**ITAI 2377 Midterm Assignment: Domain-Specific AI Assistant Project Plan**

**Data Requirements Analysis**

**Data Types:**

* Unstructured text (manuals, FAQs)

This includes raw text extracted from user manuals, troubleshooting guides, or community discussions. These documents do not follow a consistent structure but contain valuable explanations, instructions, and error scenarios that the assistant can learn from or retrieve during support interactions.

* Structured (device metadata)

Think of things like model numbers, firmware versions, connectivity protocols (e.g., Zigbee, Wi-Fi), and brand associations. This type of information is tabular and machine-readable, helping the assistant filter, match, or prioritize the right instructions for a given device context.

* Semi-structured (JSON configurations)

These are configuration templates, API responses, or diagnostic logs formatted with predictable keys and values. For instance, a JSON blob showing device pairing status or system logs can be analyzed to detect patterns and suggest next steps.

**Sources:**

* Open-source device manuals

PDFs or HTML docs are crafted publicly available by vendors or through crowd-sourced archives. These are the backbone of the assistant’s knowledge—covering setup steps, indicators, troubleshooting paths, and safety warnings.

* Forum FAQs (e.g., Reddit, SmartThings)

Community-driven answers to common problems. These discussions surface real-world phrasing and unexpected scenarios that do not always appear in formal manuals, making them excellent material for edge case training or retrieval.

* Public vendor documentation

Manufacturer help pages or official API references, often updated more frequently and written in a clearer tone. These are especially useful for keeping current with firmware changes or new product features

* Synthetic Q&A pairs for edge cases

Manually crafted examples to simulate niche queries or fill gaps in the existing data—like simulating a user asking, “Why does my thermostat reboot when it hits 78 degrees?” This helps the assistant respond more reliably to uncommon but critical issues.

**Volume & Velocity:**

Initial set: ~100 devices

A manageable scope to start with, focusing on popular product categories like thermostats, smart plugs, bulbs, and security cameras across top brands.

~50KB–1MB text per device

Each device might have multiple manuals or versions depending on model year. This estimate accounts for compressed or extracted text chunks from each resource.

Manual refresh quarterly or with firmware versions

- Update frequency (quarterly/manual refresh):

Since smart home products receive firmware and hardware updates periodically, refreshing the dataset every few months—or when version logs change—is ideal for staying relevant.

**Data Quality:**

* Accurate product-version matching

Ensure users with a 2022 smart bulb get guidance relevant to that exact model, not an outdated or unrelated version.

* Cleaned text (no OCR errors, redundant sections)

PDFs often introduce artifacts or duplicate content during extraction. Cleaning steps remove broken characters, irrelevant sections (e.g., legal disclaimers), and repetitive wording.

* Labeled with device type, brand, firmware

Tagging each document chunk allows the assistant to retrieve only the most relevant materials for the user’s query—improving both accuracy and speed.

**Challenges:**

* Inconsistent formats across vendors

Every manufacturer has their own way of writing and structuring documentation. Some publish clean HTML guides, others share cluttered PDFs with awkward layouts or split manuals into multiple, redundant files. This makes it tricky to extract and process the content in a standardized way.

* Ambiguity in model numbers

It is often-unclear which document applies to which version of a device. A smart plug labeled "V2 Plus" might refer to three different product SKUs depending on region, release year, or firmware. Without consistent labeling, the assistant risks pulling advice that does not fit the user’s specific model

* Unstructured, verbose language in manuals

Manuals are c for humans, not machines, so they include lots of fillers, legal disclaimers, overly wordy explanations, and scattered formatting. Key troubleshooting steps can get obscured in walls of text, making it harder to extract clean, usable chunks for the assistant to work with.

**Data Schema:**  
TBD??

**Processing Pipeline Design**

**Workflow:**

1. Scrape/ingest documentation

| Tool | Purpose |
| --- | --- |
| BeautifulSoup + ‘requests’ | Web scraping of HTML documentation and FAQs |
| PyMuPDF (fitz) or ‘pdfplumber’ | Extract clean text from user manuals in PDF format |
| Selenium | Headless browsing for JavaScript-heavy vendor sites |
| Google Drive / GitHub | Storing scraped docs or uploading sample files manually |

1. Clean and chunk text by topic

| Tool | Purpose |
| --- | --- |
| re (regex) + ‘nltk’ / ‘spaCy’ | Basic cleanup: remove symbols, headers, unwanted sections |
| LangChain ‘TextSplitters’ | Smart chunking of long docs into semantic sections |
| custom Python scripts | Topic labeling or section title detection based on headings |

1. Generate embeddings (e.g., using sentence-transformers)

| **Tool** | **Purpose** |
| --- | --- |
| sentence-transformers (all-MiniLM-L6-v2, multi-qa-MiniLM) | Embedding generation from text chunks |
| Hugging Face Transformers | If you want more control or to use different models |
| OpenAI Embedding API | Alternative for proprietary embedding models (paid usage) |

1. Store in vector store (e.g., FAISS, ChromaDB)

| Tool | Purpose |
| --- | --- |
| FAISS | Local, fast approximate nearest-neighbor search (great for Colab/local) |
| ChromaDB | Simple, developer-friendly vector DB with metadata filters |
| Weaviate / Pinecone (optional) | Cloud-hosted alternatives for scaling and persistent storage |

1. Build retrieval + prompt template pipeline

| Tool | Purpose |
| --- | --- |
| LangChain | Manage retrieval, context assembly, and model invocation logic |
| LlamaIndex (formerly GPT Index) | Alternative for building structured RAG systems |
| Jinja2 / f-strings | For building prompt templates manually without a framework |
| OpenAI / Hugging Face API | Language model backend for generating the final response |

**Feature Engineering:**

* Chunk context (title, device name, version, section)
* Query embeddings + metadata filters (Tag each chunk with device\_name, version, section\_title, connectivity\_protocol, etc.)

