.session\_state.deep\_config\_step = 0

if "preprocessing\_config" not in st.session\_state:

st.session\_state.preprocessing\_config = {}

if "chunking\_config" not in st.session\_state:

st.session\_state.chunking\_config = {}

if "embedding\_config" not in st.session\_state:

st.session\_state.embedding\_config = {}

if "storage\_config" not in st.session\_state:

st.session\_state.storage\_config = {}

if "deep\_df" not in st.session\_state:

st.session\_state.deep\_df = pd.DataFrame()

if "deep\_file\_meta" not in st.session\_state:

st.session\_state.deep\_file\_meta = {}

if "deep\_numeric\_meta" not in st.session\_state:

st.session\_state.deep\_numeric\_meta = []

if "deep\_chunks" not in st.session\_state:

st.session\_state.deep\_chunks = []

if "deep\_chunking\_result" not in st.session\_state:

st.session\_state.deep\_chunking\_result = None

if "deep\_embedding\_result" not in st.session\_state:

st.session\_state.deep\_embedding\_result = None

if "deep\_meta\_numeric\_cols" not in st.session\_state:

st.session\_state.deep\_meta\_numeric\_cols = []

if "deep\_meta\_categorical\_cols" not in st.session\_state:

st.session\_state.deep\_meta\_categorical\_cols = []

if "deep\_store\_metadata\_enabled" not in st.session\_state:

st.session\_state.deep\_store\_metadata\_enabled = True

# Input source selection for Deep Config

input\_source = st.radio("Select Input Source:", ["📁 Upload CSV File", "🗄️ Database Import"], key="deep\_input\_source")

if input\_source == "📁 Upload CSV File":

uploaded\_file = st.file\_uploader("Upload CSV file", type=["csv"], key="deep\_file\_upload")

else: # Database Import

st.markdown("#### 🗄️ Database Configuration")

col1, col2 = st.columns(2)

with col1:

db\_type = st.selectbox("Database Type", ["mysql", "postgresql"], key="deep\_db\_type")

host = st.text\_input("Host", "localhost", key="deep\_host")

port = st.number\_input("Port", value=3306 if db\_type == "mysql" else 5432, key="deep\_port")

with col2:

username = st.text\_input("Username", key="deep\_username")

password = st.text\_input("Password", type="password", key="deep\_password")

database = st.text\_input("Database Name", key="deep\_database")

# Test Connection and List Tables

col1, col2 = st.columns(2)

with col1:

if st.button("🔌 Test Connection", key="deep\_test\_conn", help="Tests DB connectivity. Then click \"List Tables\"."):

res = db\_test\_connection\_api({

"db\_type": db\_type,

"host": host,

"port": port,

"username": username,

"password": password,

"database": database,

})

if res.get("status") == "success":

import time as \_t

st.session\_state["deep\_conn\_ok\_until"] = \_t.time() + 5

else:

st.error(f"❌ Connection failed: {res.get('message', 'Unknown error')}")

# Ephemeral success message under the button

import time as \_t

if st.session\_state.get("deep\_conn\_ok\_until", 0) > \_t.time():

st.markdown(

'<span style="padding:6px 10px; border:1px solid #444; border-radius:6px; background:#2d2d2d; color:#ddd;">Connection successful</span>',

unsafe\_allow\_html=True

)

with col2:

if st.button("📋 List Tables", key="deep\_list\_tables"):

res = db\_list\_tables\_api({

"db\_type": db\_type,

"host": host,

"port": port,

"username": username,

"password": password,

"database": database,

})

if "error" in res:

st.error(f"❌ Failed to list tables: {res['error']}")

else:

st.session\_state.deep\_available\_tables = res.get("tables", [])

st.success(f"✅ Found {len(st.session\_state.deep\_available\_tables)} tables")

# Table selection

if hasattr(st.session\_state, 'deep\_available\_tables') and st.session\_state.deep\_available\_tables:

table\_name = st.selectbox("Select Table", st.session\_state.deep\_available\_tables, key="deep\_table\_name")

# Create DB config for Deep Config

use\_db\_config = {

"use\_db": True,

"db\_type": db\_type,

"host": host,

"port": port,

"username": username,

"password": password,

"database": database,

"table\_name": table\_name

}

else:

use\_db\_config = None

table\_name = None

uploaded\_file = None # No file upload for DB mode

# Handle data loading for both CSV and DB

if uploaded\_file or (input\_source == "🗄️ Database Import" and use\_db\_config):

if st.session\_state.deep\_config\_step == 0:

if uploaded\_file:

# CSV file upload

df = pd.read\_csv(uploaded\_file)

# Validate and normalize headers

df.columns = validate\_and\_normalize\_headers(df.columns)

st.session\_state.deep\_df = df

st.session\_state.deep\_file\_info = {

"source": "csv",

"filename": uploaded\_file.name,

"size": len(uploaded\_file.getvalue())

}

else:

# DB import - load data via API

with st.spinner("🔄 Loading data from database..."):

st.session\_state.deep\_db\_config = use\_db\_config

# Load data directly from database for Deep Config UI preview

try:

from backend import connect\_mysql, connect\_postgresql, import\_table\_to\_dataframe

if use\_db\_config['db\_type'] == 'mysql':

conn = connect\_mysql(

use\_db\_config['host'],

use\_db\_config['port'],

use\_db\_config['username'],

use\_db\_config['password'],

use\_db\_config['database']

)

elif use\_db\_config['db\_type'] == 'postgresql':

conn = connect\_postgresql(

use\_db\_config['host'],

use\_db\_config['port'],

use\_db\_config['username'],

use\_db\_config['password'],

use\_db\_config['database']

)

df = import\_table\_to\_dataframe(conn, use\_db\_config['table\_name'])

conn.close()

# Validate and normalize headers

df.columns = validate\_and\_normalize\_headers(df.columns)

st.session\_state.deep\_df = df

st.session\_state.deep\_file\_info = {

"source": f"db:{use\_db\_config['db\_type']}",

"table": use\_db\_config['table\_name'],

"database": use\_db\_config['database'],

"rows": len(df),

"columns": len(df.columns)

