Project Overview

This notebook implements an AI-powered Customer Support Desk capable of understanding user queries, retrieving relevant knowledge base information, generating natural language responses, evaluating quality, and performing escalation when required. The system integrates intent classification, semantic search using FAISS, Gemini 1.5 LLM generation, and a Gradio-based interactive UI.

1. Library Installation

Library Installation and Environment Setup

In this step, we prepare the Colab environment by uninstalling older versions of the google-generativeai package and reinstalling the latest stable release. This ensures compatibility with Gemini 1.5 models and prevents deprecated API usage. We also install all supporting dependencies required for embeddings, vector search, and the Gradio interface.

```
!pip uninstall -y google-generativeai google_ai_genai google-ai-generativelanguage
```

```
Found existing installation: google-generativeai 0.8.5
Uninstalling google-generativeai-0.8.5:
  Successfully uninstalled google-generativeai-0.8.5
WARNING: Skipping google_ai_genai as it is not installed.
Found existing installation: google-ai-generativelanguage 0.6.15
Uninstalling google-ai-generativelanguage-0.6.15:
  Successfully uninstalled google-ai-generativelanguage-0.6.15
```

```
!pip install -U google-generativeai
```

```
Collecting google-generativeai
  Using cached google_generativeai-0.8.5-py3-none-any.whl.metadata (3.9 kB)
Collecting google-ai-generativelanguage==0.6.15 (from google-generativeai)
  Using cached google_ai_generativelanguage-0.6.15-py3-none-any.whl.metadata (5.7 kB)
Requirement already satisfied: google-api-core in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (2.28.1)
Requirement already satisfied: google-api-python-client in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (2
Requirement already satisfied: google-auth>=2.15.0 in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (2.38.0
Requirement already satisfied: protobuf in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (5.29.5)
Requirement already satisfied: pydantic in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (2.11.10)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (4.67.1)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.12/dist-packages (from google-generativeai) (4.15.0)
Requirement already satisfied: proto-plus<2.0.0dev,>=1.22.3 in /usr/local/lib/python3.12/dist-packages (from google-ai-generativ
Requirement already satisfied: googleapis-common-protos<2.0.0,>=1.56.2 in /usr/local/lib/python3.12/dist-packages (from google-a
Requirement already satisfied: requests<3.0.0,>=2.18.0 in /usr/local/lib/python3.12/dist-packages (from google-api-core->google-
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.12/dist-packages (from google-auth>=2.15.0->goog
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.12/dist-packages (from google-auth>=2.15.0->googl
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.12/dist-packages (from google-auth>=2.15.0->google-genera
Requirement already satisfied: httplib2<1.0.0,>=0.19.0 in /usr/local/lib/python3.12/dist-packages (from google-api-python-client
Requirement already satisfied: google-auth-httplib2<1.0.0,>=0.2.0 in /usr/local/lib/python3.12/dist-packages (from google-api-py
Requirement already satisfied: uritemplate<5,>=3.0.1 in /usr/local/lib/python3.12/dist-packages (from google-api-python-client->
Requirement already satisfied: annotated-types>=0.6.0 in /usr/local/lib/python3.12/dist-packages (from pydantic->google-generati
Requirement already satisfied: pydantic-core==2.33.2 in /usr/local/lib/python3.12/dist-packages (from pydantic->google-generativ
Requirement already satisfied: typing-inspection>=0.4.0 in /usr/local/lib/python3.12/dist-packages (from pydantic->google-genera
Requirement already satisfied: grpcio<2.0.0,>=1.33.2 in /usr/local/lib/python3.12/dist-packages (from google-api-core[grpc]!=2.0
Requirement already satisfied: grpcio-status<2.0.0,>=1.33.2 in /usr/local/lib/python3.12/dist-packages (from google-api-core[grp
Requirement already satisfied: pyparsing<4,>=3.0.4 in /usr/local/lib/python3.12/dist-packages (from httplib2<1.0.0,>=0.19.0->goo
Requirement already satisfied: pyasn1<0.7.0,>=0.6.1 in /usr/local/lib/python3.12/dist-packages (from pyasn1-modules>=0.2.1->goog
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests<3.0.0,>=2.18.0
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests<3.0.0,>=2.18.0->google-api
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests<3.0.0,>=2.18.0->goog
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests<3.0.0,>=2.18.0->goog
Using cached google_generativeai-0.8.5-py3-none-any.whl (155 kB)
Using cached google_ai_generativelanguage-0.6.15-py3-none-any.whl (1.3 MB)
Installing collected packages: google-ai-generativelanguage, google-generativeai
Successfully installed google-ai-generativelanguage-0.6.15 google-generativeai-0.8.5
WARNING: The following packages were previously imported in this runtime:
  [google]
You must restart the runtime in order to use newly installed versions.
```

