mcp_server

April 21, 2026

1 Restaurant Reservation Chatbot using MCP or A2A

In this assignment, you are given the skeleton code for a general restaurant reservation chatbot:

- **MCP**: Students who did not take ITCS 4101 will implement this as an MCP Server and MCP Host architecture.
- **A2A**: Students who took ITCS 4101 will implement this using the Agent2Agent(A2A) protocol. You will need to refactor the skeleton code below from MCP to A2A.

The restaurant chatbot engages the users in a conversation in order to determine their restaurant preferences. Once sufficient information is gathered from the user, including location, date, time, and other preferences, the charbot indetifies one or more restaurants that satisfy those constraints. Once the user selects a restaurant and confirms the reservation details, the chatbot makes a reservation.

The provided codebase follows an MCP architecture and is structured in two folders , as shown below:

chatbot_server/

`mcp_server.ipynb`

`mcp_server.py`

`data/`

`restaurants.json`

`reservations.csv`

`embeddings/`

chatbot_host/

`mcp_host.py`

`ChatbotTest.ipynb`

You will write code in two places:

1. The MCP server in `mcp_server.ipynb` (this notebook). Once you are done with the implementation of the MCP server:

- **Export** the notebook as source code into the `chatbot_server/mcp_server.py` file. To export the notebook, use the menu options *File –> Save and Export Notebook As –> Executable Script*.
- **Launch** the MCP server using the command `fastmcp run mcp_server.py:mcp --transport http --port 8000`

2. The MCP host in `mcp_host.py`.

Once you are done with both the MCP server and MCP host, open the `ChatbotTest.ipynb` notebook and **Verify** that the chatbot passes all the test cases. If some test cases of `ChatbotTest.ipynb` result in undesirable behavior, you may have to redo one or more of the steps in this order:

- **Implement** the MCP server in `chatbot_server/mcp_server.ipynb`.
- **Export** this notebook into source code file `chatbot_server/mcp_server.py` and launch the MCP server.
- **Implement** the MCP host in `chatbot_host/mcp_host.py`.
- **Verify** the chatbot by restarting the kernel and running each test case in `ChatbotTest.ipynb`.

1.1 MCP Server guidelines

You will need to **Implement** the following tools in this `mcp_server.ipynb` botebook, where you can reuse code from the previous assignment.

1. `create_vdb_for_location()`: Restricts the vector database search only over restaurants at a given location, by creating a new vector DB for restaurants at that location.
2. `get_restaurant_strs()`: Get restaurant data as structured text from the original JSON entries.
3. `vdb_search_restaurants()`: Searches vector database for restaurants that satisfy the user's current preferences.
4. `make_reservation()`: Makes a reservation at a restaurant, given restaurant information, date, and time.

Each tool function is followed by a set of test cases. Once your tools pass their test cases, have the MCP server expose them using the appropriate function **decorators**, and export the notebook as source code into `chatbot_server/mcp_server.py`.

1.2 MCP Host guidelines

You will need to **Implement** the following functionality in the `mcp_host.py` Python script:

1. Create the `mcp_client` object that connects to the MCP server that communicates through HTTP at 'http://localhost:8000/mcp'.
2. Implement steps 2 and 3 in the function `execute_function_call()` that takes function names issued by Gemini and calls the correspondign tools and returns their results.
3. Update the set of potential restaurants in the function `chat_turn()` by using `mcp_client` to call the appropriate tool exposed by the MCP server.

