```python
# CafeGenius: Intelligent Cafe Assistant Demo
#
# This notebook demonstrates a conversational AI assistant for a cafe,
# using Gemini for language understanding and function calling, and
# ChromaDB for RAG-based menu search.

```


```python
# ============================================================================
# 1. Setup and Installation
# ============================================================================
print("Installing necessary packages...")
# Use %pip for better integration in Jupyter/Colab
# Note: Adjust versions as needed based on compatibility and latest releases
# Using versions known to be relatively stable at the time of writing (example)
%pip install --upgrade --quiet "google-generativeai>=0.5.0" "google-ai-
generativelanguage>=0.6.0" "chromadb>=0.5.0" "google-api-python-client" "google-
auth" "kaggle"

# (Optional) Uninstall conflicting packages if necessary
# %pip uninstall -y qqx jupyterlab kfp

print("Package installation complete.")
```


```python
# ============================================================================
# 2. Imports
# ============================================================================
import os
import json
import logging
from typing import List, Dict, Any, Optional, Union

# Google Generative AI
import google.generativeai as genai
import google.ai.generativelanguage as glm # Updated import for types
from google.api_core import retry
from google.protobuf import struct_pb2 # For function call responses

# ChromaDB
import chromadb
from chromadb import Documents, EmbeddingFunction, Embeddings

# Other Utilities
from IPython.display import Markdown, display
from kaggle_secrets import UserSecretsClient # Or use python-dotenv for
local .env files

print("Imports complete.")
```


```python
# ============================================================================
# 3. Logging Configuration
# ============================================================================
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
```

```
)

logger = logging.getLogger('cafe_genius')
logger.info("Logging setup complete.")
```

```python
# ================================================================================
# 4. API Key Configuration
# ================================================================================
try:
    # Prioritize Kaggle secrets if available
    GOOGLE_API_KEY = UserSecretsClient().get_secret("GOOGLE_API_KEY")
    logger.info("Using GOOGLE_API_KEY from Kaggle secrets.")

except Exception as e:
    logger.warning(f"Could not get API key from Kaggle secrets: {e}. Trying
environment variable.")
    # Fall back to environment variable for local development
    GOOGLE_API_KEY = os.environ.get("GOOGLE_API_KEY")
    if GOOGLE_API_KEY:
        logger.info("Using GOOGLE_API_KEY from environment variable.")

if not GOOGLE_API_KEY:
    logger.error("API key not found. Set GOOGLE_API_KEY Kaggle secret or
environment variable.")
    # Optional: Provide instructions or raise error
    print("ERROR: GOOGLE_API_KEY not found. Please set it as a Kaggle secret or
environment variable.")
    # raise ValueError("API key not found.") # Uncomment to halt execution if
key is missing
else:
    # Configure the GenAI client
    genai.configure(api_key=GOOGLE_API_KEY)
    logger.info("Google Generative AI client configured.")

    try:
        logger.info(f"Using google-generativeai version: {genai.__version__}")
    except AttributeError:
        logger.warning("Could not determine google-generativeai version.")
```

```python
# ================================================================================
# 5. Gemini Embedding Function for ChromaDB (Retry Logic)
# ================================================================================
# Import necessary exception types for retry logic
from google.api_core import exceptions as api_core_exceptions
from google.api_core import retry
from chromadb import Documents, EmbeddingFunction, Embeddings # Ensure these are
imported
from typing import Optional, List # Ensure these are imported

class GeminiEmbeddingFunction(EmbeddingFunction):
    """Custom embedding function using Gemini API (text-embedding-004)"""
    def __init__(self, api_key: Optional[str] = None,
task_type="retrieval_document", model_name="models/text-embedding-004"):
        # If api_key is provided, configure a temporary client (useful if global
config isn't set yet)
        if api_key:
            genai.configure(api_key=api_key)
```

```
        self.task_type = task_type
        self.model_name = model_name
        logger.info(f"GeminiEmbeddingFunction initialized with model:
{self.model_name}")

    # Retry on common transient API errors and connection issues
    @retry.Retry(predicate=retry.if_exception_type(
        api_core_exceptions.Aborted,
        api_core_exceptions.DeadlineExceeded,
        api_core_exceptions.ServiceUnavailable,
        api_core_exceptions.InternalServerError,
        ConnectionError
    ))
    def __call__(self, input: Documents) -> Embeddings:
        """Embeds a list of documents."""
        if not input:
            return []
        # Determine correct task type based on ChromaDB's usage hint if
available
        # This part might need adjustment based on how ChromaDB passes context
        current_task_type = self.task_type # Default unless overridden by
context

        logger.info(f"Embedding {len(input)} documents with task type:
{current_task_type}")
        try:
            # Ensure input is a list of strings
            if not isinstance(input, list) or not all(isinstance(doc, str) for
doc in input):
                raise TypeError("Input must be a list of strings (Documents).")

