```
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "# Veterans First AI: Building a RAG-Enabled Conversational Model\n",
    "\n",
    "This notebook provides a complete, end-to-end implementation for building the \"Veterans First\" AI assistant, as detailed in the comprehensive step-by-step guide. We will build a **Retrieval-Augmented Generation (RAG)** system, which grounds a fine-tuned Large Language Model (LLM) in a custom knowledge base of VA-related documents. This ensures the AI's answers are accurate, relevant, and drawn from authoritative sources.\n",
    "\n",
    "### Project Architecture\n",
    "\n",
    "The system follows a RAG pipeline to provide accurate, context-aware answers:\n",
    "\n",
    "1.  **User Query:** A user asks a question through the web interface.\n",
    "2.  **Backend Server:** A Flask server receives the query.\n",
    "3.  **Semantic Search:** The server converts the query into an embedding and searches a FAISS vector database (our knowledge base) to find the most relevant text chunks from VA laws, manuals, and decisions.\n",
    "4.  **Prompt Augmentation:** The retrieved text is combined with the user's original query into a detailed prompt.\n",
    "5.  **LLM Call:** This augmented prompt is sent to our fine-tuned OpenAI model. The fine-tuning helps the model understand the specific domain language, tone, and desired response format.\n",
    "6.  **Response:** The model generates a response grounded in the provided context, which is then sent back to the user.\n",
    "\n",
    ""
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "## Step 0: Setup and Installation\n",
    "\n",
    "First, we'll install all the necessary Python libraries for this project and set up our environment variables. You will need an OpenAI API key for this notebook to work."
```

```
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "# Install required libraries\n",
    "!pip install openai==1.3.3 beautifulsoup4==4.12.2
faiss-cpu==1.7.4 sentence-transformers==2.2.2 flask==3.0.0
flask-cors==4.0.0 pyngrok==7.0.0"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import os\n",
    "from google.colab import userdata\n",
    "\n",
    "# Securely get the OpenAI API key from Colab's secret manager\n",
    "try:\n",
    "    os.environ['OPENAI_API_KEY'] =
userdata.get('OPENAI_API_KEY')\n",
    "    print(\"OpenAI API key loaded successfully.\")\n",
    "except userdata.SecretNotFoundError:\n",
    "    print(\"ERROR: OpenAI API key not found. Please add it to
your Colab secrets.\")\n",
    "    print(\"Go to the '🔑' icon on the left panel and add a new
secret with the name 'OPENAI_API_KEY'.\")"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "## Step 1: Data Collection\n",
    "\n",
    "The foundation of our AI is a comprehensive knowledge base. In
this step, we gather the raw data, including laws, regulations, and
procedure manuals. For this demonstration, we'll create a few sample
text files to simulate this process. A real-world implementation would
involve extensive scraping and downloading from sources like the eCFR,
VA websites, and BVA decision repositories."
   ]
```

```
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import os\n",
    "\n",
    "# Create directories for our project structure\n",
    "os.makedirs('raw_data', exist_ok=True)\n",
    "os.makedirs('processed_data', exist_ok=True)\n",
    "os.makedirs('knowledge_base', exist_ok=True)\n",
    "os.makedirs('application', exist_ok=True)\n",
    "\n",
    "# --- Create Sample Data Files ---\n",
    "\n",
    "sample_cfr_text = \"\"\"\n",
    "Title 38 CFR § 3.303 Principles of service connection.\n",
    "Service connection will be granted if the evidence demonstrates
that a particular injury or disease resulting in disability was
incurred coincident with service in the Armed Forces, or if
pre-existing, was aggravated by service. Service connection connotes
many factors but the main elements are the fact of disease or injury