2 Create FastMCP server object

[ ]:
```python
from fastmcp import FastMCP

mcp = None # YOUR CODE HERE
```

2.1 Import necessary modules and setup Gemini client

[ ]:
```python
import pickle
from google import genai
import json
from typing import List, Dict, Any


import faiss, os

import numpy as np
from dotenv import load_dotenv, find_dotenv

# Create the Gemini API client.

from google.genai import types
import pandas as pd

_ = load_dotenv(find_dotenv())
client = genai.Client(api_key = os.environ['GEMINI_API_KEY'])
```

2.2 The Dialogue State

The chatbot needs to engage the user in a conversation with the aim of finding a restaurant that satisfies user's preferences in terms of location (City and State), Cuisine, Name (on the off chance the user knows exactly the restaurant they want), and Misc for any other preferences expressed by the user. To help the LLM track the user preferences, we instruct it to create a `DialogState` and update it at every turn in the conversation, as it receives more information from the user. The dialogue state is defined as:

```
dialog_state = {
            'City': '', 'State':'',
            "Cuisine": '',
            "Name": '',
            "Misc": ''
        }
```

Tracking and updating a dialogue states is especially useful when users change their mind and provide new preferences that conflict with the old ones. For example, if during a conversation you are asking for a kid friendly restaurant with a playground and then change your mind to try to make a reservation at a fancy restaurant alone, the model may not understand that the playground is not needed.

[ ]:
```python
# Define the dialog state structure
from pydantic import BaseModel, Field

class DialogState(BaseModel):
    """Pydantic model for tracking dialog state."""

    City: str = Field(default='-', description = "City name")
    State: str = Field(default='-', description = "State abbreviation (2␣
 ↪letters)")
    Cuisine: str = Field(default='-', description = "Cuisine type(s)")
    Name: str = Field(default='-', description = "Restaurant name(s)")
    Misc: str = Field(default='-', description = "Additional constraints")

    model_config = {'extra': 'forbid'}

    def update(self, **kwargs):
        """Update dialog state fields with new values."""
        for key, value in kwargs.items():
            if hasattr(self, key) and value:
                setattr(self, key, value)

    def get_state(self):
        """Return a dictionary copy of the current state."""
        return {
            'City': self.City,
            'State': self.State,
            'Cuisine': self.Cuisine,
            'Name': self.Name,
            'Misc': self.Misc
        }

    def to_json(self):
        """Convert dialog state to formatted JSON string."""
        return json.dumps(self.get_state(), indent=2)
```

2.3 The Vector Database

We use a vector database to store restaurant data as *embeddings*, namely one embedding vector for each restaurant. The restaurant data is stored in "data/restaurants.json" and was extracted as a subset of over 100K restaurants from the Yelp academic dataset, as described in "data/restaurants.pdf". The embeddings were then created using the Gemini API through the `client.models.embed_content()` function, indexed using the FAISS indexing function, and stored in "data/embeddings/full_vdb.faiss".

To understand the FAISS (Facebook AI Similarity Search) functionality, we recommend reading the Getting started guide.

```
[ ]: with open("data/embeddings/all_results_full.pkl",'rb') as f:
         vectors = pickle.load(f)

     # Load the full vector DB in 'vdb_for_all'.
     vdb_for_all = faiss.read_index("data/embeddings/full_vdb.faiss")
     restaurants = pd.read_json("data/restaurants.json")

     # Initialize a vector DB where to store indexed embeddings for restaurants at a␣
      ↪location.
     embedding_dimension = 3072
     vdb_for_location = faiss.IndexFlatL2(embedding_dimension)
     embdidx_to_idx = {}
```