RESTART SESSION

```
!pip show google-generativeai
```

```
Name: google-generativeai
Version: 0.8.5
Summary: Google Generative AI High level API client library and tools.
```

```
Home-page: https://github.com/google/generative-ai-python
Author: Google LLC
Author-email: googleapis-packages@google.com
License: Apache 2.0
Location: /usr/local/lib/python3.12/dist-packages
Requires: google-ai-generativelanguage, google-api-core, google-api-python-client, google-auth, protobuf, pydantic, tqdm, typing
Required-by:
```

```
import google.generativeai as genai
genai.list_models()
```

```
<generator object list_models at 0x7d3e0a99ef20>
```

```
!pip install faiss-cpu
```

```
Requirement already satisfied: faiss-cpu in /usr/local/lib/python3.12/dist-packages (1.13.0)
Requirement already satisfied: numpy<3.0,>=1.25.0 in /usr/local/lib/python3.12/dist-packages (from faiss-cpu) (2.0.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.12/dist-packages (from faiss-cpu) (25.0)
```

2. Imports

Importing Core Dependencies

Here, we import all the essential Python libraries required for the project, including modules for:

- large language model interaction
- vector embedding and FAISS-based retrieval
- dataclasses and type definitions
- Gradio UI creation
- utility functions for text processing

These imports lay the foundation for the complete AI customer support pipeline.

```
import os
import time
import json
import traceback
from dataclasses import dataclass, field
from typing import Any, Callable, Dict, List, Optional, Tuple
import threading
import numpy as np
import faiss
import tiktoken
import google.generativeai as genai
import gradio as gr
```

3. Configuration

Global Configuration and API Initialization

This section configures core project settings such as the Gemini API key, model parameters, and global constants used across the pipeline. Ensuring proper configuration is essential for seamless communication with the LLM and consistent system behavior.

```
# Configuration

from google.colab import userdata
import os

API_KEY = userdata.get("GEMINI_API_KEY")
if not API_KEY:
    print("WARNING: GEMINI_API_KEY not set in environment. LLM calls will fail until configured.")
else:
    genai.configure(api_key=API_KEY)
```

4. Utilities

📝 Utility Functions: Tokenization, Chunking, and Embedding

In this section, we implement helper functions for text preprocessing.

These utilities handle tasks such as:

- breaking documents into manageable chunks
- generating vector embeddings
- normalizing or preparing text for retrieval

These low-level operations are used throughout the retrieval pipeline.

```
# Utilities: tokenization, chunking, embeddings

try:
    _default_token_enc = tiktoken.encoding_for_model("gpt-4")
except Exception:
    _default_token_enc = tiktoken.get_encoding("cl100k_base")


def chunk_text(text: str, max_tokens: int = 300) -> List[str]:
    enc = _default_token_enc
    toks = enc.encode(text)
    chunks = []
    for i in range(0, len(toks), max_tokens):
        chunk = enc.decode(toks[i:i + max_tokens])
        chunks.append(chunk)
    return chunks


def embed_texts(text_list: List[str]) -> np.ndarray:
    """Embed list of texts using Gemini embeddings model 'text-embedding-004'.
    Returns float32 matrix of shape (N, D).
    """
    if isinstance(text_list, str):
        text_list = [text_list]


    embeddings = []
    for text in text_list:
        try:
            resp = genai.embed_content(model="text-embedding-004", content=text)
            emb = np.array(resp["embedding"], dtype="float32")
        except Exception as e:
            # graceful fallback to a random vector so FAISS code still runs in offline testing
            print("Embedding call failed:", e)
            emb = np.random.rand(768).astype("float32")
        embeddings.append(emb)


    return np.vstack(embeddings)
```

5. FAISS Index

📝 FAISS Index Construction and Helper Methods

This cell contains the logic for building and querying the FAISS vector store.