in service and a nexus between the in-service disease or injury and
the present disability. The veteran's entire service medical record
must be reviewed.\n",
    "\"\"\"\n",
    "with open('raw_data/sample_38cfr.txt', 'w') as f:\n",
    "    f.write(sample_cfr_text)\n",
    "\n",
    "sample_m21_text = \"\"\"\n",
    "M21-1 Adjudication Procedures Manual, Part V, Subpart ii, Chapter
1.\n",
    "The Appeals Modernization Act (AMA) provides veterans with three
options to seek review of a VA decision: (1) Higher-Level Review
(HLR), (2) a Supplemental Claim, or (3) an appeal to the Board of
Veterans' Appeals (BVA). For a Supplemental Claim, the claimant must
submit new and relevant evidence. A Higher-Level Review involves a de
novo review of the issue(s) based on the evidence of record at the
time of the prior decision.\n",
    "\"\"\"\n",
    "with open('raw_data/sample_m21_manual.txt', 'w') as f:\n",
    "    f.write(sample_m21_text)\n",
    "\n",
    "sample_bva_decision = \"\"\"\n",
    "BVA Decision Docket No. 20-01234\n",
    "The veteran seeks service connection for sleep apnea. The service
```

```
treatment records are silent for complaints or treatment for
sleep-disordered breathing. A post-service diagnosis shows moderate
obstructive sleep apnea. However, the private medical opinion fails to
provide a sufficient nexus linking the current condition to the
veteran's active duty service. Therefore, the Board finds that service
connection for sleep apnea is denied.\n",
    "\"\"\"\n",
    "with open('raw_data/sample_bva_decision.txt', 'w') as f:\n",
    "    f.write(sample_bva_decision)\n",
    "\n",
    "print(\"Sample data files created in 'raw_data/' directory.\")"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "## Step 2: Data Preparation\n",
    "\n",
    "Now we process the raw data. This involves two key tasks:\n",
    "1.  **Cleaning and Chunking:** We clean the raw text and split it
into small, logical chunks. These chunks will form the documents in
our searchable knowledge base.\n",
    "2.  **Generating Fine-Tuning Data:** We create high-quality
Question-Answer pairs from the text chunks. This data will be used to
teach our base model the specific style, tone, and format for
answering questions about VA benefits."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import json\n",
    "import re\n",
    "import glob\n",
    "\n",
    "# Configuration for this step\n",
    "RAW_DATA_DIR = 'raw_data'\n",
    "PROCESSED_DATA_DIR = 'processed_data'\n",
    "KNOWLEDGE_BASE_DIR = os.path.join(PROCESSED_DATA_DIR,
'knowledge_chunks')\n",
    "FINETUNE_OUTPUT_FILE = os.path.join(PROCESSED_DATA_DIR,
'finetune_data.jsonl')\n",
    "\n",
    "os.makedirs(KNOWLEDGE_BASE_DIR, exist_ok=True)\n",
```

```
    "\n",
    "def clean_text(text):\n",
    "    \"\"\"Cleans raw text by removing excessive
whitespace.\"\"\"\n",
    "    text = re.sub(r'\\s+', ' ', text).strip()\n",
    "    return text\n",
    "\n",
    "def chunk_text(text, chunk_size=150, overlap=30):\n",
    "    \"\"\"Splits text into smaller, overlapping chunks.\"\"\"\n",
    "    words = text.split()\n",
    "    if not words: return []\n",
    "    chunks = []\n",
    "    for i in range(0, len(words), chunk_size - overlap):\n",
    "        chunk = \" \".join(words[i:i + chunk_size])\n",
    "        chunks.append(chunk)\n",
    "    return chunks\n",
    "\n",
    "def generate_qa_from_chunk(chunk, source):\n",
    "    \"\"\"Generates a plausible Q&A pair from a text
chunk.\"\"\"\n",
    "    first_sentence = chunk.split('.')[0].strip()\n",
    "    if len(first_sentence) < 20 or len(first_sentence) > 200:
return None\n",
    "\n",
    "    if \"§\" in first_sentence:\n",
    "        question = f\"What does the regulation '{first_sentence}'
state?\"\n",
    "    elif \"denied\" in chunk.lower():\n",
    "        question = f\"What is a common reason for denial
mentioned in VA decisions?\"\n",