3 Tool 1: Create a new vector DB for restaurants at a given location.

Given a user specified location as `City` and `State`, create a new vector DB (a FAISS index) containing only restaurants that match the specified city and state.

```
[ ]: # YOUR DECORATOR HERE
     def create_vdb_for_location(City, State):
         """
         Restrict vector database search to restaurants in a specific city and state.

         This function creates a new FAISS index containing only restaurants that
         match the specified city and state, enabling location-filtered searches.

         Parameters
         ----------
         City : str
             City name (must match exactly with restaurant data)
         State : str
             State abbreviation (2 letters, e.g., 'CA', 'NY')

         Returns
         -------
         dict
             count: Number of restaurants in the restricted index.
             status: Success string.

         Effects
         ------------
         Modifies global variables:
         - vdb_for_location : Creates new FAISS IndexFlatL2 (vector database) with␣
      ↪filtered restaurants
         - embdidx_to_idx : Updates mapping from embedding index to restaurant index
```

```
    Behavior
    -------
    1. Creates new FAISS index with same dimensions as full vector database
    2. Filters restaurants DataFrame for matching city and state
    3. Retrieves indices of matching restaurants
    4. Adds only matching restaurant vectors to new index
    5. Updates index mapping for translation between embedding and restaurant␣
↪indices

    Examples
    --------
    >>> count = create_vdb_for_location("San Francisco", "CA")
    >>> print(f"Restricted to {count} restaurants in San Francisco, CA")
    Restricted to 1247 restaurants in San Francisco, CA

    >>> # Now searches will only return SF restaurants
    >>> results = vdb_search_restaurants("Italian restaurants", k=10)

    Notes
    -----
    - Uses L2 (Euclidean) distance metric
    - Requires global variables: vectors, restaurants, embdidx_to_idx
    - Case-sensitive matching on city and state
    """

    global vdb_for_location
    global embdidx_to_idx

    # YOUR CODE HERE









    total_restaurants_in_vdb = vdb_for_location.ntotal
    return {'count': total_restaurants_in_vdb, "status": "success"}
```

3.0.1 Query Embedding

Wrapper for Gemini API call to create an embedding for an input text query.

[ ]:
```python
def embd_txt(text: str) -> np.ndarray:
    """
    Generate embedding vector for text using Gemini embedding model.

    This function converts text into a dense vector representation optimized
    for semantic search and retrieval tasks.

    Parameters
    ----------
    text : str
        Text to embed (query, description, etc.)

    Returns
    -------
    np.ndarray
        Numpy array containing embedding vector of shape (1, embedding_dim)

    Model Details
    -------------
    - Model: gemini-embedding-001
    - Task type: RETRIEVAL_QUERY (optimized for search queries)

    Purpose
    -------
    Creates vector representations for:
    - User search queries
    - Restaurant descriptions
    - Any text that needs semantic comparison

    Examples
    --------
    >>> embedding = _embd_txt("Italian restaurants with outdoor seating")
    >>> print(embedding.shape)
    (1, (Embedding Dimension))

    >>> # Use for similarity search
    >>> query_vector = _embd_txt("pizza places")
    >>> distances, indices = vdb.search(query_vector, k=10)

    """
    result = client.models.embed_content(
        model = "gemini-embedding-001",
        contents = text,
        config = types.EmbedContentConfig(task_type = "RETRIEVAL_QUERY"))
```

```
    return np.array([result.embeddings[0].values])
```

Test cases for Create Vector DB for Location

[ ]:
```
if __name__ == '__main__':
    print(create_vdb_for_location('Philadelphia', 'PA')) # expected : 5861
    print(create_vdb_for_location('Nashville', 'TN')) # expected : 2507
    print(create_vdb_for_location('Audubon', 'PA')) # expected : 21
    print(create_vdb_for_location('Audubon', 'NJ')) # expected : 30
```

```
5861
2507
21
30
```

Showcasing the k-nearest neighbor search for the new vector DB

[ ]:
```
if __name__ == '__main__':
    count = create_vdb_for_location('Lahaska', 'PA')
    print(count) # expected 1

    embded_query = embd_txt(text = "Name : Hart's Tavern")
    distances, indices = vdb_for_location.search(embded_query, k = 1)
    print(restaurants.iloc[embdidx_to_idx.get(indices[0][0])])
```

```
1
business_id                                          PiQVIJI92Z9O37jg9YMyPw
name                                                          Hart's Tavern
address                                                     Rt 202 & Rt 263
city                                                                Lahaska
state                                                                    PA
postal_code                                                           18931
latitude                                                          40.347996
longitude                                                        -75.03258
stars_x                                                                2.5
review_count                                                           162
is_open                                                                  1
attributes             {'GoodForKids': 'True', 'BusinessParking': '{'…
categories                                Restaurants, American (Traditional)
hours                  {'Monday': '11:0-20:30', 'Tuesday': '11:0-20:3…
review_texts           [NO STARS! They used to make pretty good soup …
stars_y                                      [1, 5, 1, 5, 5, 1, 3, 1, 2, 2]
num_sampled_reviews                                                     10
Name: 19470, dtype: object
```