}

st.success(f"✅ Successfully loaded {len(df)} rows from {use\_db\_config['table\_name']}")

except Exception as e:

st.error(f"❌ Failed to load database data: {str(e)}")

st.error(f"💡 \*\*Troubleshooting\*\*: Make sure you have the required database drivers installed:")

if use\_db\_config['db\_type'] == 'postgresql':

st.error(" - For PostgreSQL: `pip install psycopg2-binary`")

elif use\_db\_config['db\_type'] == 'mysql':

st.error(" - For MySQL: `pip install mysql-connector-python`")

st.session\_state.deep\_df = pd.DataFrame()

st.subheader("Data preview")

if not st.session\_state.deep\_df.empty:

# Data summary metrics

col1, col2, col3, col4 = st.columns(4)

with col1:

st.metric("Total Rows", len(st.session\_state.deep\_df))

with col2:

st.metric("Total Columns", len(st.session\_state.deep\_df.columns))

with col3:

st.metric("Memory Usage", f"{st.session\_state.deep\_df.memory\_usage(deep=True).sum() / 1024\*\*2:.1f} MB")

with col4:

null\_count = st.session\_state.deep\_df.isnull().sum().sum()

st.metric("Null Values", null\_count)

# Enhanced scrollable dataframe

st.subheader("Data Preview")

st.dataframe(

st.session\_state.deep\_df,

height=300,

use\_container\_width=True,

hide\_index=False

)

else:

# Show DB import info

if hasattr(st.session\_state, 'deep\_file\_info') and st.session\_state.deep\_file\_info.get('source', '').startswith('db:'):

st.info(f"📊 Database Import: {st.session\_state.deep\_file\_info.get('table', 'Unknown table')} from {st.session\_state.deep\_file\_info.get('database', 'Unknown database')}")

st.info("Data will be loaded during pipeline execution")

# Step 1: Default preprocessing

if st.session\_state.deep\_config\_step == 0:

if st.button("Run Default Preprocessing", key="deep\_default\_preprocessing"):

with st.spinner("🔄 Running preprocessing via API..."):

try:

# Determine input source and call API

if uploaded\_file:

# CSV file upload

temp\_path = None

try:

# Create temporary file

import tempfile

with tempfile.NamedTemporaryFile(delete=False, suffix='.csv') as tmp\_file:

tmp\_file.write(uploaded\_file.getvalue())

temp\_path = tmp\_file.name

result = call\_deep\_config\_preprocess\_api(temp\_path, uploaded\_file.name, None)

finally:

# Clean up temp file

if temp\_path and os.path.exists(temp\_path):

os.unlink(temp\_path)

else:

# DB import

result = call\_deep\_config\_preprocess\_api(None, None, use\_db\_config)

if "error" in result:

st.error(f"❌ Preprocessing failed: {result['error']}")

else:

st.success(f"✅ Preprocessing completed successfully!")

st.info(f"📊 \*\*Results\*\*: {result.get('rows', 'N/A')} rows, {result.get('columns', 'N/A')} columns")

# Update session state with API results

st.session\_state.deep\_file\_meta = result.get('file\_info', {})

st.session\_state.deep\_numeric\_meta = []

st.session\_state.deep\_config\_step = 1

st.rerun()

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

# Step 2: Type conversion

if st.session\_state.deep\_config\_step == 1:

st.sidebar.checkbox("Default Preprocessing Done", value=True, disabled=True, key="deep\_step1\_preprocessing\_done")

st.subheader("Data Type Conversion")

# Back button

if st.button("Back to Upload", key="deep\_back\_to\_upload"):

st.session\_state.deep\_chunking\_result = None

st.session\_state.deep\_embedding\_result = None

st.session\_state.deep\_config\_step = 0

st.rerun()

# Check if data is available

if st.session\_state.deep\_df.empty:

st.error("❌ No data available for type conversion. Please go back and load data first.")

if st.button("Back to Data Loading", key="deep\_back\_to\_data\_loading\_step1"):

st.session\_state.deep\_config\_step = 0

st.rerun()

st.stop()

# Smart suggestions based on column names and data patterns

def get\_smart\_suggestion(col\_name, col\_data):

col\_name\_lower = col\_name.lower()

if any(word in col\_name\_lower for word in ['date', 'time', 'created', 'updated', 'timestamp', 'birth', 'join']):

return 'datetime'

if any(word in col\_name\_lower for word in ['flag', 'is\_', 'has\_', 'active', 'enabled', 'status', 'complaint']):

return 'boolean'

if any(word in col\_name\_lower for word in ['count', 'score', 'price', 'amount', 'quantity', 'age', 'id']):

return 'float64'

if col\_data.dtype == 'object':

sample\_values = col\_data.dropna().head(10)

if len(sample\_values) > 0:

date\_patterns = [str(val) for val in sample\_values if

any(char in str(val) for char in ['-', '/', ':']) and

len(str(val)) > 8]

if len(date\_patterns) > len(sample\_values) \* 0.7:

return 'datetime'

if col\_data.dtype == 'object':

sample\_values = col\_data.dropna().head(10)

if len(sample\_values) > 0:

bool\_values = [str(val).lower() for val in sample\_values if

str(val).lower() in ['true', 'false', 'yes', 'no', '1', '0', 'y', 'n']]

if len(bool\_values) > len(sample\_values) \* 0.7:

return 'boolean'

if str(col\_data.dtype) == 'int64':

return 'int64'

if str(col\_data.dtype) == 'float64':

return 'float64'

return 'object'

# Generate smart suggestions

suggestions = {}

for col in st.session\_state.deep\_df.columns:

suggestions[col] = get\_smart\_suggestion(col, st.session\_state.deep\_df[col])

# Display smart suggestions overview

st.write("🤖 \*\*Smart Suggestions Overview:\*\*")

suggestion\_df = pd.DataFrame({

'Column': st.session\_state.deep\_df.columns,

'Current Type': [str(dtype) for dtype in st.session\_state.deep\_df.dtypes],

'Suggested Type': [suggestions[col] for col in st.session\_state.deep\_df.columns],