FAISS enables efficient similarity search by comparing user query embeddings with knowledge base document embeddings.

These helpers support storing vectors, searching for relevant chunks, and managing the vector index.

```
# FAISS index helpers

def build_kb_embeddings(docs: List[str]) -> Tuple[np.ndarray, List[Dict[str, Any]]]:
    """Take a list of document strings, chunk them, embed them, and return embedding matrix
    plus a list of chunk metadata dictionaries: {'id','text','source'}
    """
```

```
    all_chunks = []
    for i, doc in enumerate(docs):
        chunks = chunk_text(doc)
        for j, c in enumerate(chunks):
            all_chunks.append({
                "id": f"doc_{i}_chunk_{j}",
                "text": c,
                "source": f"doc_{i}"
            })

    print(f"Total chunks: {len(all_chunks)}")
    if len(all_chunks) == 0:
        return np.zeros((0, 768), dtype="float32"), []

    texts = [c["text"] for c in all_chunks]
    emb_matrix = embed_texts(texts)
    return emb_matrix, all_chunks


def build_faiss_index(emb_matrix: np.ndarray) -> faiss.IndexFlatL2:
    if emb_matrix.size == 0:
        # create a small dummy index with dimension 768
        dim = 768
        index = faiss.IndexFlatL2(dim)
        return index
    dim = emb_matrix.shape[1]
    index = faiss.IndexFlatL2(dim)
    index.add(emb_matrix)
    print("FAISS index built with", index.ntotal, "vectors")
    return index


def search_index(index: faiss.IndexFlatL2, query: str, chunks: List[Dict[str, Any]], k: int = 3) -> List[Dict[str, Any]]:
    """Return top-k chunk dicts with fields: {id, text, source, distance}


    If embedding call fails, returns empty list.
    """
    try:
        q_emb = embed_texts([query]).reshape(1, -1)
    except Exception as e:
        print("Query embedding failed:", e)
        return []


    if index.ntotal == 0:
        return []


    distances, indices = index.search(q_emb, k)
    results = []
    for dist, idx in zip(distances[0], indices[0]):
        if idx < 0 or idx >= len(chunks):
            continue
        chunk = chunks[idx]
        out = {
            "id": chunk.get("id"),
            "text": chunk.get("text"),
            "source": chunk.get("source"),
            "distance": float(dist)
        }
        results.append(out)
    return results
```

6. Agent Base Classes and Tools

📝 Agent Base Structure, Shared Tools, and Response Definitions

This section defines the foundational components used by all agents in the system.

It includes:

- BaseAgent class (providing shared functionality)
- Structured response dataclasses

- Reusable helper tools for logging and processing

These abstractions enforce consistent behavior across all specialized agents and simplify the design of the overall pipeline.

```
# Agent base, response dataclass, tools

@dataclass
class AgentResponse:
    agent_name: str
    success: bool
    output: Any = None
    error: Optional[str] = None
    meta: Dict[str, Any] = field(default_factory=dict)


class Agent:
    def __init__(self, name: str):
        self.name = name
        self._logs: List[Dict[str, Any]] = []
        self._lock = threading.Lock()
        self.llm: Optional["LLMClient"] = None # make llm attribute explicit

    def _log(self, level: str, message: str, **meta):
        entry = {"ts": time.time(), "level": level, "agent": self.name, "message": message, "meta": meta}
        with self._lock:
            self._logs.append(entry)
        print(f"[{self.name}][{level}] {message}")

    def info(self, message: str, **meta):
        self._log("INFO", message, **meta)

    def warn(self, message: str, **meta):
        self._log("WARN", message, **meta)

    def error(self, message: str, **meta):
        self._log("ERROR", message, **meta)

    def get_logs(self) -> List[Dict[str, Any]]:
        return list(self._logs)

    def run_safe(self, fn: Callable[..., Any], *args, **kwargs) -> AgentResponse:
        start = time.time()
        try:
            out = fn(*args, **kwargs)
            latency = time.time() - start
            self.info("Call succeeded", latency=latency)
            return AgentResponse(agent_name=self.name, success=True, output=out, meta={"latency": latency})
        except Exception as e:
            latency = time.time() - start
            tb = traceback.format_exc()
            self.error("Call failed", error=str(e), latency=latency)
            return AgentResponse(agent_name=self.name, success=False, error=str(e), meta={"latency": latency, "traceback": tb})


class Tool:
    def __init__(self, name: str, fn: Callable[..., Any]):
        self.name = name
        self.fn = fn

    def call(self, *args, **kwargs):
        return self.fn(*args, **kwargs)
```

7. LLM Client Wrapper

LLM Client Wrapper for Gemini 1.5 Models

Here, we implement a dedicated wrapper class around the Google Gemini API. This wrapper standardizes how prompts are sent to the model, manages generation configuration, and handles errors gracefully.