            response = genai.embed_content(
                model=self.model_name,
                content=input,
                task_type=current_task_type # Use determined task type
            )
            # Ensure embeddings are returned correctly
            if 'embedding' in response and isinstance(response['embedding'],
list):
                 # Check if it's a list of lists or just a list (for single
input case, though __call__ expects list)
                 embeddings = response['embedding']
                 if embeddings and not isinstance(embeddings[0], list):
                      # API might return a single list for a single input
document
                      # The ChromaDB interface likely expects a list of lists
                      logger.info(f"Successfully embedded {len(input)} documents
(single list wrapper).")
                      return [embeddings]
                 else:
                      logger.info(f"Successfully embedded {len(embeddings)}
documents.")
                      return embeddings

            # Adapt based on actual response structure if different
            elif hasattr(response, 'embedding') and
isinstance(response.embedding, list):
                 logger.info(f"Successfully embedded {len(response.embedding)}
documents (object access).")
                 # Assuming response.embedding contains Embedding objects with a
'values' attribute
                 return [list(e.values) for e in response.embedding if
hasattr(e, 'values')]
```

```
            else:
                 logger.error(f"Unexpected embedding response format:
{response}")
                 raise ValueError("Failed to extract embeddings from response.")

        except TypeError as te: # Catch specific TypeError from input validation
             logger.error(f"Input type error during embedding: {te}")
             raise # Re-raise the type error
        except Exception as e:
            logger.exception(f"Error during embedding call: {e}")
            # Propagate the error or return empty list based on desired
robustness
            raise # Re-raise the exception after logging
```


```python
# ================================================================================
# 6. Menu Data Definition - CafeGenius Full Menu
# ================================================================================

menu_data = [
    # Coffee
    {"name": "Espresso", "category": "Coffee", "description": "A concentrated
form of coffee served in small, strong shots.", "price": 3.50, "available":
True, "modifiers": ["Single", "Double", "Triple", "Decaf", "Regular"]},
    {"name": "Americano", "category": "Coffee", "description": "Espresso diluted
with hot water for a smoother taste.", "price": 3.75, "available": True,
"modifiers": ["Hot", "Iced", "Decaf"]},
    {"name": "Latte", "category": "Coffee", "description": "Espresso with
steamed milk and a light layer of foam.", "price": 4.50, "available": True,
"modifiers": ["Whole Milk", "2%", "Skim", "Oat Milk", "Almond", "Vanilla",
"Caramel", "Hazelnut"]},
    {"name": "Cappuccino", "category": "Coffee", "description": "Equal parts
espresso, steamed milk, and foam.", "price": 4.25, "available": True,
"modifiers": ["Whole Milk", "2%", "Oat Milk", "Almond", "Cinnamon",
"Chocolate"]},
    {"name": "Flat White", "category": "Coffee", "description": "Ristretto shots
topped with steamed milk and a velvety microfoam.", "price": 4.25, "available":
True, "modifiers": ["Whole Milk", "Oat Milk", "Almond"]},
    {"name": "Cold Brew", "category": "Coffee", "description": "Coffee brewed
with cold water over an extended period.", "price": 4.75, "available": True,
"modifiers": ["Sweet Cream", "Vanilla", "Caramel", "Oat Milk"]},
    {"name": "Mocha", "category": "Coffee", "description": "Espresso with
chocolate syrup and steamed milk, topped with whipped cream.", "price": 4.95,
"available": True, "modifiers": ["Dark Chocolate", "White Chocolate", "Oat
Milk", "Almond Milk", "No Whip"]},

    # Tea
    {"name": "Chai Latte", "category": "Tea", "description": "Spiced tea
concentrate with steamed milk.", "price": 4.50, "available": True, "modifiers":
["Whole Milk", "2%", "Oat Milk", "Almond", "Extra Spicy"]},
    {"name": "Matcha Latte", "category": "Tea", "description": "Japanese green
tea powder with steamed milk.", "price": 5.00, "available": True, "modifiers":
["Whole Milk", "2%", "Oat Milk", "Almond", "Vanilla"]},
    {"name": "Earl Grey", "category": "Tea", "description": "Classic black tea
infused with bergamot citrus.", "price": 3.25, "available": True, "modifiers":
["Honey", "Lemon", "Milk"]},
    {"name": "Herbal Tea", "category": "Tea", "description": "Caffeine-free
blend of herbs and botanicals.", "price": 3.00, "available": True, "modifiers":
["Peppermint", "Chamomile", "Lemon Ginger"]},
    {"name": "Iced Tea", "category": "Tea", "description": "Refreshing black tea
served over ice.", "price": 3.50, "available": True, "modifiers": ["Sweetened",
"Unsweetened", "Lemon", "Peach", "Raspberry"]},
```

```
    # Pastries
    {"name": "Croissant", "category": "Pastry", "description": "Buttery, flaky
pastry of French origin.", "price": 3.25, "available": True, "modifiers":
["Butter", "Almond", "Chocolate"]},
    {"name": "Blueberry Muffin", "category": "Pastry", "description": "Sweet
breakfast bread with blueberries.", "price": 3.50, "available": True,
"modifiers": ["Warmed"]},
    {"name": "Banana Bread", "category": "Pastry", "description": "Moist banana
bread with a hint of cinnamon.", "price": 3.75, "available": True, "modifiers":
["Warmed", "Add Butter"]},
    {"name": "Cinnamon Roll", "category": "Pastry", "description": "Swirled
pastry with cinnamon and icing.", "price": 4.00, "available": True, "modifiers":
["Extra Icing", "Warmed"]},

    # Food
    {"name": "Avocado Toast", "category": "Food", "description": "Toasted bread
topped with mashed avocado.", "price": 7.50, "available": True, "modifiers":
["Add Egg", "Add Tomato", "Add Feta"]},
    {"name": "Breakfast Sandwich", "category": "Food", "description": "Egg and
cheese on a croissant or English muffin.", "price": 6.50, "available": True,
"modifiers": ["Bacon", "Sausage", "Avocado"]},
    {"name": "Quiche", "category": "Food", "description": "Savory egg tart with
cheese and seasonal vegetables.", "price": 6.75, "available": True, "modifiers":
["Vegetarian", "Add Ham", "Gluten-Free"]},
    {"name": "Yogurt Parfait", "category": "Food", "description": "Layers of
Greek yogurt, granola, and seasonal fruit.", "price": 5.50, "available": True,
"modifiers": ["Honey", "No Granola", "Add Chia Seeds"]},
    # Add more items as needed
]


logger.info(f"Loaded {len(menu_data)} menu items.")