'Reason': [

f"Detected {'date/time' if suggestions[col] == 'datetime' else 'boolean' if suggestions[col] == 'boolean' else 'numeric' if suggestions[col] in ['int64','float64'] else 'text'} patterns"

for col in st.session\_state.deep\_df.columns

]

})

st.dataframe(suggestion\_df, use\_container\_width=True, height=250)

# Group columns by their current data type

dtype\_groups = {}

for col in st.session\_state.deep\_df.columns:

current\_type = str(st.session\_state.deep\_df[col].dtype)

if current\_type not in dtype\_groups:

dtype\_groups[current\_type] = []

dtype\_groups[current\_type].append(col)

# Initialize type conversions dictionary

type\_conversions = {}

st.write("📋 \*\*Grouped Conversion Interface:\*\*")

# Create conversion interface for each data type group

for current\_type, columns in dtype\_groups.items():

st.write(f"\*\*Current Type: `{current\_type}` ({len(columns)} columns)\*\*")

# Get the most common suggestion for this group

group\_suggestions = [suggestions[col] for col in columns]

most\_common\_suggestion = max(set(group\_suggestions), key=group\_suggestions.count)

# Target conversion type selection

options = ["No change", "object", "int64", "float64", "datetime", "boolean"]

default\_map = {

'object': 1,

'int64': 2,

'float64': 3,

'datetime': 4,

'boolean': 5,

}

target\_type = st.selectbox(

f"Convert {current\_type} columns to:",

options,

index=default\_map.get(most\_common\_suggestion, 0),

key=f"deep\_target\_type\_{current\_type}"

)

if target\_type != "No change":

# Column selection checkboxes with smart suggestions

st.write("Select columns to convert:")

selected\_columns = []

# Create columns layout for checkboxes

cols\_per\_row = 3

for i in range(0, len(columns), cols\_per\_row):

cols = st.columns(cols\_per\_row)

for j, col in enumerate(columns[i:i+cols\_per\_row]):

with cols[j]:

suggested\_type = suggestions[col]

is\_suggested = suggested\_type == target\_type

if st.checkbox(f"{col} (suggested: {suggested\_type})",

value=is\_suggested,

key=f"deep\_convert\_{col}"):

selected\_columns.append(col)

# Add selected columns to type conversions

for col in selected\_columns:

type\_conversions[col] = target\_type

# Show smart suggestion summary

suggested\_cols = [col for col in columns if suggestions[col] == target\_type]

if suggested\_cols:

st.info(f"💡 Smart suggestion: {target\_type} for {', '.join(suggested\_cols)}")

st.divider()

# Apply type conversion button

if st.button("Apply Type Conversion", key="deep\_apply\_type\_conversion"):

if type\_conversions:

with st.spinner(f"🔄 Converting {len(type\_conversions)} columns via API..."):

try:

result = call\_deep\_config\_type\_convert\_api(type\_conversions)

if "error" in result:

st.error(f"❌ Type conversion failed: {result['error']}")

else:

st.success(f"✅ Type conversion completed successfully!")

st.info(f"📊 \*\*Converted\*\*: {len(type\_conversions)} columns")

# Update session state with API results

st.session\_state.deep\_file\_meta = result.get('file\_info', {})

st.session\_state.deep\_numeric\_meta = []

st.session\_state.deep\_config\_step = 2

st.rerun()

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

else:

st.info("No type conversions selected. Moving to next step.")

st.session\_state.deep\_config\_step = 2

st.rerun()

# Skip button

if st.button("Apply No Changes (Skip Type Conversion)", key="deep\_skip\_type\_conversion"):

st.info("Type conversion skipped.")

st.session\_state.deep\_config\_step = 2

st.rerun()

# Step 3: Null handling

if st.session\_state.deep\_config\_step == 2:

st.sidebar.checkbox("Default Preprocessing Done", value=True, disabled=True, key="deep\_step2\_preprocessing\_done")

st.sidebar.checkbox("Type Conversion Done", value=True, disabled=True, key="deep\_step2\_type\_conversion\_done")

st.subheader("Null Handling")

# Back button

if st.button("Back to Data Types", key="deep\_back\_to\_types"):

st.session\_state.deep\_chunking\_result = None

st.session\_state.deep\_embedding\_result = None

st.session\_state.deep\_config\_step = 1

st.rerun()

# Check if data is available

if st.session\_state.deep\_df.empty:

st.error("❌ No data available for null handling. Please go back and load data first.")

if st.button("Back to Data Loading", key="deep\_back\_to\_data\_loading"):

st.session\_state.deep\_config\_step = 0

st.rerun()

st.stop()

st.write("Smart suggestions based on null ratio, dtype, and column semantics.")

# Smart suggestions overview

profile = profile\_nulls\_enhanced(st.session\_state.deep\_df)

profile['suggested'] = profile.apply(lambda r: suggest\_null\_strategy\_enhanced(r['column'], st.session\_state.deep\_df[r['column']]), axis=1)

filtered\_profile = profile[profile['null\_count'] > 0].copy()

if filtered\_profile.empty:

st.info("No null values detected; you can proceed to the next step.")

if st.button("Proceed to Stop Words Removal", key="deep\_proceed\_no\_nulls"):

st.session\_state.deep\_config\_step = 4

st.rerun()

else:

st.write("🤖 Smart Suggestions (Nulls):")

st.dataframe(

filtered\_profile[['column','dtype','null\_count','null\_pct','suggested']].sort\_values('null\_pct', ascending=False),

use\_container\_width=True,

height=250

)