By abstracting these operations, downstream agents can easily generate content using a clean interface.

```
# LLM client wrapper (Google Generative AI) - updated default model

class LLMClient:
    def __init__(self, model="gemini-1.5-flash", temperature=0.0, max_output_tokens=512):
        self.model_name = model
        self.temperature = temperature
        self.max_output_tokens = max_output_tokens

        try:
            self.model = genai.GenerativeModel(self.model_name)
        except Exception as e:
            print("Failed to initialize LLM model:", e)
            self.model = None

    def generate_content(self, prompt: str):
        class Resp:
            text = ""

        r = Resp()

        if not self.model:
            r.text = "Generation failed: LLM not initialized"
            return r

        try:
            response = self.model.generate_content(
                prompt,   # <-- SIMPLE STRING (new API)
                generation_config={
                    "temperature": self.temperature,
                    "max_output_tokens": self.max_output_tokens
                }
            )
            r.text = response.text
            return r

        except Exception as e:
            r.text = f"Generation failed: {e}"
            return r
```

8. Agent Implementations

📝 Specialized Agents for Classification, Retrieval, Generation, Evaluation, and Escalation

This cell defines the individual agents that perform the core system tasks:

- intent classification
- semantic retrieval
- LLM-based response generation
- quality evaluation
- escalation decision-making

Each agent inherits from the base structure and implements customized behavior for its respective role.

```
# Agent implementations: classifier, retriever, generator, evaluator, escalation

class IntentClassifierAgent(Agent):
    def __init__(self):
        super().__init__("classifier")

    def classify_intent(self, user_msg: str) -> str:
        # For now, a very simple keyword-based classification
        if "sso" in user_msg.lower() or "oauth" in user_msg.lower():
            return "sso_oauth_inquiry"
        elif "refund" in user_msg.lower():
            return "refund_request"
        return "general_inquiry"

class RetrieverAgent(Agent):
    def __init__(self, index: faiss.IndexFlatL2, chunks: List[Dict[str, Any]]):
        super().__init__("retriever")
```

```
            self.index = index
            self.chunks = chunks

        def retrieve(self, query: str, k: int = 3) -> List[Dict[str, Any]]:
            resp = self.run_safe(lambda: search_index(self.index, query, self.chunks, k=k))
            if resp.success and resp.output:
                return resp.output
            return []


    class ResponseGeneratorAgent(Agent):
        def __init__(self):
            super().__init__("generator")

        def generate_answer(self, intent: str, user_msg: str, retrieved_chunks: List[Dict[str, Any]]) -> str:
            # compose context
            context = "\n\n".join([c.get("text", "") for c in retrieved_chunks])
            prompt = f"You are a SaaS support assistant. Intent: {intent}\nContext:\n{context}\nUser message:\n{user_msg}\nProvide

            if not self.llm or not self.llm.model:
                # fallback canned response
                return "Thanks for reaching out. We support SSO/OAuth integrations; please check your account settings or contact s

            resp = self.run_safe(lambda: self.llm.generate_content(prompt).text)
            if resp.success and isinstance(resp.output, str):
                return resp.output
            return "Generation failed: see agent logs."


    class EvaluationAgent(Agent):
        def __init__(self):
            super().__init__("evaluator")

        def evaluate(self, user_msg: str, answer: str) -> int:
            if not self.llm or not self.llm.model:
                return 3
            prompt = f"Score 1-5 how helpful this answer is for the user.\nUser:\n{user_msg}\nAnswer:\n{answer}\nReturn only a sing
            resp = self.run_safe(lambda: self.llm.generate_content(prompt).text)
            if not resp.success:
                return 3
            try:
                return int(str(resp.output).strip().split()[0])
            except Exception:
                return 3


    class EscalationAgent(Agent):
        def __init__(self):
            super().__init__("escalation")

        def check_escalation(self, score: int) -> Tuple[bool, str]:
            if score < 3:
                return True, "Escalating to human support."
            return False, "No escalation needed."
```

9. Session Memory

📝 Session Memory and Conversation Tracking

This part implements lightweight memory storage for tracking conversation history within a session.