# Create detailed descriptions for RAG
menu_descriptions = []
for item in menu_data:
    description = f"{item['name']}: {item['description']} Priced at $
{item['price']:.2f}."
    if item.get('modifiers'): # Use .get for safety
        description += f" Available modifiers: {', '.join(item['modifiers'])}."
    menu_descriptions.append(description)


logger.info(f"Generated {len(menu_descriptions)} descriptions for RAG.")
### LOGGING - Enable/ Disable
print("Sample menu descriptions for RAG:")
for i in range(min(3, len(menu_descriptions))):
    print(f"- {menu_descriptions[i]}")
```


```python
# ==============================================================================
# 7. ChromaDB Setup for RAG
# ==============================================================================
DB_NAME = "cafegenius_menu_db"
# Use ephemeral client for simplicity in notebooks, or PersistentClient for
saving DB
# chroma_client = chromadb.Client() # In-memory
chroma_client = chromadb.PersistentClient(path="./chroma_db_cafe") # Saves to
disk

# Instantiate the embedding function
```

```
# Pass the API key if not configured globally, otherwise it uses the global
config
embed_fn = GeminiEmbeddingFunction(api_key=GOOGLE_API_KEY if 'GOOGLE_API_KEY' in
locals() else None)

# Create or get the collection
try:
    # Set the embedding function task type appropriately for adding documents
    embed_fn.task_type = "retrieval_document"
    db = chroma_client.get_or_create_collection(
        name=DB_NAME,
        embedding_function=embed_fn,
        metadata={"hnsw:space": "cosine"} # Specify distance metric if needed
    )
    logger.info(f"ChromaDB collection '{DB_NAME}' created or retrieved.")

    # Add menu descriptions to the vector database (only if collection is new or
empty)
    if db.count() == 0:
         logger.info(f"Adding {len(menu_descriptions)} documents to
ChromaDB...")
         db.add(
             documents=menu_descriptions,
             ids=[item['name'] for item in menu_data] # Use item names as IDs
             # Optionally add metadata: metadatas=[{"category":
item["category"], "price": item["price"]} for item in menu_data]
         )
         logger.info("Documents added successfully.")
    else:
         logger.info(f"Collection '{DB_NAME}' already contains {db.count()}
documents. Skipping add.")

except Exception as e:
    logger.exception(f"Failed to setup ChromaDB or add documents: {e}")
    # Handle error appropriately - maybe exit or proceed without RAG

# Verification Step
try:
    menu_documents_check = db.get(ids=[item['name'] for item in menu_data[:3]])
# Get first 3 by ID
    logger.info(f"ChromaDB Verification: Retrieved
{len(menu_documents_check.get('documents', []))} docs.")
    print("Sample documents from ChromaDB:")
    print(menu_documents_check.get('documents'))
except Exception as e:
    logger.error(f"Error verifying ChromaDB content: {e}")
```


```python
# ============================================================================
# 8. RAG Search Function
# ============================================================================
# Improvement 2: Enhanced error handling and return type
def search_menu(query: str, n_results: int = 3) -> Optional[List[str]]:
    """Search the menu database for relevant items using vector similarity."""
    if 'db' not in locals() or db is None:
         logger.error("ChromaDB collection 'db' is not available for
searching.")
         return None # Cannot search

    # Set the embedding function task type for querying
    embed_fn.task_type = "retrieval_query"
    logger.info(f"Performing RAG search for query: '{query}' with
```

```
n_results={n_results}")

    try:
        results = db.query(
            query_texts=[query],
            n_results=n_results,
            include=['documents'] # Only fetch documents
        )
        # Check results structure carefully
        if results and isinstance(results.get('documents'), list) and
results['documents']:
            # results['documents'] is a list containing one list of results (for
the one query)
            retrieved_docs = results['documents'][0]
            logger.info(f"RAG Search found {len(retrieved_docs)} documents.")
            return retrieved_docs
        else:
            logger.info(f"No relevant documents found via RAG for query:
'{query}'")
            return [] # Return empty list if no documents found

    except Exception as e:
        # Log the full exception traceback for debugging
        logger.exception(f"ChromaDB RAG search error for query '{query}': {e}")
        return None # Indicate failure occurred

### LOGGING - Enable/ Disable
### Test the search function
print("\nTesting RAG Search:")
test_query = "Something with espresso and milk"
search_results = search_menu(test_query)
if search_results is not None:
     print(f"Search results for '{test_query}':")
     for doc in search_results:
          print(f"- {doc}")
else:
     print("Search failed.")