4 Tool 2: Get restaurant data as structured text from original JSON

Given a list of restaurant vDB indexes, find their JSON entries and translate each entry into a string that can later be mapped to an embedding.

[ ]:
```python
# YOUR DECORATOR HERE
def get_restaurant_strs(restaurant_indexes: list):
    """
    Format restaurant information as a string for display or processing.

    Retrieves restaurant data from the DataFrame and formats it into a
    structured string containing key information.

    Parameters
    ----------
    restaurant_indexes : list
        List of indexes of restaurants in the vector database
        - If using restricted VDB: index in embedding space
        - If using full VDB: direct restaurant index

    Returns
    -------
    list of formatted strings, one string per restaurant, where each formatted␣
 ↪string is created by calling `get_restaurant_str()` below
    """

    rstrings = [] # YOUR CODE HERE

    return {'status': 'Successful', 'restaurant_strs': rstrings}



def get_restaurant_str(restaurant_index: int):
    """
    Format restaurant information as a string for display or processing.

    Retrieves restaurant data from the DataFrame and formats it into a
    structured string containing key information.

    Parameters
    ----------
    restaurant_index : int
        Index of restaurant in the database
        - If using restricted VDB: index in embedding space
        - If using full VDB: direct restaurant index
```

```
    Returns
    -------
    str
        Formatted string with restaurant details in the format:
        "RestaurantId: {id} Name: {name} Location: {city} {state} other:␣
↪{attributes} {categories} {hours}"

    Example
    --------
    >>> info = get_restaurant_str(42)
    >>> print(info)
    RestaurantId: abc123 Name: Tony's Pizza Location: Boston MA other:␣
↪{'parking': True} Italian, Pizza Mon-Fri: 11:00-22:00

    """

    global vdb_for_location

    # YOUR CODE HERE




    return f""
```