    "    else:\n",
    "        question = f\"Can you explain the VA process regarding
'{first_sentence}'?\"\n",
    "\n",
    "    answer = chunk\n",
    "    return {\n",
    "        \"messages\": [\n",
    "            {\"role\": \"user\", \"content\": question},\n",
    "            {\"role\": \"assistant\", \"content\": f\"According
to {source}, {answer}\"}\n",
    "        ]\n",
    "    }\n",
    "\n",
    "# --- Main Execution ---\n",
    "finetune_examples = []\n",
    "raw_files = glob.glob(os.path.join(RAW_DATA_DIR, \"*.txt\"))\n",
    "\n",
    "for filepath in raw_files:\n",
```

```
    "    filename = os.path.basename(filepath)\n",
    "    print(f\"Processing {filename}...\")\n",
    "    with open(filepath, 'r', encoding='utf-8') as f:\n",
    "        raw_text = f.read()\n",
    "\n",
    "    cleaned_text = clean_text(raw_text)\n",
    "    text_chunks = chunk_text(cleaned_text)\n",
    "\n",
    "    for i, chunk in enumerate(text_chunks):\n",
    "        chunk_filename =
f\"{os.path.splitext(filename)[0]}_chunk_{i+1}.txt\"\n",
    "        with open(os.path.join(KNOWLEDGE_BASE_DIR,
chunk_filename), 'w', encoding='utf-8') as cf:\n",
    "            cf.write(chunk)\n",
    "\n",
    "    for chunk in text_chunks:\n",
    "        qa_pair = generate_qa_from_chunk(chunk, filename)\n",
    "        if qa_pair:\n",
    "            finetune_examples.append(qa_pair)\n",
    "\n",
    "with open(FINETUNE_OUTPUT_FILE, 'w', encoding='utf-8') as f:\n",
    "    for example in finetune_examples:\n",
    "        f.write(json.dumps(example) + '\\n')\n",
    "\n",
    "print(f\"\\nSuccessfully created {len(finetune_examples)}
fine-tuning examples at: {FINETUNE_OUTPUT_FILE}\")\n",
    "print(f\"Knowledge base chunks saved in: {KNOWLEDGE_BASE_DIR}\")"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "## Step 3: Build the Knowledge Base with Embeddings\n",
    "\n",
    "This is the core of our RAG system. We convert each text chunk
from the previous step into a numerical representation (an
\"embedding\") using an OpenAI model. These embeddings capture the
semantic meaning of the text. We then store them in a FAISS vector
database, which allows for incredibly fast and efficient similarity
searching."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
```

```
   "source": [
    "import faiss\n",
    "import numpy as np\n",
    "import pickle\n",
    "from openai import OpenAI\n",
    "\n",
    "# --- Configuration ---\n",
    "KNOWLEDGE_CHUNKS_DIR = 'processed_data/knowledge_chunks'\n",
    "VECTOR_STORE_DIR = 'knowledge_base'\n",
    "FAISS_INDEX_FILE = os.path.join(VECTOR_STORE_DIR,
'vector_store.index')\n",
    "METADATA_FILE = os.path.join(VECTOR_STORE_DIR,
'metadata.pkl')\n",
    "\n",
    "client = OpenAI()\n",
    "\n",
    "def get_embedding(text, model=\"text-embedding-3-small\"):\n",
    "    \"\"\"Generates an embedding for a given text.\"\"\"\n",
    "    text = text.replace(\"\\n\", \" \")\n",
    "    return client.embeddings.create(input=[text],
model=model).data[0].embedding\n",
    "\n",
    "# --- Main Execution ---\n",
    "chunk_files = glob.glob(os.path.join(KNOWLEDGE_CHUNKS_DIR,
\"*.txt\"))\n",
    "embeddings = []\n",
    "metadata = []\n",
    "\n",
    "print(f\"Creating embeddings for {len(chunk_files)} text
chunks...\")\n",
    "for filepath in chunk_files:\n",
    "    with open(filepath, 'r', encoding='utf-8') as f:\n",
    "        text_content = f.read()\n",
    "    if text_content.strip():\n",
    "        embedding_vector = get_embedding(text_content)\n",
    "        embeddings.append(embedding_vector)\n",
    "        metadata.append({'content': text_content, 'source':
os.path.basename(filepath)})\n",
    "\n",
    "embedding_matrix = np.array(embeddings).astype('float32')\n",
    "d = embedding_matrix.shape[1] # Dimension of embeddings\n",
    "index = faiss.IndexFlatL2(d)\n",
    "index.add(embedding_matrix)\n",
    "\n",
