**Doc-Scanner**

**🧩 Key Components Involved**

Let’s first remember what’s involved:

* ✅ **Flask App** – User interface & file upload
* 📄 **Uploaded File** – Document to review
* ✂️ **LangChain** – Splits text into small parts
* 🧮 **Google Embedding-001** – Turns each part into numbers
* 🗃️ **ChromaDB** – Stores those numbers + chunks
* 🔍 **Retriever** – Finds similar past content
* 🤖 **Gemini LLM** – Gives suggestions based on that
* 🧠 **GeminiAISuggestionEngine** – Your custom logic for all this

**🔁 Sequence: What Happens When You Upload a File**

**1️⃣ You Upload a Document**

* You drag and drop or choose a document (PDF, DOCX, etc.)
* Flask receives the file

**2️⃣ Text is Extracted**

* The app reads the document and **extracts plain text**
  + python-docx, PyPDF2, or BeautifulSoup might be used here depending on file type

**3️⃣ Text is Split into Chunks**

* LangChain **splits** the text into small parts (sentences or paragraphs)
  + This is so the AI can analyze and compare small meaningful chunks
  + It may use RecursiveCharacterTextSplitter or SentenceSplitter

**4️⃣ Embeddings Are Generated**

* Each chunk is passed to **Google's Embedding-001 model**
* It converts each chunk into a list of numbers (vector) that **represents its meaning**

**5️⃣ Embeddings + Chunks Are Stored in ChromaDB**

* ChromaDB saves:
  + The chunk (text)
  + The embedding (vector)
  + Metadata (like filename or section)

This makes your document searchable for future comparisons.

**6️⃣ RAG Starts: Review + Suggestion Begins**

* When reviewing a sentence:
  + A **retriever** checks ChromaDB for similar content
  + These "similar chunks" are passed to Gemini 1.5 Flash

**7️⃣ Gemini LLM Gives Suggestions**

* Gemini sees:
  + The sentence to review
  + Similar examples from your document (retrieved from ChromaDB)
* It gives a **smart, context-aware suggestion** or comment

**8️⃣ Custom Logic Wraps It Up**

* Your GeminiAISuggestionEngine class combines all the parts:
  + Formatting
  + Style rule checks
  + Suggestion formatting
  + HTML rendering for the UI

**9️⃣ Flask App Displays the Feedback**

* The user sees:
  + Original sentence
  + Suggestion from Gemini
  + Explanation (if any)
  + Option to accept, ignore, or revise

**✅ Bonus: If You Upload More Files**

* More files → More data chunks + embeddings → ChromaDB grows
* Now Gemini can pull smarter suggestions using a **larger memory of your writing style**
* Your app keeps getting "contextually smarter" over time!

**🔁 Visual Summary (Simple Flow)**

User uploads file

↓

Text is extracted

↓

Text is chunked

↓

Embeddings are generated

↓

Chunks + embeddings stored in ChromaDB

↓

Retriever finds similar content

↓

Gemini generates review/suggestions

↓

Flask shows them on the UI

**AI Agent** = You define **what you want the it to achieve**, and the **agent figures out how to do it**.

**The AI Agent Workflow:**

1. **You set the goal**

e.g., “Improve the quality of user manuals.”

1. **The agent observes the environment**

e.g., It reads the document, checks your writing style, context, terminology, etc.

1. **It makes decisions**

e.g., “This sentence is unclear,” or “This doesn't follow the style guide.”

1. **It takes actions**

e.g., Gives you suggestions, highlights issues, rephrases text, or summarizes feedback.

1. **It learns and improves** (if learning is enabled)

It remembers your preferences and adapts next time.

**RAG = Retrieval-Augmented Generation**

It’s a technique where a **language model (like GPT or Gemini)** is combined with a **knowledge retrieval system** (like a search engine or vector database) to give better, more accurate, and up-to-date answers.

**How RAG Works (Step-by-Step)**

1. **User asks a question**  
   ➤ *“What’s the latest style rule for headings?”*
2. **Retriever searches your data sources**  
   ➤ It finds relevant documents, PDFs, or internal manuals using keywords or vector similarity.
3. **Generator (LLM) uses that info to answer**  
   ➤ It reads the retrieved content and writes a smart, context-aware answer.

| **Advantages of RAG**   * LLM uses **real, up-to-date info** from your own documents * Gives fact based answers, do not hallucinate * Can be customized to your company, project, or manuals |
| --- |

**NLP = Natural Language Processing**

So, **NLP logic** means:

The **rules**, **algorithms**, or **AI models** your app uses to **understand, process, and analyze human language** (text or speech).

It’s the “brain” of your app that works with language.

**🛠️ What Does NLP Logic Include?**

Here are some common elements:

| **NLP Function** | **What It Does** | **Example** |
| --- | --- | --- |
| **Tokenization** | Breaks text into words or sentences | “This is good.” → 4 tokens |
| **Part-of-speech tagging** | Identifies words as noun, verb, etc. | “run” → verb or noun |
| **Named Entity Recognition (NER)** | Finds names, dates, places | “Google was founded in 1998” |
| **Parsing / Dependency Tree** | Understands how words are related | Links subject–verb–object |
| **Custom Rule Matching** | Applies your grammar/style rules | Flag "is being done" as passive |
| **Text classification** | Categorizes text | Label a sentence as "unclear" |