Session memory allows the orchestrator to maintain context across multiple user inputs, ensuring more coherent and consistent responses.

```
    # Session Memory

    class SessionMemory:
        def __init__(self, max_turns: int = 10):
            self.max_turns = max_turns
            self.sessions: Dict[str, List[Dict[str, str]]] = {}

        def append_message(self, session_id: str, role: str, text: str):
```

```
            if session_id not in self.sessions:
                self.sessions[session_id] = []
            self.sessions[session_id].append({"role": role, "text": text})
            if len(self.sessions[session_id]) > self.max_turns:
                self.sessions[session_id] = self.sessions[session_id][-self.max_turns:]

        def get_context(self, session_id: str, window: Optional[int] = None) -> List[Dict[str, str]]:
            ctx = self.sessions.get(session_id, [])
            if window:
                return ctx[-window:]
            return ctx
```

10. Orchestrator

📝 Central Orchestrator for Coordinating the AI Pipeline

The orchestrator controls the full lifecycle of a user request by invoking all agents in the correct sequence.

It handles:

1. intent detection
2. context retrieval
3. response generation
4. evaluation
5. escalation decisions

It returns a fully structured result that is ready to present to the user.

```
# Orchestrator
class Orchestrator(Agent):
    def __init__(self, intent_agent: IntentClassifierAgent, retriever_agent: RetrieverAgent, generator_agent: ResponseGenerator
                 evaluator_agent: EvaluationAgent, escalation_agent: EscalationAgent,
                 session_service: Optional[SessionMemory] = None, metrics: Optional[Dict[str, Any]] = None,
                 llm_client: Optional[LLMClient] = None): # Added llm_client parameter
        super().__init__("orchestrator")
        self.intent_agent = intent_agent
        self.retriever_agent = retriever_agent
        self.generator_agent = generator_agent
        self.evaluator_agent = evaluator_agent
        self.escalation_agent = escalation_agent
        self.session_service = session_service
        self.metrics = metrics

        # All agents should use the same LLM client for consistency and to avoid re-initializing
        self.llm = llm_client # Use the passed llm_client
        if self.llm: # Only assign if llm is present
            self.generator_agent.llm = self.llm
            self.evaluator_agent.llm = self.llm

    def _create_ticket(self, session_id: str, user_message: str, assistant_response: str) -> Dict[str, Any]:
        self.warn("Creating support ticket", session_id=session_id, user_message=user_message)
        # Placeholder for actual ticket creation logic
        return {"ticket_id": f"TICKET-{hash(session_id + user_message)}", "status": "open"}

    def process_message(self, user_message: str, session_id: str = "default_session") -> Dict[str, Any]:
        trace = {"user_message": user_message, "session_id": session_id, "started_at": time.time()}

        if self.session_service:
            self.session_service.append_message(session_id, "user", user_message)
            context_messages = self.session_service.get_context(session_id)
            trace["context_messages"] = context_messages

        # Classifier
        intent = self.intent_agent.classify_intent(user_message)
        trace["intent"] = intent

        # Retrieval
        retrieved_chunks = self.retriever_agent.retrieve(user_message)
        trace["retrieved_chunks"] = retrieved_chunks

        # Generation
```

```
            gen_out = self.generator_agent.generate_answer(intent, user_message, retrieved_chunks)
            trace["generated"] = {"text": gen_out}

            # Evaluation
            try:
                eval_score = self.evaluator_agent.evaluate(user_message, gen_out)
            except Exception as e:
                self.error("Evaluation error", error=str(e))
                eval_score = 3
            trace["evaluation"] = {"score": eval_score}

            # Escalation
            try:
                escalate_flag, escalate_note = self.escalation_agent.check_escalation(eval_score)
            except Exception as e:
                self.error("Escalation check failed", error=str(e))
                escalate_flag = False
                escalate_note = str(e)
            trace["escalation"] = {"escalate": bool(escalate_flag), "note": escalate_note}

            if escalate_flag:
                ticket = self._create_ticket(session_id, user_message, gen_out)
                trace["escalation"]["ticket"] = ticket

            if self.session_service:
                try:
                    self.session_service.append_message(session_id, "assistant", gen_out)
                except Exception as e:
                    self.error("Failed to append assistant message to session", error=str(e))

            if self.metrics:
                try:
                    if isinstance(self.metrics, dict):
                        # simple metrics update
                        self.metrics["total_messages"] = self.metrics.get("total_messages", 0) + 1
                        self.metrics["avg_eval_score"] = ((self.metrics.get("avg_eval_score", 0) * (self.metrics.get("total_message
                        if escalate_flag:
                            self.metrics["total_escalations"] = self.metrics.get("total_escalations", 0) + 1
                except Exception as e:
                    self.error("Metrics update failed", error=str(e))

            trace["completed_at"] = time.time()
            return trace
```