    "print(f\"Saving FAISS index to {FAISS_INDEX_FILE}...\")\n",
    "faiss.write_index(index, FAISS_INDEX_FILE)\n",
    "\n",
    "print(f\"Saving metadata to {METADATA_FILE}...\")\n",
```

```
    "with open(METADATA_FILE, 'wb') as f:\n",
    "    pickle.dump(metadata, f)\n",
    "\n",
    "print(f\"\\nKnowledge base created with {index.ntotal}
documents.\")"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "## Step 4: Fine-Tune the Model\n",
    "\n",
    "With our knowledge base built, we now fine-tune a base model
(like `gpt-3.5-turbo`) using the Q&A pairs we generated. This doesn't
teach the model new facts—the RAG system handles that. Instead, it
teaches the model the *style* of a helpful VA claims expert: how to
structure answers, what tone to use, and how to refer to sources. This
makes the model a better \"reasoning engine\" for our specific
task.\n",
    "\n",
    "**This step will create a fine-tuning job on OpenAI's servers. It
can take some time to complete (from minutes to hours depending on
data size). You will need to copy the final model ID for the next
step.**"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import time\n",
    "\n",
    "# --- Configuration ---\n",
    "FINETUNE_DATA_FILEPATH = 'processed_data/finetune_data.jsonl'\n",
    "MODEL_SUFFIX = 'veterans-first-ai-colab'\n",
    "\n",
    "# --- Main Execution ---\n",
    "print(f\"Uploading training file: {FINETUNE_DATA_FILEPATH}\")\n",
    "with open(FINETUNE_DATA_FILEPATH, \"rb\") as f:\n",
    "    training_file = client.files.create(file=f,
purpose=\"fine-tune\")\n",
    "print(f\"File uploaded successfully. File ID:
{training_file.id}\")\n",
    "\n",
    "print(f\"\\nCreating fine-tuning job...\")\n",
```

```
    "job = client.fine_tuning.jobs.create(\n",
    "    training_file=training_file.id,\n",
    "    model=\"gpt-3.5-turbo\",\n",
    "    suffix=MODEL_SUFFIX,\n",
    ")\n",
    "print(f\"Fine-tuning job created successfully. Job ID:
{job.id}\")\n",
    "print(\"You can monitor the job's progress on the OpenAI
website.\")\n",
    "\n",
    "# --- Optional: Monitor job progress in Colab ---\n",
    "print(\"\\nMonitoring job status (press Ctrl+C to stop)...\")\n",
    "try:\n",
    "    while True:\n",
    "        job_status = client.fine_tuning.jobs.retrieve(job.id)\n",
    "        status = job_status.status\n",
    "        print(f\"Current status: {status}\")\n",
    "        if status == \"succeeded\":\n",
    "            fine_tuned_model_id = job_status.fine_tuned_model\n",
    "            print(f\"\\n✅ Fine-tuning succeeded! Your model ID
is: {fine_tuned_model_id}\")\n",
    "            print(\"➡️ PLEASE COPY THIS ID. You will need it for
the next step.\")\n",
    "            break\n",
    "        elif status in [\"failed\", \"cancelled\"]:\n",
    "            print(f\"\\n❌ Job {status}. Check the OpenAI
dashboard for details.\")\n",
    "            break\n",
    "        time.sleep(60)\n",
    "except KeyboardInterrupt:\n",
    "    print(\"\\nMonitoring stopped. You can check the job status
on the OpenAI website.\")"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "## Step 5 & 6: Run the Backend and Web Interface\n",
    "\n",
    "Now we'll combine everything into a functional application. The
code below does the following:\n",
    "1.  **Defines a Flask Backend:** This is a lightweight web server
that will load our FAISS knowledge base and handle the full RAG
pipeline.\n",
    "2.  **Creates a Public URL:** We use `pyngrok` to create a
secure, public URL for our Flask app, allowing our web interface to
access it from within Colab.\n",
```

```
    "3.  **Launches the Web UI:** We embed the HTML and JavaScript for
our chat interface directly into the notebook. The JavaScript is
configured to communicate with the `ngrok` URL of our backend.\n",
    "\n",
    "**IMPORTANT:** Paste the fine-tuned model ID you copied from the
previous step into the `FINE_TUNED_MODEL_ID` variable in the code cell
below."