# Grouped controls by dtype

st.write("📋 \*\*Grouped Controls:\*\*")

type\_groups = {}

for \_, row in filtered\_profile.iterrows():

type\_groups.setdefault(row['dtype'], []).append(row['column'])

null\_strategies = {}

options = ["No change", "leave", "drop", "mean", "median", "mode", "zero", "unknown", "ffill", "bfill"]

for dtype\_name, cols in type\_groups.items():

st.write(f"\*\*{dtype\_name}\*\* ({len(cols)} columns)")

group\_sugs = [filtered\_profile.loc[filtered\_profile['column']==c, 'suggested'].values[0] for c in cols]

if group\_sugs:

from collections import Counter

most\_common = Counter(group\_sugs).most\_common(1)[0][0]

default\_index = options.index(most\_common) if most\_common in options else 0

else:

default\_index = 0

group\_choice = st.selectbox(

f"Default strategy for {dtype\_name}:",

options,

index=default\_index,

key=f"deep\_null\_group\_{dtype\_name}"

)

# Per-column overrides

cols\_per\_row = 3

for i in range(0, len(cols), cols\_per\_row):

ccols = st.columns(cols\_per\_row)

for j, col in enumerate(cols[i:i+cols\_per\_row]):

with ccols[j]:

sug = filtered\_profile.loc[filtered\_profile['column']==col, 'suggested'].values[0]

choice = st.selectbox(

f"{col} (sugg: {sug})",

options,

index=options.index(sug) if sug in options else default\_index,

key=f"deep\_null\_choice\_{col}"

)

if choice != "No change":

null\_strategies[col] = choice

st.divider()

if st.button("Apply Null Handling", key="deep\_apply\_null\_handling"):

with st.spinner("🔄 Applying null handling via API..."):

try:

result = call\_deep\_config\_null\_handle\_api(null\_strategies)

if "error" in result:

st.error(f"❌ Null handling failed: {result['error']}")

else:

st.success(f"✅ Null handling completed successfully!")

st.info(f"📊 \*\*Processed\*\*: {len(null\_strategies)} columns with null strategies")

# Update session state with API results

st.session\_state.deep\_file\_meta = result.get('file\_info', {})

st.session\_state.deep\_numeric\_meta = []

st.session\_state.deep\_config\_step = 4

st.rerun()

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

# Skip button

if st.button("Apply No Changes (Skip Null Handling)", key="deep\_skip\_null\_handling"):

st.info("Null handling skipped.")

st.session\_state.deep\_config\_step = 4

st.rerun()

# Step 4: Stop Words Removal

if st.session\_state.deep\_config\_step == 4:

st.subheader("Stop Words Removal")

# Back button

if st.button("Back to Null Handling", key="deep\_back\_to\_nulls"):

st.session\_state.deep\_chunking\_result = None

st.session\_state.deep\_embedding\_result = None

st.session\_state.deep\_config\_step = 2

st.rerun()

# Check if data is available

if st.session\_state.deep\_df.empty:

st.error("❌ No data available for stop words removal. Please go back and load data first.")

if st.button("Back to Data Loading", key="deep\_back\_to\_data\_loading\_step4"):

st.session\_state.deep\_config\_step = 0

st.rerun()

st.stop()

text\_cols = st.session\_state.deep\_df.select\_dtypes(include=["object"]).columns

if text\_cols.empty:

st.info("No text column found to apply stop words removal.")

if st.button("Proceed to Normalization (Skip Stopwords)", key="deep\_proceed\_no\_text\_stopwords"):

st.session\_state.deep\_config\_step = 5

st.rerun()

else:

choice = st.radio(

"Choose stop word handling:",

["Apply stop word removal", "Skip stop word removal"],

index=1,

key="deep\_stopword\_choice"

)

if st.button("Continue", key="deep\_continue\_stopwords"):

with st.spinner("🔄 Processing stop words via API..."):

try:

remove\_stopwords = choice == "Apply stop word removal"

result = call\_deep\_config\_stopwords\_api(remove\_stopwords)

if "error" in result:

st.error(f"❌ Stop words processing failed: {result['error']}")

else:

if remove\_stopwords:

st.success("✅ Stop words removed from detected text columns.")

else:

st.info("Stop words removal skipped.")

# Update session state with API results

st.session\_state.deep\_file\_meta = result.get('file\_info', {})

st.session\_state.deep\_numeric\_meta = []

st.session\_state.deep\_config\_step = 5

st.rerun()

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

# Step 5: Text Normalization

if st.session\_state.deep\_config\_step == 5:

st.subheader("Text Normalization")

# Back button

if st.button("Back to Stop Words", key="deep\_back\_to\_stopwords"):

st.session\_state.deep\_chunking\_result = None

st.session\_state.deep\_embedding\_result = None

st.session\_state.deep\_config\_step = 4

st.rerun()

choice = st.radio(

"Choose an option:",

[

"Apply lemmatization",

"Apply stemming",

"Skip text normalization",

],

index=2,

key="deep\_text\_norm\_choice",

)

if st.button("Apply Changes", key="deep\_apply\_text\_norm"):

with st.spinner("🔄 Applying text normalization via API..."):

try:

# Determine method based on choice

if choice == "Apply lemmatization":

method = "lemmatize"

elif choice == "Apply stemming":

method = "stem"

else:

method = "none"

result = call\_deep\_config\_normalize\_api(method)

if "error" in result:

st.error(f"❌ Text normalization failed: {result['error']}")

else:

if method == "lemmatize":

st.success("✅ Applied lemmatization")

elif method == "stem":

st.success("✅ Applied stemming")

else:

st.info("Skipped text normalization")

# Update session state with API results

st.session\_state.deep\_file\_meta = result.get('file\_info', {})

st.session\_state.deep\_numeric\_meta = []

# Add download button for preprocessed data

st.markdown("---")

st.subheader("📥 Download Preprocessed Data")

if st.button("📄 Download Preprocessed CSV", key="deep\_download\_preprocessed"):

try:

csv\_data = download\_deep\_config\_preprocessed()

filename = "preprocessed\_data.csv"

st.download\_button(

label="⬇️ Download Preprocessed Data",

data=csv\_data,

file\_name=filename,

mime="text/csv",

use\_container\_width=True

)

except Exception as e:

st.error(f"Download failed: {str(e)}")

st.session\_state.deep\_config\_step = 6

st.rerun()