```


```python
# ============================================================================
# 9. Order Management Class
# ============================================================================
# Improvement 1: Using the dedicated Order class
class Order:
    """Manages the items in a customer's order."""
    def __init__(self):
        self.items: List[Dict[str, Any]] = []
        logger.info("Order initialized.")

    def add_items(self, new_items: List[Dict[str, Any]]):
        """Adds validated items to the order."""
        if new_items:
             self.items.extend(new_items)
             logger.info(f"Added {len(new_items)} items to order. Current count:
{len(self.items)}")
        else:
             logger.warning("Attempted to add an empty list of items to the
order.")


    def get_total(self) -> float:
        """Calculates the total price of the order."""
```

```
        if not self.items:
            return 0.0
        return sum(item.get("price", 0.0) * item.get("quantity", 1) for item in
self.items)

    def display(self) -> str:
        """Formats the current order for display."""
        if not self.items:
            return "Your order is currently empty."

        order_text = "**Your Current Order:**\n\n" # Use Markdown
        for item in self.items:
            item_name = item.get("name", "Unknown Item")
            quantity = item.get("quantity", 1)
            price = item.get("price", 0.0)
            modifiers = item.get("modifiers", [])

            modifiers_text = ""
            if modifiers:
                modifiers_text = f" (Modifiers: {', '.join(modifiers)})"

            item_total = price * quantity
            order_text += f"- {quantity}x {item_name}{modifiers_text}: $
{item_total:.2f}\n"

        total = self.get_total()
        order_text += f"\n**Total: ${total:.2f}**"
        logger.info(f"Displaying order with {len(self.items)} items, total: $
{total:.2f}")
        return order_text

    def is_empty(self) -> bool:
        """Checks if the order is empty."""
        return not self.items
```


```python
# ================================================================================
# 10. Tool Function Definitions (for Gemini Function Calling)
# ================================================================================

def get_menu() -> Dict[str, Any]:
    """
    Retrieves the full cafe menu, organized by category,
    including names, prices, and modifiers for available items.
    """
    logger.info("Executing 'get_menu' function.")
    categories: Dict[str, List[Dict[str, Any]]] = {}
    for item in menu_data:
        if item.get("available", False):
            category = item.get("category", "Uncategorized")
            if category not in categories:
                categories[category] = []
            categories[category].append({
                "name": item.get("name", "N/A"),
                "price": item.get("price", 0.0),
                "modifiers": item.get("modifiers", [])
            })
    if not categories:
        logger.warning("'get_menu' found no available items.")
        return {"message": "Sorry, it seems nothing is available on the menu
right now."}
    return categories
```

```
def get_item_details(item_name: str) -> Optional[Dict[str, Any]]:
    """
    Gets detailed information about a specific menu item by its name.
    Returns the item's details if found, otherwise None.
    """
    logger.info(f"Executing 'get_item_details' for: {item_name}")
    item_name_lower = item_name.lower()
    # Exact match first
    for item in menu_data:
        if item.get("name", "").lower() == item_name_lower:
            if item.get("available", False):
                logger.info(f"Found exact match for available item:
{item_name}")
                return item # Return full item details
            else:
                logger.warning(f"Found exact match for '{item_name}', but it's
unavailable.")
                return {"name": item_name, "available": False, "message":
f"Sorry, {item_name} is currently unavailable."}

    # If no exact match, try RAG search as a fallback (optional, can be
demanding)
    logger.warning(f"Exact match not found for '{item_name}'. Trying RAG
search...")
    query = f"Detailed information about the menu item: {item_name}"
    similar_item_docs = search_menu(query, n_results=1) # Get the most similar
description

    if similar_item_docs:
        try:
            # Attempt to parse the item name from the RAG result
            # This assumes format "Item Name: Description..."
            retrieved_name = similar_item_docs[0].split(":")[0].strip()
            logger.info(f"RAG suggested similar item: {retrieved_name}")
            # Find this suggested item in the actual menu data
            for item in menu_data:
                if item.get("name", "").lower() == retrieved_name.lower() and
item.get("available", True):
                    logger.info(f"Returning details for RAG-suggested item:
{retrieved_name}")
                    # Inform the user it's a suggestion
                    item_with_note = item.copy()
                    item_with_note["note"] = f"Showing details for
'{retrieved_name}', which seemed similar to your request for '{item_name}'."
                    return item_with_note
        except Exception as e:
            logger.error(f"Error processing RAG result for similar item details:
{e}")

    # If still not found
    logger.error(f"Item '{item_name}' could not be found on the menu.")
    return {"found": False, "message": f"Sorry, I couldn't find '{item_name}' on
our menu. Please check the spelling or ask for the main menu."}
    # Returning a dictionary helps structure the "not found" response for the
LLM


def add_to_order(items: List[Dict[str, Any]]) -> Dict[str, Any]:
    """
    Adds items to the customer's order after validating item names and
modifiers.
    Returns a dictionary containing lists of successfully added items
```

```
('valid_items')
    and items that could not be added ('invalid_items_details').
