System Design Document: Campus Companion

Name: Aviral Saxena
Branch: CSE
College: IIT Bhilai

1. Introduction:

The Problem: The Student's "Digital Overload"

In today's hyper-connected university environment, students are inundated with a relentless firehose of information. Critical opportunities for career-defining internships, skill-building workshops, hackathons, competitive programming contests and essential academic deadlines are **scattered across a chaotic digital landscape** of department **websites, club emails, and PDF flyers.** The manual effort required to track, filter, and organize this information is not just a chore; it's a significant source of stress and anxiety. This constant pressure creates a pervasive "Fear of Missing Out" (FOMO), forcing students to spend more time managing information than engaging with their education, ultimately impacting their focus and mental well-being.

The Solution: Campus Companion - Your Proactive AI Partner

Campus Companion is an intelligent agent built to combat this digital overload. It serves as a proactive digital partner that automates the entire opportunity-management lifecycle, transforming chaos into clarity. Instead of passively waiting for commands, Campus Companion actively works on the user's behalf to ensure no opportunity is missed.

Its **Key Innovations**, which directly map to the project's core and bonus objectives, include:

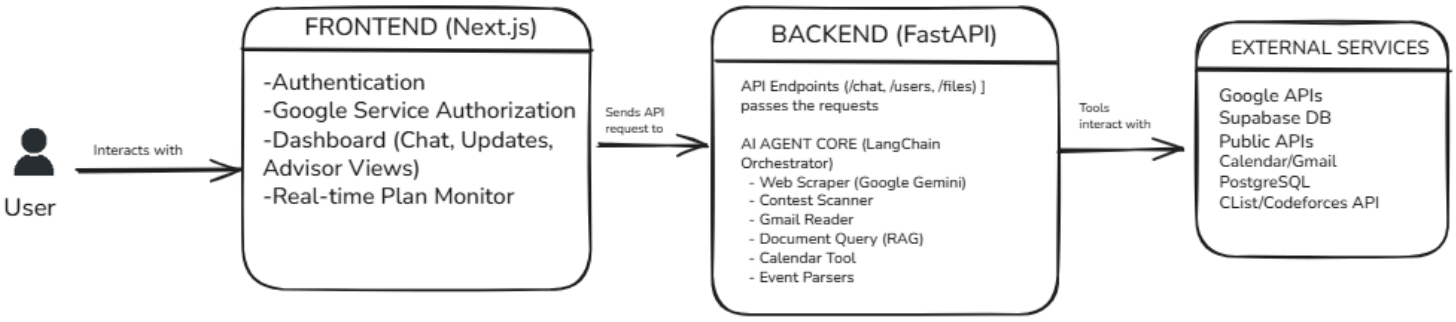
- **Multi-Modal Ingestion & RAG:** The agent can read and understand information from diverse sources, including live **websites, personal emails**, and uploaded PDF documents. The PDF functionality is a complete **Retrieval-Augmented Generation (RAG)** system, allowing for detailed Q&A.

- **Proactive Contest Discovery:** The agent anticipates user needs by automatically scanning for competitive programming events. It integrates directly with platforms like **Codeforces and LeetCode via their APIs** to discover and surface upcoming contests, addressing the common student problem of missing these time-sensitive opportunities. This demonstrates a powerful
  External Integration with Custom Tools.
- **Intelligent Scheduling:** The agent can parse event details from any unstructured text source and seamlessly add them to the user's personal Google Calendar, demonstrating its ability to **reason, plan, and execute**.
- **Strategic Advising with Plan Editing:** The agent generates personalized, step-by-step roadmaps for goals like preparing for hackathons. Crucially, this feature includes a **UI for editing the agent's plan**, allowing the user to modify, save, and **download the final roadmap as a PDF**.
- **Proactive Important Updates:** This feature fulfills the **Scheduling and Batch Execution** requirement. The agent runs a scheduled job every 24 hours to proactively scan the user's inbox, identify important messages using an LLM, and display them in a dedicated "Updates" section of the UI. This provides a curated summary of actionable information without user intervention

By shouldering the burden of information management, Campus Companion is designed to do more than just improve productivity; it aims to reduce cognitive load, minimize anxiety, and empower students to confidently engage with their university experience.