4.1 Test case for get restaurant data as text tool

[ ]:
```
if __name__ == '__main__':
    create_vdb_for_location('Audubon', 'NJ')

    restaurants_test = [1,7 ,28 ,4]

    '''
    Expected output from get_restaurant_strs() below:
```

```
   ['RestaurantId: oVfTDLWObg4VMXlSI-tpTA Name: Smoke BBQ Location: Audubon NJ␣
↪other: {\'RestaurantsTableService\': \'False\', \'RestaurantsAttire\':␣
↪"u\'casual\'", \'DogsAllowed\': \'False\', \'NoiseLevel\': "u\'average\'",␣
↪\'RestaurantsGoodForGroups\': \'True\', \'BikeParking\': \'True\',␣
↪\'BusinessParking\': "{\'garage\': False, \'street\': True, \'validated\':␣
↪False, \'lot\': False, \'valet\': False}", \'WheelchairAccessible\':␣
↪\'True\', \'HappyHour\': \'False\', \'WiFi\': "u\'no\'",␣
↪\'RestaurantsPriceRange2\': \'2\', \'ByAppointmentOnly\': \'False\',␣
↪\'Alcohol\': "\'none\'", \'BYOB\': \'True\', \'HasTV\': \'False\',␣
↪\'OutdoorSeating\': \'True\', \'BusinessAcceptsBitcoin\': \'False\',␣
↪\'Caters\': \'True\', \'GoodForMeal\': "{\'dessert\': False, \'latenight\':␣
↪False, \'lunch\': True, \'dinner\': True, \'brunch\': False, \'breakfast\':␣
↪False}", \'RestaurantsReservations\': \'False\',␣
↪\'BusinessAcceptsCreditCards\': \'True\', \'Ambience\': "{\'touristy\':␣
↪False, \'hipster\': False, \'romantic\': False, \'divey\': False,␣
↪\'intimate\': False, \'trendy\': None, \'upscale\': False, \'classy\':␣
↪False, \'casual\': True}", \'RestaurantsDelivery\': \'False\',␣
↪\'GoodForKids\': \'True\', \'RestaurantsTakeOut\': \'True\', \'Corkage\':␣
↪\'False\'} Barbeque, Restaurants {\'Monday\': \'0:0-0:0\', \'Thursday\':␣
↪\'12:0-18:0\', \'Friday\': \'12:0-18:0\', \'Saturday\': \'12:0-18:0\',␣
↪\'Sunday\': \'12:0-17:0\'}',
    "RestaurantId: z494JUd3IGXgilROotL-8A Name: Futomaki Location: Audubon NJ␣
↪other: {'RestaurantsReservations': 'True', 'RestaurantsTakeOut': 'None',␣
↪'RestaurantsDelivery': 'True'} Japanese, Asian Fusion, Restaurants, Sushi␣
↪Bars None",
    'RestaurantId: gHrJTTvCsCPOqeKILVktxQ Name: Simply Soups & A Little More␣
↪Location: Audubon NJ other: {\'RestaurantsTakeOut\': \'True\',␣
↪\'RestaurantsAttire\': "u\'casual\'", \'NoiseLevel\': "u\'average\'",␣
↪\'RestaurantsGoodForGroups\': \'False\', \'BusinessAcceptsCreditCards\':␣
↪\'False\', \'BikeParking\': \'True\', \'WiFi\': "u\'no\'", \'Caters\':␣
↪\'True\', \'OutdoorSeating\': \'False\', \'GoodForKids\': \'True\',␣
↪\'RestaurantsReservations\': \'False\', \'RestaurantsPriceRange2\': \'1\',␣
↪\'RestaurantsDelivery\': \'True\', \'Alcohol\': "u\'none\'", \'HasTV\':␣
↪\'True\', \'BusinessParking\': "{\'garage\': False, \'street\': True,␣
↪\'validated\': False, \'lot\': False, \'valet\': False}", \'GoodForMeal\':␣
↪\'None\', \'Ambience\': \'None\'} Desserts, Comfort Food, Soup, Food,␣
↪Restaurants, Sandwiches {\'Monday\': \'10:0-19:0\', \'Tuesday\': \'10:0-19:
↪0\', \'Wednesday\': \'10:0-19:0\', \'Thursday\': \'10:0-19:0\', \'Friday\':␣
↪\'10:0-19:0\', \'Saturday\': \'11:0-16:0\'}',
    'RestaurantId: nx8fPYluWlV-XU3vMOZYlw Name: Desserts by Design Location:␣
↪Audubon NJ other: {\'BusinessParking\': "{\'garage\': False, \'street\':␣
↪True, \'validated\': False, \'lot\': False, \'valet\': False}",␣
↪\'RestaurantsPriceRange2\': \'1\', \'BusinessAcceptsCreditCards\': \'True\',␣
↪\'BikeParking\': \'True\', \'RestaurantsDelivery\': \'False\',␣
↪\'RestaurantsTakeOut\': \'False\'} Restaurants, Food, Bakeries {\'Tuesday\':␣
↪\'8:0-18:0\', \'Wednesday\': \'8:0-18:0\', \'Thursday\': \'8:0-18:0\',␣
↪\'Friday\': \'8:0-18:0\', \'Saturday\': \'8:0-17:0\'}']
```

```
    '''
    get_restaurant_strs(restaurants_test) ['restaurant_strs']
```