    """
    logger.info(f"Executing 'add_to_order' for {len(items)} requested item(s).")
    valid_items_added: List[Dict[str, Any]] = []
    invalid_items_details: List[Dict[str, Any]] = []

    for requested_item in items:
        item_name = requested_item.get("name", "")
        requested_modifiers = requested_item.get("modifiers", [])
        quantity = requested_item.get("quantity", 1)

        if not item_name or quantity < 1:
            logger.warning(f"Invalid request format in 'add_to_order':
{requested_item}")
            invalid_items_details.append({"requested": requested_item, "reason":
"Missing name or invalid quantity."})
            continue

        # Find the item in menu_data (case-insensitive)
        menu_item_found = None
        item_name_lower = item_name.lower()
        for m_item in menu_data:
            if m_item.get("name", "").lower() == item_name_lower:
                menu_item_found = m_item
                break

        if menu_item_found and menu_item_found.get("available", False):
            # Item exists and is available, validate modifiers
            valid_modifiers_for_item = []
            invalid_modifiers_for_item = []
            available_modifiers = menu_item_found.get("modifiers", [])

            for mod in requested_modifiers:
                # Case-insensitive modifier check
                mod_found = False
                for avail_mod in available_modifiers:
                    if mod.lower() == avail_mod.lower():
                        valid_modifiers_for_item.append(avail_mod) # Use the
canonical name
                        mod_found = True
                        break
                if not mod_found:
                    invalid_modifiers_for_item.append(mod)

            # Prepare the item to be potentially added
            item_to_add = {
                "name": menu_item_found["name"], # Use canonical name
                "price": menu_item_found.get("price", 0.0),
                "quantity": quantity,
                "modifiers": valid_modifiers_for_item
            }
            valid_items_added.append(item_to_add)

            # Log if any requested modifiers were invalid for this valid item
            if invalid_modifiers_for_item:
                 logger.warning(f"Invalid modifiers requested for '{item_name}':
{invalid_modifiers_for_item}")
                 # Optionally add details about invalid modifiers to the
response structure
                 invalid_items_details.append({
                     "requested_item_name": item_name,
                     "reason": "Some modifiers were invalid.",
                     "invalid_modifiers": invalid_modifiers_for_item,
```

```
                    "added_item": item_to_add # Show what was added despite
invalid mods
                })

        else:
            # Item not found or not available
            reason = "Item not found on menu." if not menu_item_found else
f"Item '{item_name}' is currently unavailable."
            logger.warning(f"Could not add item '{item_name}'. Reason:
{reason}")
            invalid_items_details.append({"requested_item_name": item_name,
"reason": reason})

    # Construct the result dictionary
    result = {
        "status": f"Processed {len(items)} requests. Added
{len(valid_items_added)} items.",
        "valid_items": valid_items_added, # These should be added to the Order
object by the calling code
        "invalid_items_details": invalid_items_details
    }
    logger.info(f"Finished 'add_to_order'. Result: {result}")
    return result


def get_recommendations(preferences: List[str], dietary_restrictions: List[str]
= None) -> Dict[str, Any]:
    """
    Gets personalized recommendations based on customer preferences and optional
dietary restrictions,
    using RAG search on menu descriptions.
    """
    if dietary_restrictions is None:
        dietary_restrictions = []

    logger.info(f"Executing 'get_recommendations'. Preferences: {preferences},
Restrictions: {dietary_restrictions}")

    # Construct a query for RAG
    query = f"Recommend cafe items for someone who likes {',
'.join(preferences)}"
    if dietary_restrictions:
        query += f" and needs options suitable for {',
'.join(dietary_restrictions)} dietary restrictions (e.g., gluten-free, vegan,
dairy-free)."
    query += ". Consider item descriptions, ingredients, and common
associations." # Add more context

    # Get relevant item descriptions from the vector database
    relevant_descriptions = search_menu(query, n_results=5) # Increase results
for better filtering

    if relevant_descriptions is None:
         return {"error": "Recommendation search failed. Please try again
later."}
    if not relevant_descriptions:
         return {"message": "I couldn't find specific recommendations based on
that. Maybe try broadening your preferences?"}


    # Extract item names and retrieve full details from menu_data
    recommended_items_details: List[Dict[str, Any]] = []
    seen_names = set()
```

```
    for desc in relevant_descriptions:
        try:
            # Attempt to robustly extract item name (assuming "Name:
Description..." format)
            item_name = desc.split(":")[0].strip()
            if item_name and item_name not in seen_names:
                 # Find the full item details in menu_data
                 item_found = None
                 item_name_lower = item_name.lower()
                 for item in menu_data:
                     if item.get("name","").lower() == item_name_lower and
item.get("available", True):
                         item_found = item
                         break

                 if item_found:
                     # Basic filtering (can be improved with more
metadata/logic)
                     # e.g., check if description mentions dietary needs if
provided
                     passes_filter = True # Add filtering logic here if needed
                     if passes_filter:
                          recommended_items_details.append(item_found)
                          seen_names.add(item_name)

                 # Limit the number of recommendations returned
                 if len(recommended_items_details) >= 3: # Limit to top 3
relevant items
                     break

        except Exception as e:
            logger.warning(f"Could not parse item name from description
'{desc[:50]}...': {e}")
            continue # Skip this description

    if not recommended_items_details:
         return {"message": "Based on the search, I don't have specific
recommendations matching everything. You could check the full menu!"}
    else:
         logger.info(f"Returning {len(recommended_items_details)}
recommendations.")