11. Example KB & Index Build

📝 Sample Knowledge Base and Vector Index Construction

In this section, we include a simple example knowledge base and build the FAISS vector index.

This ensures that the file can run end-to-end without external data dependencies and demonstrates the system workflow in a self-contained setup.

```
# Example KB and index build (so file runs end-to-end locally)

KB_DOCS = [
"SaaS product with subscription billing, role-based access, usage analytics.",
"User onboarding flow with health checks and email verification.",
"Support escalation process with L1 → L2 routing.",
"Refunds are issued within 7-14 business days after approval."
]

emb_matrix, kb_chunks = build_kb_embeddings(KB_DOCS)
faiss_index = build_faiss_index(emb_matrix)
```

```
Total chunks: 4
FAISS index built with 4 vectors
```

12. Instantiate Agents & Orchestrator

Initializing All Agents and the System Orchestrator

Here, we create instances of all previously defined agents and assemble them into the orchestrator.

This prepares the entire support desk pipeline for real-time interaction through the Gradio interface.

```
# Instantiate agents & orchestrator

intent_agent = IntentClassifierAgent()
retriever_agent = RetrieverAgent(index=faiss_index, chunks=kb_chunks)
generator_agent = ResponseGeneratorAgent()
evaluator_agent = EvaluationAgent()
escalation_agent = EscalationAgent()

session_service = SessionMemory(max_turns=6)
metrics = {"total_messages": 0, "avg_eval_score": 0.0, "total_escalations": 0}

# Attach a proper LLM client if API key is present; otherwise agents keep heuristic fallbacks
llm_client = LLMClient(model="gemini-1.5-flash") if API_KEY else None
orch = Orchestrator(intent_agent, retriever_agent, generator_agent, evaluator_agent, escalation_agent, session_service=session_
```

13. Gradio UI

Interactive Gradio Interface for User Interaction

This final cell builds and launches the Gradio web interface.