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "from flask import Flask, request, jsonify\n",
    "from flask_cors import CORS\n",
    "from pyngrok import ngrok\n",
    "from IPython.display import display, HTML\n",
    "import threading\n",
    "\n",
    "# --- PASTE YOUR FINE-TUNED MODEL ID HERE ---\n",
    "FINE_TUNED_MODEL_ID = \"ft:gpt-3.5-turbo...\" # <-- REPLACE
THIS\n",
    "\n",
    "# --- Flask Backend Application ---\n",
    "app = Flask(__name__)\n",
    "CORS(app)\n",
    "\n",
    "# Load knowledge base\n",
    "rag_index = faiss.read_index(FAISS_INDEX_FILE)\n",
    "with open(METADATA_FILE, 'rb') as f:\n",
    "    rag_metadata = pickle.load(f)\n",
    "\n",
    "@app.route('/chat', methods=['POST'])\n",
    "def chat():\n",
    "    data = request.json\n",
    "    user_query = data.get('message')\n",
    "\n",
    "    # 1. Retrieve context\n",
    "    query_embedding =
np.array([get_embedding(user_query)]).astype('float32')\n",
    "    distances, indices = rag_index.search(query_embedding,
k=3)\n",
    "    retrieved_context = [rag_metadata[i]['content'] for i in
indices[0]]\n",
    "    context_str = \"\\n\\n---\\n\\n\".join(retrieved_context)\n",
    "\n",
```

```
    "    # 2. Construct prompt\n",
    "    system_prompt = {\n",
    "        \"role\": \"system\",\n",
    "        \"content\": f\"You are the Veterans First AI. Answer the
user's question based only on the provided
context.\\nCONTEXT:\\n{context_str}\"\n",
    "    }\n",
    "    messages = [system_prompt, {\"role\": \"user\", \"content\":
user_query}]\n",
    "\n",
    "    # 3. Call fine-tuned model\n",
    "    response = client.chat.completions.create(\n",
    "        model=FINE_TUNED_MODEL_ID,\n",
    "        messages=messages,\n",
    "        temperature=0.3\n",
    "    )\n",
    "    return jsonify({\"response\":
response.choices[0].message.content})\n",
    "\n",
    "def run_app():\n",
    "    app.run(port=5000)\n",
    "\n",
    "# --- Launch Backend and UI ---\n",
    "if FINE_TUNED_MODEL_ID == \"ft:gpt-3.5-turbo...\":\n",
    "    print(\"ERROR: Please paste your fine-tuned model ID into the
'FINE_TUNED_MODEL_ID' variable.\")\n",
    "else:\n",
    "    # Start flask app in a new thread\n",
    "    flask_thread = threading.Thread(target=run_app)\n",
    "    flask_thread.daemon = True\n",
    "    flask_thread.start()\n",
    "\n",
    "    # Expose the flask app with ngrok\n",
    "    public_url = ngrok.connect(5000).public_url\n",
    "    print(f\"Backend is running at: {public_url}\")\n",
    "\n",
    "    # --- HTML and JavaScript for the Frontend ---\n",
    "    html_template = f\"\"\"\n",
    "    <!DOCTYPE html>\n",
    "    <html lang=\"en\">\n",
    "    <head>\n",
    "        <meta charset=\"UTF-8\">\n",
    "        <meta name=\"viewport\" content=\"width=device-width,
initial-scale=1.0\">\n",
    "        <title>Veterans First AI Assistant</title>\n",
    "        <script src=\"https://cdn.tailwindcss.com\"></script>\n",
    "        <style>body {{ font-family: sans-serif; }}</style>\n",
    "    </head>\n",
```

```
    "    <body class=\"bg-slate-100\">\n",
    "        <div class=\"bg-white shadow-lg rounded-lg max-w-4xl
mx-auto my-8 flex flex-col h-[80vh]\">\n",
    "            <header class=\"bg-slate-800 text-white p-4
rounded-t-lg\"><h1 class=\"text-xl font-bold\">Veterans First AI
Assistant</h1></header>\n",
    "            <main id=\"chat-container\" class=\"flex-1 p-6
overflow-y-auto\"></main>\n",
    "            <footer class=\"p-4 bg-white border-t\">\n",
    "                <form id=\"chat-form\" class=\"flex items-center
space-x-4\">\n",
    "                    <input type=\"text\" id=\"user-input\"
placeholder=\"Ask a question...\" class=\"w-full px-4 py-2 border
rounded-lg\">\n",
    "                    <button type=\"submit\" id=\"send-btn\"
class=\"bg-blue-600 text-white px-6 py-2 rounded-lg
font-semibold\">Send</button>\n",
    "                </form>\n",
    "            </footer>\n",
    "        </div>\n",
    "        <script>\n",
    "            const BACKEND_URL = '{public_url}';\n",
    "            const chatForm =
document.getElementById('chat-form');\n",
    "            const userInput =
document.getElementById('user-input');\n",
    "            const chatContainer =
document.getElementById('chat-container');\n",
    "\n",
    "            function appendMessage(role, content) {{\n",
    "                const messageDiv =
document.createElement('div');\n",
    "                const alignment = role === 'user' ? 'text-right'
: 'text-left';\n",
    "                const bubbleColor = role === 'user' ?