         return {"recommendations": recommended_items_details}

```

```python
# ==============================================================================
# 11. Define Tools for Gemini Model (Corrected Schema)
# ==============================================================================
# Ensure glm (google.ai.generativelanguage) is imported

tools_list = [
    glm.Tool(
        function_declarations=[
            glm.FunctionDeclaration(
                name="get_menu",
                description="Retrieves the full cafe menu, organized by
category, including names, prices, and modifiers for available items.",
                parameters=glm.Schema(type=glm.Type.OBJECT, properties={}) # No
parameters needed
            ),
            glm.FunctionDeclaration(
                name="get_item_details",
```

```
                description="Gets detailed information (description, price,
modifiers, availability) about a specific menu item by its name.",
                parameters=glm.Schema(
                    type=glm.Type.OBJECT,
                    properties={
                        "item_name": glm.Schema(type=glm.Type.STRING,
description="The exact name of the menu item to look up.")
                    },
                    required=["item_name"]
                )
            ),
            glm.FunctionDeclaration(
                name="add_to_order",
                description="Adds one or more items to the customer's current
order. Specify item name, quantity, and any desired modifiers.",
                parameters=glm.Schema(
                    type=glm.Type.OBJECT,
                    properties={
                        "items": glm.Schema(
                            type=glm.Type.ARRAY,
                            description="A list of items to add to the order.",
                            items=glm.Schema(
                                type=glm.Type.OBJECT,
                                properties={
                                    "name": glm.Schema(type=glm.Type.STRING,
description="The name of the menu item."),

                                    "quantity":
glm.Schema(type=glm.Type.INTEGER, description="How many of this item to order
(default will be handled by the function if not provided)."),
                                    "modifiers": glm.Schema(
                                        type=glm.Type.ARRAY,
                                        description="List of optional modifiers
(e.g., 'Oat Milk', 'Extra Shot', 'Warmed').",
                                        items=glm.Schema(type=glm.Type.STRING)
                                    )
                                },
                                required=["name"] # Quantity is not strictly
required here; handled in function
                            )
                        )
                    },
                    required=["items"]
                )
            ),
            glm.FunctionDeclaration(
                name="get_recommendations",
                description="Suggests menu items based on customer preferences
(e.g., 'sweet', 'strong coffee', 'breakfast') and optional dietary restrictions
(e.g., 'vegan', 'gluten-free').",
                parameters=glm.Schema(
                    type=glm.Type.OBJECT,
                    properties={
                        "preferences": glm.Schema(
                            type=glm.Type.ARRAY,
                            description="List of customer preferences (flavors,
meal types, etc.).",
                            items=glm.Schema(type=glm.Type.STRING)
                        ),
                        "dietary_restrictions": glm.Schema(
                            type=glm.Type.ARRAY,
                            description="List of dietary needs or restrictions
(optional).",
                            items=glm.Schema(type=glm.Type.STRING)
```

```
                    )
                },
                required=["preferences"]
            )
        )
    ]
)
]

logger.info(f"Defined {len(tools_list[0].function_declarations)} tools for the
Gemini model (schema corrected).")
```


```python
# ================================================================================
# 12. System Prompt and Model Configuration
# ================================================================================
system_prompt = """You are CafeGenius, a cheerful, knowledgeable, and efficient
AI assistant for our cafe. Your goal is to help customers explore the menu, make
selections, get recommendations, and place orders smoothly.

**Key Instructions:**
* **Use Tools:** Rely on your available tools (`get_menu`, `get_item_details`,
`add_to_order`, `get_recommendations`) to answer questions accurately about the
menu, item details, ordering, and recommendations. Don't guess menu details or
availability.
* **Clarity:** When providing menu information or order summaries, format it
clearly using Markdown (like bullet points or categories).
* **Order Confirmation:** When adding items via `add_to_order`, clearly state
what was successfully added and mention any issues (like invalid items or
modifiers) reported by the tool.
* **Recommendations:** Base recommendations on the results from the
`get_recommendations` tool. Explain *why* you're recommending something based on
the customer's request.
* **Tone:** Be friendly, polite, and helpful, like a great barista. Keep
responses concise but informative.
* **Limitations:** If you cannot fulfill a request (e.g., item not found, RAG
search fails), politely explain the issue and suggest alternatives (like viewing
the full menu or rephrasing). Do not make up information.
"""

# Configure Generation Settings (example, adjust as needed)
generation_config = genai.GenerationConfig(
    temperature=0.7, # Balance creativity and coherence
    top_p=0.95,
    top_k=40,
    # max_output_tokens=1024, # Optional: Limit response length
    # stop_sequences=["User:", "\n\n"], # Optional: Define stop sequences
)

safety_settings = [ # Adjust safety settings based on requirements
    {"category": "HARM_CATEGORY_HARASSMENT", "threshold":
"BLOCK_MEDIUM_AND_ABOVE"},
    {"category": "HARM_CATEGORY_HATE_SPEECH", "threshold":
"BLOCK_MEDIUM_AND_ABOVE"},
    {"category": "HARM_CATEGORY_SEXUALLY_EXPLICIT", "threshold":
"BLOCK_MEDIUM_AND_ABOVE"},
    {"category": "HARM_CATEGORY_DANGEROUS_CONTENT", "threshold":
"BLOCK_MEDIUM_AND_ABOVE"},
]


logger.info("System prompt and generation configuration set.")