5 Tool 3: Search for restaurants in vector DB that match user preferences

Given the `dialogue_state` and a `query` string that is assembled by the LLM to capture the user preferences expressed in the conversation so far, this tool returns a list of potential restaurants that match the user preferences. For efficiency reasons, this proceeds in 2 steps:

- First, a vector DB search is done for the top-K restaurants whose embeddings match the umbedding of the current user preferences in the `query`.
- Second, the LLM is used to verify that each of the top-K restaurants truly satisfies the preferences in `dialog_state`. To make it fast and cost effective, the LLM splits the top-K restaurants into batches, and verifies all the restaurants in a batch within one API call in the function `check_restaurants_batch()`.

Before we implement this tool, we will implement two auxiliary functions:

1. `check_restaurants_batch()` gets a list of restaurants and uses an LLM to verify which of them satisfy the user prerefences.
2. `update_potential_restaurants()` splits a long list of potential restaurants into batches and calls `check_restaurants_batch()` on each batch.

5.1 Check which restaurants from batch satisfy user preferences

Use the LLM to verify which of the top-K restaurants truly satisfies the current user preferences. Verify all the restaurants in the input batch within one API call in the function `check_restaurants_batch()`.

[ ]:
```
def check_restaurants_batch(dialog_state: DialogState, restaurant_indices:␣
↪list) -> list[bool]:
    """
    Check multiple restaurants against dialog state constraints in a single API␣
↪call.

    This function uses the Gemini AI model to evaluate whether each restaurant
    in a batch matches the user's constraints, enabling efficient batch␣
↪filtering.

    Parameters
    ----------
    dialog_state : DialogState
        User constraints and preferences
    restaurant_indices : list
        List of restaurant indices to check
```

```
    Returns
    -------
    list[bool]
        Boolean list where True = keep restaurant, False = remove restaurant
        Length matches len(restaurant_indices)

    Behavior
    -------
    1. Builds numbered list of restaurant details
    2. Constructs prompt with constraints and restaurants
    3. Calls gemini-2.5-flash-lite model
    4. Parses comma-separated response (e.g., "1,3,5,7")
    5. Converts to boolean list

    Error Handling
    --------------
    - If API call fails: Returns all True (fail-safe, keeps all)
    - If response is "none": Returns all False
    - If response is empty: Returns all False
    - Invalid numbers in response: Skipped

    Examples
    --------
    >>> ds = DialogState()
    >>> ds.update(Cuisine="Italian", Misc="outdoor seating")
    >>> indices = [10, 25, 33, 47, 52]
    >>> results = _check_restaurants_batch(ds, indices)
    >>> print(results)
    [True, False, True, False, True]  # Restaurants 10, 33, 52 match

    >>> # Filter restaurants
    >>> kept = [idx for idx, keep in zip(indices, results) if keep]
    >>> print(kept)
    [10, 33, 52]

    """

    # YOUR CODE HERE
    # Build the prompt with instructions and data about all restaurants in the␣
↪batch.
```

```
    # Get the response from the LLM (try 'gemini-2.5-flash-lite' as it is␣
↪faster for filtering), extract list of Booleans from it and return it.
```

5.2 Update Potential Restaurants

Given a current list of `potential_restaurants`, filter out those that do not satisfy the user preferences expressed in the `dialog_state`. Do this efficiently by splitting the potential restaurants into one or more batches, and verifying the restaurant in each batch using `check_restaurants_batch()`.