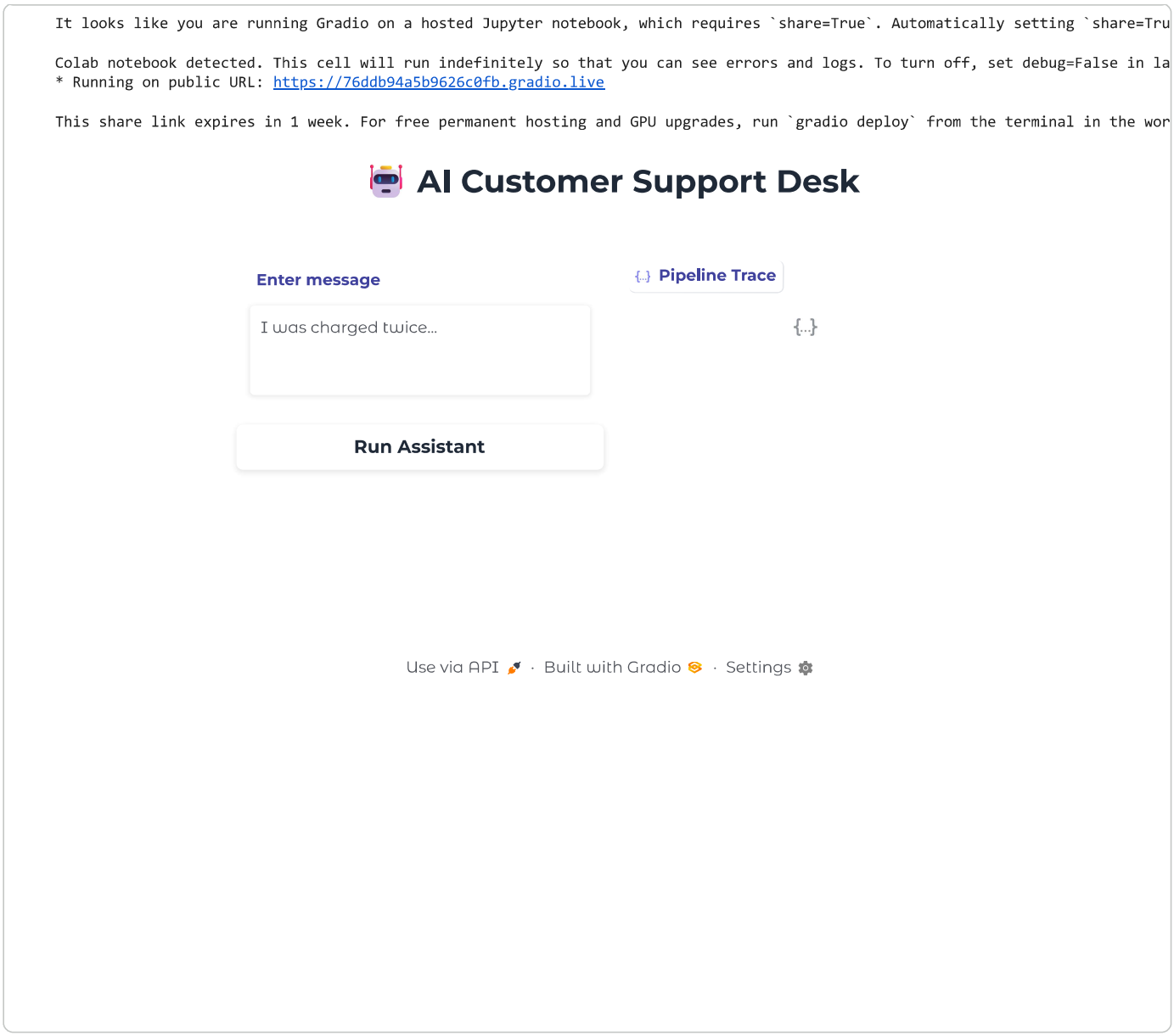
The UI enables users to type queries and view structured outputs including:

- intent category
- retrieved knowledge base snippets
- AI-generated response
- evaluation score
- escalation decision

This delivers a complete, interactive customer support experience.

```
# Gradio UI (simple)

def run_orchestrator_sync(user_message: str):
    try:
        trace = orch.process_message(user_message, session_id="gradio-user")
        return trace
    except Exception as e:
        return {"error": str(e)}

custom_css = """
#title {text-align:center; font-size: 28px; font-weight: bold;}
#inner-box {background:#f7f7f7;padding:20px;border-radius:12px}
"""

with gr.Blocks(css=custom_css, theme=gr.themes.Soft()) as demo:
    gr.HTML("<h2 id='title'>🤖 AI Customer Support Desk</h2>")
    with gr.Column(elem_id="inner-box"):
        with gr.Row():
            with gr.Column(scale=1):
                user_input = gr.Textbox(label="Enter message", placeholder="I was charged twice...", lines=3)
                run_btn = gr.Button("Run Assistant")
            with gr.Column(scale=1):
                output_box = gr.JSON(label="Pipeline Trace")
        run_btn.click(fn=run_orchestrator_sync, inputs=user_input, outputs=output_box)

# Demo launch is guarded so importing this module won't auto-launch in some environments
if __name__ == "__main__":
    demo.launch(debug=True)
```

```
It looks like you are running Gradio on a hosted Jupyter notebook, which requires `share=True`. Automatically setting `share=Tru

Colab notebook detected. This cell will run indefinitely so that you can see errors and logs. To turn off, set debug=False in la
* Running on public URL: https://76ddb94a5b9626c0fb.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the wor
```



Next steps: Deploy to Cloud Run