```

```

```python
# ================================================================================
# 13. Main Chat Function
# ================================================================================

# Mapping function names to actual Python functions
available_functions = {
    "get_menu": get_menu,
    "get_item_details": get_item_details,
    "add_to_order": add_to_order,
    "get_recommendations": get_recommendations,
}

def handle_function_call(function_call: glm.FunctionCall, current_order: Order)
-> glm.Part:
    """Handles executing a function call from the Gemini model."""
    function_name = function_call.name
    args_dict = {k: v for k, v in function_call.args.items()}
    logger.info(f"Received function call: {function_name} with args:
{args_dict}")

    func = available_functions.get(function_name)
    if not func:
        logger.error(f"Unknown function called: {function_name}")
        result = {"error": f"Unknown function name: {function_name}"}
    else:
        try:
            # Call the appropriate Python function with its arguments
            result = func(**args_dict)
            logger.info(f"Function '{function_name}' executed successfully.")

            # --- Improvement 1: Integrate Order Class ---
            # If add_to_order was called, update the actual order object
            if function_name == "add_to_order" and isinstance(result, dict) and
"valid_items" in result:
                items_to_officially_add = result.get("valid_items", [])
                if items_to_officially_add:
                     current_order.add_items(items_to_officially_add)
                     logger.info(f"Updated Order object with
{len(items_to_officially_add)} items.")
                # The 'result' dict already contains info about valid/invalid
items,
                # which will be sent back to the LLM below.

        except Exception as e:
            logger.exception(f"Error executing function '{function_name}': {e}")
            result = {"error": f"Error during execution of {function_name}:
{str(e)}"}

      # Ensure the result is serializable (convert to dict if needed)
    if not isinstance(result, dict):
        # Handle cases where functions might return lists, strings, None, etc.
        # Wrap them in a dictionary for consistent processing by the LLM.
        if result is None:
            result = {"status": "Operation completed, no specific data
returned."}
        elif isinstance(result, list):
             result = {"items": result} # Example wrapping for a list result
        else:
             result = {"result": str(result)} # Default wrapping
```
```

```
    # Convert the result dictionary to a Protobuf Struct
    try:
        response_struct = struct_pb2.Struct()
        response_struct.update(result)
        function_response = glm.FunctionResponse(name=function_name,
response=response_struct)
        return glm.Part(function_response=function_response)

    except Exception as e:
        logger.exception(f"Error converting result for {function_name} to
Struct: {e}")
        # Fallback: return an error message within the FunctionResponse
structure
        error_struct = struct_pb2.Struct()
        error_struct.update({"error": f"Failed to serialize result: {str(e)}"})
        function_response = glm.FunctionResponse(name=function_name,
response=error_struct)
        return glm.Part(function_response=function_response)
```

```python
##############################################################################
# 14. Initiates and manages the chat session with the CafeGenius assistant.
##############################################################################
def chat_with_cafe_genius():


    if not GOOGLE_API_KEY:
         print("Cannot start chat: GOOGLE_API_KEY is not configured.")
         return

    # Select the model - Check availability and choose appropriately
    # List models - requires API call, handle potential errors
    available_models = []
    try:
        # Note: genai.list_models() might return an iterator or list depending
on version
        for m in genai.list_models():
             # Check if the model supports 'generateContent' (required for chat)
             if 'generateContent' in m.supported_generation_methods:
                  available_models.append(m.name)
    except Exception as e:
        logger.error(f"Failed to list available models: {e}")
        print("Error: Could not retrieve list of available models. Cannot
proceed.")
        return

    # Choose a preferred model, falling back if necessary
    preferred_model = "models/gemini-2.0-flash" # Or "models/gemini-1.5-pro-
latest"
    fallback_model = "models/gemini-1.5-flash-latest" # A generally available
fallback

    if preferred_model in available_models:
         model_to_use = preferred_model
    elif fallback_model in available_models:
         model_to_use = fallback_model
    else:
         # Try finding any 'pro' model as a last resort
         pro_models = [m for m in available_models if 'pro' in m and 'embedding'
not in m]
         if pro_models:
             model_to_use = pro_models[0]
```

```
        else:
            logger.error(f"Could not find a suitable Gemini model. Available:
{available_models}")
            print("Error: No suitable Gemini model found (tried 2.0-flash ,
1.5-flash). Cannot start chat.")
            return

    print(f"--> CafeGenius Assistant is using: {model_to_use} , Gemini Model.")
    # Keep the existing logger line as well:
    logger.info(f"Using Gemini model: {model_to_use}")

    # Initialize the chat model
    model = genai.GenerativeModel(
        model_name=model_to_use,
        generation_config=generation_config,
        safety_settings=safety_settings,
        tools=tools_list,
        system_instruction=system_prompt # Pass system prompt here
    )

    # Start a chat session (maintains history)
    chat = model.start_chat(enable_automatic_function_calling=False) # Manual
control over function calls
    logger.info("Chat session started.")