[ ]:
```
def update_potential_restaurants(dialog_state: DialogState ,␣
↪potential_restaurants):
    """
    Filter potential restaurants based on dialog state constraints using batch␣
↪processing.

    This function refines the list of potential restaurants by checking each one
    against the current dialog state constraints. It uses batch processing for
    efficiency, checking up to 50 restaurants per API call.

    Parameters
    ----------
    dialog_state : DialogState
        Current user preferences and constraints (City, State, Cuisine, Name,␣
↪Misc)
    potential_restaurants : set
        Set of restaurant indices to filter

    Returns
    -------
    set
        Updated set of restaurant indices that match all constraints

    Behavior
    ---------
    1. Adds unrefined indexes to potential restaurants
    2. Converts set to list for batch processing
    3. Processes in batches of 50:
       a. Call _check_restaurants_batch() for each batch
```

```
    b. Mark restaurants that don't match for removal
4. Removes non-matching restaurants from set
5. Returns filtered set

Batch Processing
----------------
- Batch size: 50 restaurants
- Uses gemini-2.5-flash-lite for fast filtering
- Collects all removals before modifying set

Examples
--------
>>> ds = DialogState()
>>> ds.update(City="Portland", State="OR", Cuisine="Thai")
>>> potential = {0, 5, 12, 18, 23, 45}
>>> filtered = update_potential_restaurants(ds, potential)
>>> print(filtered)
{5, 18, 23}  # Only Thai restaurants in Portland, OR

Performance
-----------
- Processes 50 restaurants per API call
- For 500 restaurants: ~10 API calls
- Typical processing time: 5-15 seconds
"""
restaurant_list = list(potential_restaurants)

batch_size = 50
restaurants_to_remove = set()

for i in range(0,len(restaurant_list), batch_size):
    batch = restaurant_list[i:i + batch_size]
    batch_res = check_restaurants_batch(dialog_state, batch)

    for idx, keep in zip(batch, batch_res):
        if not keep:
            restaurants_to_remove.add(idx)

potential_restaurants -= restaurants_to_remove

return potential_restaurants
```

5.3 Tool 3 implementation

Now that we have the auxiliary functions ready, we can implement the search tool.

```
[ ]: # YOUR DECORATOR HERE
def vdb_search_restaurants(dialog_state, query: str, k: int = 250) -> dict:
    """
    Search vector database for restaurants matching user preferences expressed␣
 ↪in the query and the dialog_state.

    This function performs semantic search on the restaurant database using
    vector embeddings. It adds matching restaurant indices to the unrefined
    set for filtering.

    Parameters
    ----------
    query : str
        Search query string containing information from dialog state
        Examples: "Italian restaurants", "seafood Boston", "pizza outdoor␣
 ↪seating"
    k : int, optional
        Number of top results to return (default: 250)

    Returns
    -------
    str
        Status message: "Successful"

    Behavior
    --------------
    1. Embeds query text using _embd_txt()
    2. Determines which vector database to search
    3. Performs FAISS similarity search with k neighbors
    5. Returns success message, and potential restaurants

    Examples
    --------
    >>> # Simple search
    >>> result = vdb_search_restaurants("Thai restaurants")
    >>> print(result)
    'Successful'

    >>> # After restricting by location
    >>> create_vdb_for_location("Austin", "TX")
    >>> result = vdb_search_restaurants("BBQ", k=50)
    >>> # Now only searches Austin restaurants

    >>> # With more specific query
    >>> result = vdb_search_restaurants("vegetarian Indian buffet", k=100)
    """
```

```
    matching_restaurants = set([])

    # YOUR CODE HERE









    potential_restaurants = update_potential_restaurants(dialog_state,␣
↪matching_restaurants)

    return {"status": 'Successful' , "restaurants": list(potential_restaurants)}
```

5.4 Test case for vDB search restaurants tool

[ ]:
```
if __name__ == '__main__':
    create_vdb_for_location('Audubon', 'PA')

    ds = DialogState()
    ds.update(Cuisine = 'Italian_food')
    potential_restaurants = vdb_search_restaurants(ds, "Italian Food", k =␣
↪20)['restaurants']

    for i in list(potential_restaurants):
        idx = embdidx_to_idx.get(i)
        print(restaurants.iloc[idx]['name'])
```

```
Tony's Pizzeria & Italian Restaurant
Corropolese Italian Bakery
```

6 Tool 4: Make restaurant reservation (5 points)

Makes a reservation at a restaurant, given restaurant ID, date, and time. Appends the reservation information to the "reservations.csv" file.