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

# Step 6: Final metadata and chunking

if st.session\_state.deep\_config\_step == 6:

st.sidebar.checkbox("Default Preprocessing Done", value=True, disabled=True, key="deep\_step6\_preprocessing\_done")

st.sidebar.checkbox("Type Conversion Done", value=True, disabled=True, key="deep\_step6\_type\_conversion\_done")

st.sidebar.checkbox("Null Handling Done", value=True, disabled=True, key="deep\_step6\_null\_handling\_done")

# Back button

if st.button("Back to Text Normalization", key="deep\_back\_to\_text\_norm"):

st.session\_state.deep\_chunking\_result = None

st.session\_state.deep\_embedding\_result = None

st.session\_state.deep\_config\_step = 5

st.rerun()

# Download preprocessed CSV

csv\_data = st.session\_state.deep\_df.to\_csv(index=False).encode("utf-8")

# Generate appropriate filename based on data source

if uploaded\_file:

filename = f"processed\_{uploaded\_file.name}"

else:

# DB import case

table\_name = st.session\_state.deep\_file\_info.get('table', 'database\_table')

filename = f"processed\_{table\_name}.csv"

st.download\_button(

label="Download preprocessed CSV",

data=csv\_data,

file\_name=filename,

mime="text/csv",

)

st.divider()

# Metadata selection

st.subheader("Select metadata columns to store in ChromaDB")

store\_metadata = st.checkbox(

"Store metadata in ChromaDB",

value=st.session\_state.deep\_store\_metadata\_enabled,

help="Enable this to store selected metadata columns in ChromaDB for filtering and retrieval",

key="deep\_store\_metadata\_checkbox"

)

st.session\_state.deep\_store\_metadata\_enabled = store\_metadata

if store\_metadata:

df\_current = st.session\_state.deep\_df

numeric\_candidates = df\_current.select\_dtypes(include=['number']).columns.tolist()

max\_categorical\_cardinality = 50

raw\_categorical = df\_current.select\_dtypes(include=['object']).columns.tolist()

categorical\_candidates = [c for c in raw\_categorical if df\_current[c].nunique(dropna=True) <= max\_categorical\_cardinality]

max\_numeric\_cap = min(10, len(numeric\_candidates))

max\_categorical\_cap = min(5, len(categorical\_candidates))

def \_num\_rank(col):

try:

var = float(pd.to\_numeric(df\_current[col], errors='coerce').var())

except Exception:

var = 0.0

miss = float(pd.to\_numeric(df\_current[col], errors='coerce').isna().mean())

return (-var, miss)

ranked\_numeric = sorted(numeric\_candidates, key=\_num\_rank)

def \_cat\_rank(col):

s = df\_current[col]

miss = float(s.isna().mean())

uniq = int(s.nunique(dropna=True))

return (miss, uniq)

ranked\_categorical = sorted(categorical\_candidates, key=\_cat\_rank)

if numeric\_candidates:

num\_numeric\_to\_store = st.number\_input(

"How many numeric columns to include (store min/mean/max per chunk)",

min\_value=0,

max\_value=max\_numeric\_cap,

value=max\_numeric\_cap,

key="deep\_num\_numeric"

)

default\_numeric = ranked\_numeric[: int(num\_numeric\_to\_store)]

selected\_numeric\_cols = st.multiselect(

"Select numeric columns",

options=numeric\_candidates,

default=(st.session\_state.deep\_meta\_numeric\_cols[: int(num\_numeric\_to\_store)] if st.session\_state.deep\_meta\_numeric\_cols else default\_numeric),

key="deep\_selected\_numeric"

)

if len(selected\_numeric\_cols) > int(num\_numeric\_to\_store):

selected\_numeric\_cols = selected\_numeric\_cols[: int(num\_numeric\_to\_store)]

else:

st.info("No numeric columns detected.")

selected\_numeric\_cols = []

if categorical\_candidates:

num\_categorical\_to\_store = st.number\_input(

"How many categorical columns to include (store mode per chunk)",

min\_value=0,

max\_value=max\_categorical\_cap,

value=min(2, max\_categorical\_cap),

key="deep\_num\_categorical"

)

default\_categorical = ranked\_categorical[: int(num\_categorical\_to\_store)]

selected\_categorical\_cols = st.multiselect(

"Select categorical columns",

options=categorical\_candidates,

default=(st.session\_state.deep\_meta\_categorical\_cols[: int(num\_categorical\_to\_store)] if st.session\_state.deep\_meta\_categorical\_cols else default\_categorical),

key="deep\_selected\_categorical"

)

if len(selected\_categorical\_cols) > int(num\_categorical\_to\_store):

selected\_categorical\_cols = selected\_categorical\_cols[: int(num\_categorical\_to\_store)]

high\_card = [c for c in selected\_categorical\_cols if df\_current[c].nunique(dropna=True) > max\_categorical\_cardinality]

if high\_card:

st.warning(f"High-cardinality categorical columns selected: {', '.join(high\_card)}. This may reduce filter usefulness.")

else:

st.info("No low-cardinality categorical columns detected.")

selected\_categorical\_cols = []

st.session\_state.deep\_meta\_numeric\_cols = selected\_numeric\_cols

st.session\_state.deep\_meta\_categorical\_cols = selected\_categorical\_cols

if st.button("Apply Metadata Selection", key="deep\_apply\_metadata"):

st.session\_state.metadata\_selection\_applied = True

st.success("Metadata selection saved. These fields will be stored with chunks in ChromaDB.")

else:

st.info("Metadata storage is disabled. No metadata will be stored in ChromaDB.")

st.session\_state.deep\_meta\_numeric\_cols = []

st.session\_state.deep\_meta\_categorical\_cols = []

# Add chunking section

st.divider()

st.subheader("CSV Chunking")

if st.button("Start Chunking Process", key="deep\_start\_chunking"):

st.session\_state.deep\_config\_step = 7

st.rerun()

# Step 7: Chunking Workflow

if st.session\_state.deep\_config\_step == 7:

st.divider()

st.subheader("CSV Chunking Process")

st.sidebar.checkbox("Preprocessing Complete", value=True, disabled=True, key="deep\_chunking\_preprocessing\_complete")