    # Improvement 1: Instantiate the Order class here
    current_order = Order()

    def print_boxed_message():
        lines = [
            "☕ Welcome to CafeGenius Assistant V1.0!",
            "(Developed by: Erwin R. Pasia | erwinpasia@gmail.com)",
            "",
            "Hi there! I’m your virtual barista.",
            "Curious about what’s brewing? Check the menu, get a personalized
pick, or place your order anytime.",
            "",
            "Just type:",
            "  'menu'        – View our offerings",
            "  'recommend'   – Get a drink suggestion",
            "  'order'       – Start your order",
            "  'show order'  – Review your current order",
            "  'exit or bye' – Leave the assistant",
            "",
            "For example:",
            "Step 1: What are on the menu today?",
            "Step 2: I would like to order Flat White, Mocha, and Latte. No
modifiers on them. Add 3 banana bread."
        ]

        # Find the length of the longest line
        max_length = max(len(line) for line in lines)
        border = "—" * (max_length + 4)

        print("┌" + border + "┐")
        for line in lines:
            print(f"|  {line.ljust(max_length)}  |")
        print("└" + border + "┘")

        print("-" * (max_length + 10))  # Optional footer line

    print_boxed_message()
```

```
    while True:
        try:
            user_input = input("You: ")
            print("-" * 28) # Separator

            if user_input.lower() in ["exit", "quit", "bye", "goodbye"]:
                print("CafeGenius: Thanks for visiting! Have a great day!")
                if not current_order.is_empty():
                     print("\nFinal Order Summary:")
                     display(Markdown(current_order.display())) # Display final
order nicely
                break

            # --- Improvement 1: Use Order class for 'show order' ---
            if user_input.lower() in ["show order", "my order", "current order",
"view order"]:
                print("CafeGenius:")
                display(Markdown(current_order.display())) # Use the class
method
                print("-" * 28)
                continue

            # --- Direct Menu Request ---
            if user_input.lower() in ["menu", "show menu", "see menu", "what's
on the menu", "what do you have"]:
                 print("CafeGenius: Getting the menu for you...")
                 menu_result = get_menu()
                 if isinstance(menu_result, dict) and "message" in menu_result:
                      print(f"CafeGenius: {menu_result['message']}")
                 elif isinstance(menu_result, dict):
                      menu_text = "**Full Menu:**\n\n"
                      for category, items in menu_result.items():
                           menu_text += f"### {category}\n" # H3 for categories
                           for item in items:
                                mods = f" (Modifiers: {',
'.join(item['modifiers'])})" if item.get('modifiers') else ""
                                menu_text += f"- **{item['name']}**: $
{item['price']:.2f}{mods}\n"
                           menu_text += "\n"
                      display(Markdown(menu_text))
                 else:
                      print("CafeGenius: Sorry, I couldn't retrieve the menu
format correctly.")
                 print("-" * 28)
                 continue

            # --- Send message to Gemini ---
            logger.info(f"Sending user input to Gemini: '{user_input}'")
            response = chat.send_message(user_input)
            logger.info("Received response from Gemini.")

            # --- Process Gemini's response (Check for Function Calls) ---
            response_part = response.candidates[0].content.parts[0]

            if hasattr(response_part, 'function_call') and
response_part.function_call:
                 function_call = response_part.function_call
                 logger.info("Gemini requested a function call.")


                 api_request_part = handle_function_call(function_call,
current_order)
```

```
                logger.info(f"Sending function response back to Gemini for
function: {function_call.name}")
                response = chat.send_message(api_request_part)
                logger.info("Received final response from Gemini after function
call.")

                # Process the final text response after the function call cycle
                final_text = response.candidates[0].content.parts[0].text
                print(f"CafeGenius: {final_text}")
                display(Markdown(final_text)) # Display nicely

           elif hasattr(response_part, 'text'):
                # --- Handle regular text response ---
                text_response = response_part.text
                logger.info("Received text response from Gemini.")
                print("CafeGenius:")
                display(Markdown(text_response)) # Display nicely
           else:
                # Handle unexpected response content (e.g., blocked content)
                logger.warning(f"Received unexpected response part:
{response_part}")
                print("CafeGenius: I received a response I couldn't process.
Could you try rephrasing?")

           print("-" * 28) # Separator


       except KeyboardInterrupt:
            print("\nCafeGenius: Exiting chat.")
            if not current_order.is_empty():
                print("\nFinal Order Summary:")
                display(Markdown(current_order.display()))
            break
       except Exception as e:
           logger.exception(f"An error occurred in the chat loop: {e}")
           print(f"\n🤖 CafeGenius: Oops! Something went wrong: {str(e)}.
Please try again or rephrase.")
           # Optional: attempt to restart chat or exit gracefully
           #chat = model.start_chat(...) # Potential restart (careful with
history)
```


```python
# ====================================================================================
# 15. Run/Re-Run to Initiate/Re-Initiate CafeGenius Assistant
# ====================================================================================
if __name__ == "__main__" or '__file__' not in locals(): # Check if running as
script or notebook cell
    # Only run the chat if the API key was successfully configured
    if 'GOOGLE_API_KEY' in locals() and GOOGLE_API_KEY:
         chat_with_cafe_genius()
    else:
         print("Chat cannot start because the Google API Key is missing or
invalid.")

```