[ ]:
```
# YOUR DECORATOR HERE
def make_reservation(restaurant_id: str, date: str, time: str) -> dict:
    """
    Create and save a restaurant reservation to CSV file.

    This function validates reservation details and appends them to a CSV file
```

```
    for record-keeping. It performs comprehensive validation on all inputs.

    Parameters
    ----------
    restaurant_id : str
        Business ID of the restaurant (must exist in restaurants DataFrame)
    date : str
        Reservation date in YYYY-MM-DD format (e.g., "2025-10-15")
        Must be today or a future date
    time : str
        Reservation time in HH:MM format using 24-hour notation
        Examples: "19:30" for 7:30 PM, "12:00" for noon

    Returns
    -------
    dict
        {"status": "success"} if reservation successful.
        {"status": "fail"} if reservation failed.

    Checks
    ------------------
    1. Restaurant Validation:
       - restaurant_id must exist in restaurants['business_id']
       - Returns False if not found

    2. Date Validation:
       - Must be in YYYY-MM-DD format
       - Must be today or future date (not past)
       - Returns False if invalid format or past date

    3. Time Validation:
       - Must be in HH:MM format (24-hour)
       - Returns False if invalid format
       - Does not validate against restaurant hours

    4. File Operations:
       - Must successfully write to CSV
       - Returns False if write fails

    Output File
    -----------
    - File: 'reservations.csv'
    - Location: Current working directory
    - Format: CSV with headers
    - Columns: restaurant_id, date, time, timestamp

    Reservation Record
```

```
    ------------------
    Each reservation contains:
    - restaurant_id: Business ID
    - date: Reservation date (YYYY-MM-DD)
    - time: Reservation time (HH:MM)
    - timestamp: When reservation was made (YYYY-MM-DD HH:MM:SS)

    File Format
    -----------
    reservations.csv:
        restaurant_id,date,time,timestamp
        abc123,2025-12-25,19:30,2025-10-10 14:30:45
        def456,2025-12-31,20:00,2025-10-10 14:35:12

    Effects
    ------------
    - Creates 'reservations.csv' if it doesn't exist
    - Appends new reservation to existing file
    - Prints status/error messages to console

    Error Messages
    --------------
    Printed to console:
    - "Error: Restaurant ID {restaurant_id} not found"
    - "Error: Date {date} is in the past"
    - "Error: Invalid date format {date}. Use YYYY-MM-DD"
    - "Error: Invalid time format {time}. Use HH:MM"
    - "Error writing reservation: {exception}"
    - "Reservation successfully made for {date} at {time}"

    """
    from datetime import datetime
    import os
    import csv

    # YOUR CODE HERE
```

6.1 Test case for make reservation tool

[ ]:
```
if __name__ == '__main__':
    # This reservation should go through.
    make_reservation(restaurant_id =␣
↪'PiQVIJI92Z9O37jg9YMyPw',date="2025-12-01",time = "10:10")

    # These next reservations should fail.

    # Invalid restaurant ID.
    make_reservation(restaurant_id = 'PiQVIJI92Z97jg9YMyPw',date="2025-12-01",␣
↪time = "10:10")
    # Date in the past.
    make_reservation(restaurant_id =␣
↪'PiQVIJI92Z9O37jg9YMyPw',date="2021-12-01", time = "10:10")
    # Incorrect timestamp format.
    make_reservation(restaurant_id =␣
↪'PiQVIJI92Z9O37jg9YMyPw',date="2025-12-01", time = "02:00 PM")
```

```
Reservation successfully made for 2025-12-01 at 10:10
Error: Restaurant ID PiQVIJI92Z97jg9YMyPw not found
Error: Date 2021-12-01 is in the past
Error: Invalid time format 02:00 PM. Use HH:MM
```