# Check if data is available

if st.session\_state.deep\_df.empty:

st.error("❌ No data available for chunking. Please go back and load data first.")

if st.button("Back to Data Loading", key="deep\_back\_to\_data\_loading\_step7"):

st.session\_state.deep\_config\_step = 0

st.rerun()

st.stop()

st.subheader("Select Chunking Method")

chunking\_method = st.radio(

"Choose a chunking method:",

[

"Fixed Size",

"Recursive",

"Semantic",

"Document",

],

key="deep\_chunking\_method"

)

if chunking\_method == "Fixed Size":

st.info("Splits data into fixed-size chunks of characters with overlap")

chunk\_size = st.number\_input("Chunk Size (characters)", min\_value=50, max\_value=20000, value=400, step=50, key="deep\_fixed\_chunk\_size")

overlap = st.number\_input("Overlap (characters)", min\_value=0, max\_value=chunk\_size-1 if chunk\_size>0 else 0, value=50, key="deep\_fixed\_overlap")

elif chunking\_method == "Recursive":

st.info("Splits key-value formatted lines with recursive separators and overlap")

chunk\_size = st.number\_input("Chunk Size (characters)", min\_value=50, max\_value=20000, value=400, step=50, key="deep\_recursive\_chunk\_size")

overlap = st.number\_input("Overlap (characters)", min\_value=0, max\_value=chunk\_size-1 if chunk\_size>0 else 0, value=50, key="deep\_recursive\_overlap")

elif chunking\_method == "Semantic":

st.info("Clusters rows semantically and concatenates each cluster as a chunk")

n\_clusters = st.number\_input("Number of clusters", min\_value=2, max\_value=max(2, len(st.session\_state.deep\_df)), value=10, key="deep\_semantic\_clusters")

elif chunking\_method == "Document":

st.info("Group by a key column and split by token limit (headers optional)")

key\_column = st.selectbox("Key column", st.session\_state.deep\_df.columns.tolist(), key="deep\_document\_key\_column")

token\_limit = st.number\_input("Token limit per chunk", min\_value=200, max\_value=10000, value=2000, step=100, key="deep\_document\_token\_limit")

preserve\_headers = st.checkbox("Include headers in each chunk", value=True, key="deep\_document\_preserve\_headers")

if st.button("Apply Chunking Method", key="deep\_apply\_chunking"):

with st.spinner("🔄 Applying chunking via API..."):

try:

# Prepare chunking parameters based on method

if chunking\_method == "Fixed Size":

chunk\_params = {

"method": "fixed",

"chunk\_size": int(chunk\_size),

"overlap": int(overlap)

}

elif chunking\_method == "Recursive":

chunk\_params = {

"method": "recursive",

"chunk\_size": int(chunk\_size),

"overlap": int(overlap)

}

elif chunking\_method == "Semantic":

chunk\_params = {

"method": "semantic",

"n\_clusters": int(n\_clusters)

}

elif chunking\_method == "Document":

chunk\_params = {

"method": "document",

"key\_column": key\_column,

"token\_limit": int(token\_limit),

"preserve\_headers": preserve\_headers

}

result = call\_deep\_config\_chunk\_api(chunk\_params)

if "error" in result:

st.error(f"❌ Chunking failed: {result['error']}")

else:

st.success(f"✅ Successfully created {result.get('total\_chunks', 'N/A')} chunks!")

st.info(f"📊 \*\*Method\*\*: {chunking\_method}")

# Update session state with API results

st.session\_state.deep\_chunking\_result = {

"chunks": result.get('chunks', []),

"metadata": result.get('metadata', []),

"method": chunking\_method.lower().replace(" ", "\_"),

"total\_chunks": result.get('total\_chunks', 0)

}

# Add download button for chunks

st.markdown("---")

st.subheader("📥 Download Chunks")

if st.button("📄 Download Chunks CSV", key="deep\_download\_chunks"):

try:

chunks\_data = download\_deep\_config\_chunks()

filename = "chunks.csv"

st.download\_button(

label="⬇️ Download Chunks",

data=chunks\_data,

file\_name=filename,

mime="text/csv",

use\_container\_width=True

)

except Exception as e:

st.error(f"Download failed: {str(e)}")

st.session\_state.deep\_config\_step = 8

st.rerun()

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

# Back button

if st.button("Back to Metadata Selection", key="deep\_back\_to\_metadata"):

st.session\_state.deep\_chunking\_result = None

st.session\_state.deep\_embedding\_result = None

st.session\_state.deep\_config\_step = 6

st.rerun()

# Step 8: Embedding Generation

if st.session\_state.deep\_config\_step == 8:

st.sidebar.checkbox("Preprocessing Complete", value=True, disabled=True, key="deep\_embedding\_preprocessing\_complete")

st.sidebar.checkbox("Chunking Complete", value=True, disabled=True, key="deep\_embedding\_chunking\_complete")

st.subheader("Generate Embeddings")

available\_models = [

"all-MiniLM-L6-v2",

"paraphrase-MiniLM-L6-v2",

"text-embedding-ada-002"

]

model\_choice = st.radio(

"Choose an embedding model:",

available\_models,

key="deep\_embedding\_model"

)

col1, col2 = st.columns(2)

with col1:

st.metric("Model", model\_choice)

with col2:

st.metric("Type", "OpenAI" if "text-embedding" in model\_choice else "Local")

# Configuration options

st.subheader("Configuration")

batch\_size = st.selectbox("Batch Size", options=[32, 64, 128, 256], index=0, key="deep\_embedding\_batch\_size", help="Larger batch sizes are faster but use more memory")

use\_parallel = st.checkbox("Use parallel encoding (local models)", value=True, key="deep\_use\_parallel")

if "text-embedding" in model\_choice:

openai\_api\_key = st.text\_input("OpenAI API Key (optional)", type="password", key="deep\_openai\_api\_key")

openai\_base\_url = st.text\_input("OpenAI Base URL (optional)", value="", key="deep\_openai\_base\_url")

else:

openai\_api\_key = None

openai\_base\_url = None

if st.button("Generate Embeddings", key="deep\_generate\_embeddings"):

with st.spinner("🔄 Generating embeddings via API..."):

try:

