**Documentation of Process**

**1.Overview**

This project implements an intelligent email assistant that automates the processing of Gmail messages using a large language model (LLM). The assistant performs the following key functions:

* Fetches and processes emails from Gmail.
* Summarizes email content.
* Detects meeting requests and schedules events via Google Calendar.
* Generates professional reply suggestions.
* Performs web searches for context enhancement.
* Sends Slack notifications for important emails.
* Automatically replies to safe-to-handle messages.
* Logs all processed emails in a local SQLite database.

**2.Tools and Technologies**

| **Component** | **Technology / Service** |
| --- | --- |
| Email Access | Gmail API (googleapiclient.discovery) |
| Calendar Integration | Google Calendar API |
| Natural Language Model | Groq (Gemma2-9b-It) via LangChain |
| Prompt Chaining | LangChain LLMChain and PromptTemplate |
| Notifications | Slack Bot API |
| Web Search Integration | Google Custom Search API |
| Database Storage | SQLite (via Python's sqlite3 module) |
| Authentication | Google OAuth 2.0 via google-auth-oauthlib |

**3.Workflow Overview**

1. **Authentication**
   * Utilizes OAuth2.0 to authenticate Gmail and Calendar access.
   * Token is saved as token.json for reuse.
   * Auto-refreshes token when expired.
2. **Fetching Emails**
   * Retrieves messages using the Gmail API.
   * Parses message metadata and extracts plain text body.
3. **Processing Emails**
   * **Summarization**: Email body is summarized using Groq's LLM.
   * **Reply Suggestion**: Generates a professional reply draft.
   * **Meeting Detection**: Checks for meeting requests via LLM classification.
   * **Meeting Scheduling**: If applicable, extracts structured details (date, time, attendees) and creates a Google Calendar event.
   * **Slack Notification**: Sends alert if keywords indicating urgency or importance are detected.
   * **Web Search**: If the email requires additional information, it performs a Google Custom Search and appends results to the reply.
   * **Auto-reply**: Automatically replies to emails deemed safe by the LLM.
   * **Database Logging**: Saves processed email metadata to a local SQLite database for future reference.

**4.Prompt Design**

| **Function** | **Prompt (Template)** |
| --- | --- |
| Summarization | "Summarize the following email:\n{text}" |
| Reply Suggestion | "Generate a professional reply to this email:\n{text}" |
| Meeting Detection | "Does this email contain a request to schedule a meeting? Answer with 'Yes' or 'No'.\n{text}" |
| Meeting Detail Extraction | "Extract meeting details from this email (date, time, title, attendees) and return as JSON.\n{text}" |
| Web Search Decision | "Does this email require searching the web for information? Answer 'Yes' or 'No'.\n{text}" |
| Auto-Reply Safety Classification | "Can this email be safely auto-replied to with minimal human oversight? Reply with 'Yes' or 'No'.\n\nEmail:\n{text}" |

**5.Design Decisions**

* **Language Model Selection**: Groq LLM was selected for its fast response times and reliable output format when used with LangChain.
* **Chunking Strategy**: Emails are split into manageable chunks using RecursiveCharacterTextSplitter to ensure LLM context window limits are respected.
* **Urgency Detection**: Uses a static keyword list to determine high-priority messages and trigger Slack notifications.
* **Data Persistence**: A lightweight SQLite database is used to store all processed email metadata.
* **Web Search Integration**: Custom Google Search Engine allows context-aware information enrichment in replies.

**6.Assumptions**

* Emails contain a text/plain MIME part. HTML parsing is not yet implemented.
* Meeting information is consistently extractable as structured JSON using the LLM.
* Important messages include commonly known urgency-related keywords.
* Calendar events are created with UTC timezone (future enhancement could include dynamic timezone detection).
* OAuth credentials are securely stored locally for development use.

**7.Challenges Faced During Development and Their Solutions**

**1. Google API Authentication (OAuth 2.0)**

**Challenge:**  
Frequent authentication prompts or token errors, especially when switching devices or re-running the app after some time.

**Why it happened:**  
Token expiration or missing token.json caused the app to fail silently or repeatedly ask for re-authentication.

**Solution:**

* Implemented persistent token storage (token.json) and checked for validity before each use.
* Added logic to auto-refresh tokens using creds.refresh(Request()).
* Made the flow fail-safe: if the token is invalid or missing, the user is prompted via browser only once.

**2. Rate Limits (Google Search + Gmail)**

**Challenge:**  
Frequent testing or looped executions quickly exhausted free quotas or triggered temporary blocks.

**Why it happened:**  
Repeated API hits (e.g., every email checking for meeting details + web search) overloaded Google limits.

**Solution:**

* Limited fetched emails (maxResults=5) during development.
* Added conditional logic: search Google **only if** the LLM flagged it as needed.
* Future fix: caching, backoff timers, or batch processing of emails.

**3. Over-Triggering Auto-Replies**

**Challenge:**  
The system sometimes sent replies even when emails weren’t actually appropriate for auto-responding (e.g., spam, newsletters).

**Why it happened:**  
LLM misclassified emails as safe for auto-reply based on superficial text.

**Solution:**

* Added a is\_safe\_for\_autoreply() LLM step to gate auto-replies.
* Will explore email classification using metadata (e.g., thread depth, known contacts) in future versions.

**4. Slow Performance for Long Emails**

**Challenge:**  
Processing long emails (especially multi-part threads) caused delays during LLM summarization and reply generation.

**Why it happened:**  
LangChain’s splitter handles up to 2000 characters at once, but LLM calls still take time.

**Solution:**

* Used RecursiveCharacterTextSplitter to chunk long bodies.
* Currently processes only the first chunk (most relevant part).
* Future plan: batch summarize or use preview heuristics.

**8.Use of AI Coding Assistants**

During the development of this intelligent email assistant, AI coding tools like **ChatGPT** were extensively used to streamline and enhance the development process. ChatGPT significantly reduced the time spent on debugging, writing boilerplate code, and handling complex integrations. It was particularly useful in planning the architecture, designing prompt templates for summarization, meeting detection, and auto-reply generation using LLMs. ChatGPT helped refine JSON extraction prompts to ensure reliable formatting and accurate data parsing. Challenges such as OAuth2 authentication issues, token refresh errors, MIME decoding problems in Gmail messages, and Slack API permission errors were quickly resolved with ChatGPT’s assistance, which provided relevant code snippets and configuration tips. It also guided improvements in logic such as adding safety checks with try/except for json.loads, detecting urgent email keywords, and conditionally triggering web searches to reduce unnecessary API calls. Additionally, ChatGPT recommended best practices like modular function design and clean error handling. These interactions not only accelerated development but also ensured code quality and robustness. Overall, ChatGPT acted as a virtual coding partner, offering real-time solutions and contextual reasoning that helped overcome common development hurdles efficiently. The use of ChatGPT alone is estimated to have reduced the total development and debugging time by approximately 30–40%, making it an invaluable tool in building this production-ready assistant.