**Data Transformations:**

* Text: normalization, section splitting, version tagging (with nltk or spaCy)
* Metadata: standardized labels (e.g., device\_type = "thermostat") – manual mapping or JSON schema

**Infrastructure Considerations:**

* Works entirely in Colab or local dev for MVP (no install, easy GPU access)
* Lightweight vector store and JSON-based retrieval system

**Implementation Strategy**

**Tech Stack:**

* Colab + Python
* FAISS or Chroma for embeddings
* HuggingFace transformers or OpenAI API
* Streamlit/Gradio for demo UI

**Timeline:**

* Week 1–2: Data collection & preprocessing
* Week 3: Embedding + retrieval setup
* Week 4–5: Assistant logic + UI
* Week 6: Testing, feedback, polish

**Team Roles:**  
(If working solo, these can reflect development phases)

* Data ingestion & cleaning
* Retrieval logic / pipeline
* Front-end / demo scripting

**Resources:**

* LLM API access
* GitHub repo for versioning
* Google Drive for storage
* Device manuals dataset
* - Streamlit / Gradio account – for deployment or team preview access
* - Notion / Google Docs – for shared task planning, feature tracking, and meeting notes
* - VS Code + GitHub Codespaces (optional) – for local editing if Colab feels limiting

**Implementation Risks:**

* Knowledge gaps for niche devices
* Prompt sensitivity
* Inference latency in demo

**Integration Plan:**  
EX: User query → Query vector (embedding)

→ Vector DB search (FAISS/Chroma)

→ Top-k document chunks

→ Prompt assembly

→ LLM response generation

→ Final assistant reply

**Evaluation Framework**

**Success Metrics:**

* ≥ Eighty percent match accuracy for known queries

The assistant should correctly retrieve and respond to at least eight out of ten test queries based on its knowledge base. These are predefined examples tied to specific documentation chunks to evaluate how well the retrieval logic and embeddings are supported.

* User satisfaction in test prompts

Collect subjective feedback from teammates or test users on how helpful, understandable, and complete the assistant’s responses are. A short feedback form or 1–5 star rating scale can help quantify satisfaction.

* Response latency under 5 seconds

The system should return answers quickly enough to feel responsive. Five seconds is a reasonable ceiling for latency during retrieval + generation on the MVP. Anything longer could feel sluggish in real use.

**Testing Strategy:**

* Unit tests for retrieval

Ensure that queries return the most relevant chunks from the vector store. You can write tests for known inputs and expected top-k outputs to catch precision and recall issues.

* Scenario-based testing (e.g., “my smart bulb blinks purple”)

Try real-world prompts like “Why isn’t my thermostat showing up in the app?” or “My smart bulb blinks purple—what does that mean?” This helps simulate actual user phrasing and ambiguity.

* Manual validation of top K responses

For a sample of user queries, manually inspect whether the top returned documentation chunks are truly relevant. This helps calibrate chunking granularity and embedding fidelity

**User Feedback Plan:**

* Collect inputs on unclear responses

Use a feedback form, spreadsheet, or checkbox system to flag responses that felt vague, irrelevant, or overly technical. You can also include a “Was this helpful?” toggle in your UI if time allows.

* Adjust prompts or add context chunks

When feedback reveals misunderstanding, you can tweak your prompt templates, add clarifying language, or include more metadata in your retrieved chunks to give the LLM richer context.

**Benchmarks:**

* Model should resolve common issues faster than help article search

The assistant should be able to resolve widespread support questions more quickly—and with more direct answers—than searching through manufacturer websites or PDF manuals.

**Improvement Loop:**

* Monitor failed queries

Log user queries that lead to poor or “I don’t know” responses. This reveals where documentation is lacking or where retrieval broke down.

* Expand dataset and refine chunking

You can re-chunk long docs more intelligently or embed new user-generated examples to improve coverage of less common queries.

* Add new device classes based on trends

If testers frequently ask about cameras or doorbells that are not in the dataset yet, those should become top priorities for the next phase.

** Installs Dependencies**:

* + sentence-transformers: For generating semantic embeddings.
  + PyYAML: For YAML parsing (though not used directly here).
  + chromadb: Installed but not yet used—for future vector database integration.

1. **Loads Sample Data**:
   * Two documents: one YAML automation rule, one FAQ-style explanation.
   * Two user queries that semantically relate to the documents.
2. **Embeds Documents**:
   * Uses the paraphrase-MiniLM-L6-v2 model to convert document content into dense vector embeddings.
3. **Retrieves Best Match**:

* For each query, it computes cosine similarity between the query embedding and all document embeddings.