'bg-blue-500 text-white' : 'bg-slate-200 text-slate-800';\n",
    "                messageDiv.className = `my-2 ${alignment}`;\n",
    "                messageDiv.innerHTML = `<div class=\"inline-block
p-3 rounded-lg ${bubbleColor}\">${content}</div>`;\n",
    "                chatContainer.appendChild(messageDiv);\n",
    "                chatContainer.scrollTop =
chatContainer.scrollHeight;\n",
    "            }}\n",
    "\n",
    "            chatForm.addEventListener('submit', async (e) =>
{{\n",
    "                e.preventDefault();\n",
    "                const userMessage = userInput.value.trim();\n",
```

```
    "                if (!userMessage) return;\n",
    "                appendMessage('user', userMessage);\n",
    "                userInput.value = '';\n",
    "\n",
    "                const response = await
fetch(`${BACKEND_URL}/chat`, {{\n",
    "                    method: 'POST',\n",
    "                    headers: {{ 'Content-Type':
'application/json' }},\n",
    "                    body: JSON.stringify({{ message: userMessage
}})\n",
    "                }});\n",
    "                const data = await response.json();\n",
    "                appendMessage('assistant', data.response);\n",
    "            }});\n",
    "        </script>\n",
    "    </body>\n",
    "    </html>\n",
    "    \"\"\"\n",
    "    # Display the frontend\n",
    "    display(HTML(html_template))"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "## Step 7: Evaluation and Maintenance\n",
    "\n",
    "Building the model is just the beginning. Continuous evaluation
and maintenance are crucial for ensuring the AI remains accurate and
helpful over time.\n",
    "\n",
    "### Evaluation Plan\n",
    "\n",
    "1.  **Functional Testing:** Test a wide range of queries covering
the entire claims process, from initial filing to appeals. Verify that
the RAG system retrieves the correct context and the final answers are
factually accurate.\n",
    "2.  **User Scenario Testing:** Simulate multi-turn conversations
to assess the AI's ability to maintain context and guide users through
complex scenarios.\n",
    "3.  **Accuracy Checks:** Involve Subject Matter Experts (SMEs) to
review generated answers for correctness and clarity. Measure system
performance, such as response time and retrieval quality.\n",
    "\n",
    "### Maintenance Strategy\n",
    "\n",
```

```
    "1.  **Regular Data Updates:** The world of VA benefits changes. Schedule regular updates (e.g., quarterly) to the knowledge repository by re-running the data collection and knowledge base creation scripts with new laws, manuals, and BVA decisions.\n",
    "2.  **Monitor Performance and Feedback:** Implement a feedback mechanism (like thumbs-up/down ratings on answers) to identify areas for improvement. Use this feedback to expand the knowledge base or add new examples to the fine-tuning dataset.\n",
    "3.  **Ethical Compliance:** Always include a disclaimer that the AI is an informational tool and not a substitute for legal advice. Ensure user privacy is protected and that the AI's advice is never harmful."
   ]
  }
 ],
 "metadata": {
  "colab": {
   "collapsed_sections": [],
   "name": "Veterans_First_AI_Notebook.ipynb",
   "provenance": []
  },
  "kernelspec": {
   "display_name": "Python 3",
   "name": "python3"
  },
  "language_info": {
   "name": "python"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 0
}
```