# Prepare embedding parameters

embed\_params = {

"model\_name": model\_choice,

"batch\_size": int(batch\_size),

"use\_parallel": bool(use\_parallel)

}

# Add OpenAI parameters if using OpenAI model

if "text-embedding" in model\_choice:

if openai\_api\_key:

embed\_params["openai\_api\_key"] = openai\_api\_key

if openai\_base\_url:

embed\_params["openai\_base\_url"] = openai\_base\_url

result = call\_deep\_config\_embed\_api(embed\_params)

if "error" in result:

st.error(f"❌ Embedding generation failed: {result['error']}")

else:

st.success(f"✅ Successfully generated embeddings for {result.get('total\_chunks', 'N/A')} chunks!")

st.info(f"📊 \*\*Model\*\*: {model\_choice}")

st.info(f"📊 \*\*Vector Dimension\*\*: {result.get('vector\_dimension', 'N/A')}")

# Update session state with API results

st.session\_state.deep\_embedding\_result = {

'model\_used': model\_choice,

'total\_chunks': result.get('total\_chunks', 0),

'vector\_dimension': result.get('vector\_dimension', 0),

'embeddings': result.get('embeddings', []),

'chunk\_texts': result.get('chunk\_texts', []),

}

# Add download button for embeddings

st.markdown("---")

st.subheader("📥 Download Embeddings")

if st.button("📄 Download Embeddings JSON", key="deep\_download\_embeddings"):

try:

embeddings\_data = download\_deep\_config\_embeddings()

filename = "embeddings.json"

st.download\_button(

label="⬇️ Download Embeddings",

data=embeddings\_data,

file\_name=filename,

mime="application/json",

use\_container\_width=True

)

except Exception as e:

st.error(f"Download failed: {str(e)}")

st.session\_state.deep\_config\_step = 9

st.rerun()

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

# Back button

if st.button("Back to Chunking", key="deep\_back\_to\_chunking"):

st.session\_state.deep\_embedding\_result = None

st.session\_state.deep\_config\_step = 7

st.rerun()

# Step 9: Storage & Retrieval

if st.session\_state.deep\_config\_step == 9:

st.sidebar.checkbox("Preprocessing Complete", value=True, disabled=True, key="deep\_storage\_preprocessing\_complete")

st.sidebar.checkbox("Chunking Complete", value=True, disabled=True, key="deep\_storage\_chunking\_complete")

st.sidebar.checkbox("Embeddings Generated", value=True, disabled=True, key="deep\_storage\_embeddings\_generated")

st.subheader("Vector Storage & Retrieval")

storage\_choice = st.radio("Choose storage backend:", ["ChromaDB", "FAISS"], index=0, key="deep\_storage\_choice")

if storage\_choice == "ChromaDB":

default\_collection = f"csv\_chunks\_\_{uploaded\_file.name.replace('.csv','')}" if uploaded\_file else "csv\_chunks"

collection\_name = st.text\_input("Collection Name", value=default\_collection, key="deep\_collection\_name")

st.session\_state.collection\_name = collection\_name

else:

st.session\_state.collection\_name = "csv\_chunks"

# Retrieval metric selection (shown below storage selection)

retrieval\_metric = st.selectbox(

"Retrieval similarity metric",

["cosine", "dot", "euclidean"],

index=0,

key="deep\_retrieval\_metric"

)

# removed deep config captions per request (Config-1 only requested, keeping deep captions optional)

col\_a, col\_b = st.columns(2)

with col\_a:

if st.button("Store Embeddings", key="deep\_store\_embeddings"):

with st.spinner("🔄 Storing embeddings via API..."):

try:

if st.session\_state.deep\_embedding\_result is None:

st.error("No embeddings to store. Please generate embeddings first.")

else:

# Prepare storage parameters

storage\_type\_mapping = {

"ChromaDB": "chroma",

"FAISS": "faiss"

}

store\_params = {

"storage\_type": storage\_type\_mapping[storage\_choice],

"retrieval\_metric": retrieval\_metric

}

if storage\_choice == "ChromaDB":

store\_params["collection\_name"] = collection\_name

result = call\_deep\_config\_store\_api(store\_params)

if "error" in result:

st.error(f"❌ Storage failed: {result['error']}")

else:

if storage\_choice == "ChromaDB":

st.success(f"✅ Stored {result.get('total\_vectors', 'N/A')} vectors in ChromaDB collection '{collection\_name}'.")

else:

st.success(f"✅ Stored {result.get('total\_vectors', 'N/A')} vectors in FAISS index.")

st.info(f"📊 \*\*Storage Type\*\*: {storage\_choice}")

st.info(f"📊 \*\*Retrieval Metric\*\*: {retrieval\_metric}")

except Exception as e:

st.error(f"❌ API Error: {str(e)}")

with col\_b:

query = st.text\_input("Enter query to retrieve relevant chunks", value="", key="deep\_retrieval\_query")

top\_k = st.slider("Top K", min\_value=1, max\_value=50, value=5, step=1, key="deep\_top\_k")

if st.button("Search Vector DB", key="deep\_search\_vector\_db"):

try:

if not query:

st.warning("Please enter a query.")

elif st.session\_state.deep\_embedding\_result is None:

st.error("Embedding result not found. Generate embeddings first.")

else:

# Model used

er = st.session\_state.deep\_embedding\_result

model\_used = er.get('model\_used', 'unknown') or 'unknown'

# Build retriever based on storage choice

if storage\_choice == "ChromaDB":

collection\_name = st.session\_state.get('collection\_name', 'csv\_chunks')

try:

import chromadb

client = chromadb.PersistentClient(path="chromadb\_store")

collection = client.get\_collection(collection\_name)

# Generate query embedding

from sentence\_transformers import SentenceTransformer

model = SentenceTransformer("paraphrase-MiniLM-L6-v2")

query\_embedding = model.encode([query])

# Search

results = collection.query(

query\_embeddings=query\_embedding.tolist(),

n\_results=int(top\_k)

)

# Format results

docs = results.get('documents', [[]])[0] if results else []

metas = results.get('metadatas', [[]])[0] if results else []

dists = results.get('distances', [[]])[0] if results else []

except Exception as e:

st.error(f"ChromaDB search failed: {e}")

docs, metas, dists = [], [], []

else:

try:

import faiss

import pickle

# Load FAISS index and data

index = faiss.read\_index("faiss\_store/index.faiss")

with open("faiss\_store/data.pkl", "rb") as f:

faiss\_data = pickle.load(f)

# Generate query embedding

from sentence\_transformers import SentenceTransformer

model = SentenceTransformer("paraphrase-MiniLM-L6-v2")

query\_embedding = model.encode([query])

# Search

distances, indices = index.search(query\_embedding, int(top\_k))

# Format results

docs = [faiss\_data['documents'][i] for i in indices[0] if i < len(faiss\_data['documents'])]

metas = [faiss\_data['metadata'][i] for i in indices[0] if i < len(faiss\_data['metadata'])]

dists = distances[0].tolist()

except Exception as e:

st.error(f"FAISS search failed: {e}")

docs, metas, dists = [], [], []

st.success(f"Query completed successfully! (metric: {retrieval\_metric})")

st.subheader("Top Results")

if not docs:

st.info("No results found.")

else:

st.success(f"Found {len(docs)} results!")

for i, doc in enumerate(docs[: int(top\_k)]):

meta = metas[i] if i < len(metas) else {}

score = dists[i] if i < len(dists) else None

with st.expander(f"Result {i+1} — score: {score:.4f}", expanded=False):

st.write(f"\*\*Metadata:\*\* {meta}")

st.write("\*\*Full Content:\*\*")

st.text\_area(

f"Chunk {i+1} Content",

value=doc,

height=300,

key=f"deep\_chunk\_content\_{i}",

help="Scroll to view the complete chunk content"

)

except Exception as e:

st.error(f"Search failed: {e}")

# Back button

if st.button("Back to Embedding", key="deep\_back\_to\_embedding"):

st.session\_state.deep\_config\_step = 8

st.rerun()

# Complete process button

if st.button("Complete Deep Config Process", key="deep\_complete\_process"):

st.balloons()

st.success("🎉 Deep Config process completed successfully!")

st.info("You can now use the vector database for semantic search and retrieval.")

# Vector Retrieval Section with Scrollable Chunks

if st.session\_state.api\_results and st.session\_state.api\_results.get('summary', {}).get('retrieval\_ready'):

st.markdown("---")

st.markdown("## 🔍 Semantic Search (Vector DB)")

st.markdown("Search for similar content using semantic similarity")

col1, col2 = st.columns([3, 1])

with col1:

vector\_query = st.text\_input("Enter semantic search query:", placeholder="Search for similar content...", key="vector\_query")

with col2:

k = st.slider("Top K results", 1, 10, 3, key="vector\_k")

if vector\_query:

with st.spinner("Searching..."):

try:

st.session\_state.process\_status["retrieval"] = "running"

retrieval\_result = call\_retrieve\_api(vector\_query, k)

st.session\_state.process\_status["retrieval"] = "completed"

st.session\_state.retrieval\_results = retrieval\_result

if "error" in retrieval\_result:

st.error(f"Retrieval error: {retrieval\_result['error']}")

else:

st.success(f"✅ Found {len(retrieval\_result['results'])} results")

# Display each result with scrollable chunk content

for i, result in enumerate(retrieval\_result['results']):

display\_scrollable\_chunk(result, i)

except Exception as e:

st.error(f"Retrieval error: {str(e)}")

# Export Section

if st.session\_state.api\_results:

st.markdown("---")

st.markdown("## 💾 Export Results")

col1, col2 = st.columns(2)

with col1:

st.markdown("#### 📥 Download Chunks")

# Config-1: export as CSV; others: TXT

chunks\_btn\_label = "📄 Export Chunks as CSV" if st.session\_state.current\_mode == "config1" else "📄 Export Chunks as TXT"

if st.button(chunks\_btn\_label, use\_container\_width=True):

try:

chunks\_content = download\_file("/export/chunks", "chunks.csv" if st.session\_state.current\_mode == "config1" else "chunks.txt")

st.download\_button(

label="⬇️ Download Chunks",

data=chunks\_content,

file\_name=("chunks.csv" if st.session\_state.current\_mode == "config1" else "chunks.txt"),

mime=("text/csv" if st.session\_state.current\_mode == "config1" else "text/plain"),

use\_container\_width=True

)

except Exception as e:

st.error(f"Error exporting chunks: {str(e)}")

with col2:

st.markdown("#### 📥 Download Embeddings")

# Config-1: export as JSON; others: TXT

emb\_btn\_label = "🔢 Export Embeddings as JSON" if st.session\_state.current\_mode == "config1" else "🔢 Export Embeddings as TXT"

if st.button(emb\_btn\_label, use\_container\_width=True):

try:

embeddings\_content = download\_embeddings\_text()

st.download\_button(

label="⬇️ Download Embeddings",

data=embeddings\_content,

file\_name=("embeddings.json" if st.session\_state.current\_mode == "config1" else "embeddings.txt"),

mime=("application/json" if st.session\_state.current\_mode == "config1" else "text/plain"),

use\_container\_width=True

)

except Exception as e:

st.error(f"Error exporting embeddings: {str(e)}")

# Footer

st.markdown("---")

st.markdown("""

<div style="text-align: center; color: #666; font-size: 0.9em;">

<p>📦 Chunking Optimizer v2.0 • FastAPI + Streamlit • 3GB+ File Support • Performance Optimized</p>

<p><strong>🚀 Enhanced with Turbo Mode & Parallel Processing • 📜 Scrollable Chunk Display</strong></p>

</div>

""", unsafe\_allow\_html=True)