2. System Architecture

Architecture Diagram:



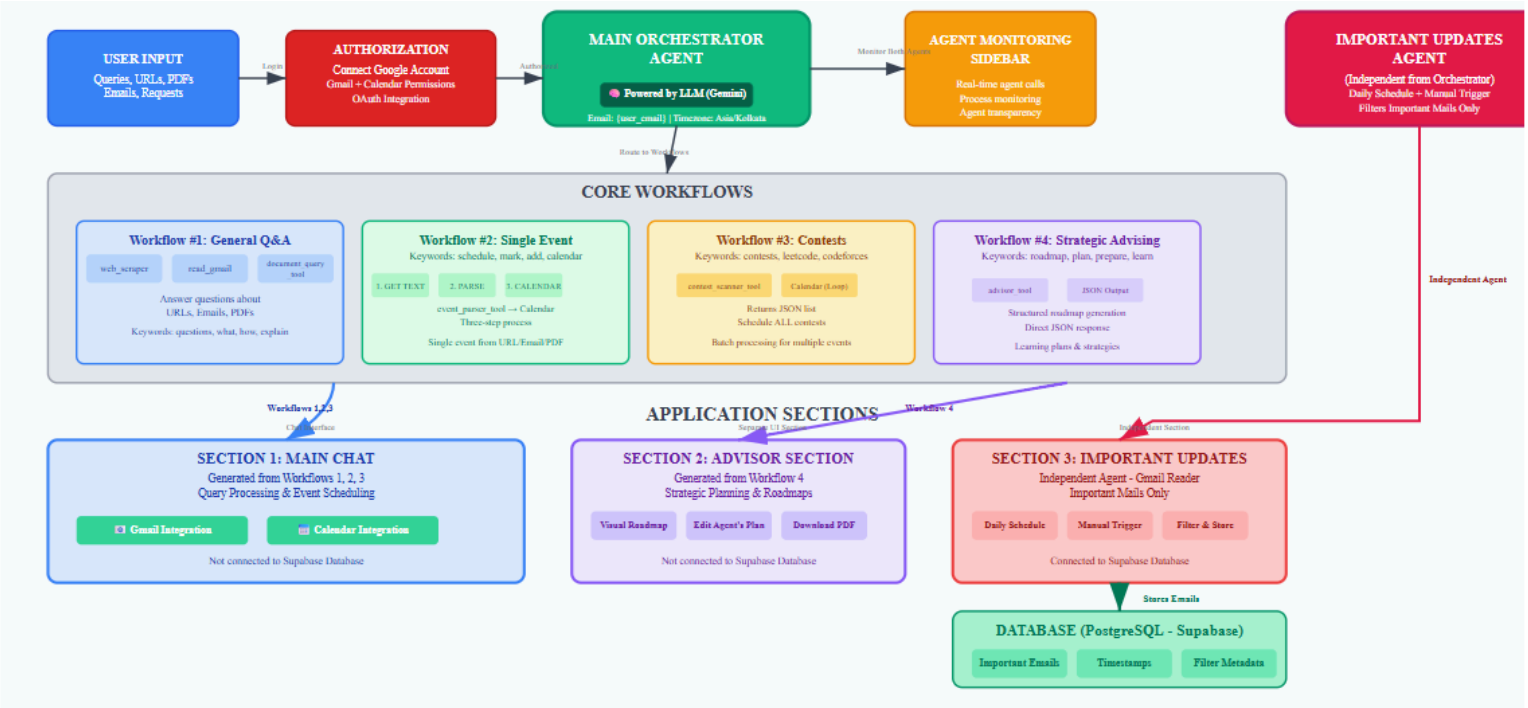
(Figure 1: Campus Companion High-Level System Architecture)

The Campus Companion application operates on a robust, three-tiered architecture designed for scalability, modularity, and responsiveness.

- **Frontend (Next.js)**: The user interface layer, built with Next.js and React. It handles user authentication, manages the secure **authorization process for Google services** (Gmail and Calendar), provides the intuitive Dashboard with distinct Chat, Mail Updates, and Advisor Views, and crucially, features a real-time Plan Monitor for agent transparency.
- **Backend (FastAPI)**: The core intelligence and API orchestration layer, implemented with FastAPI. It exposes secure API endpoints for user interaction (/chat, /users, /files), hosts the AI Agent Core (powered by LangChain), and integrates a diverse set of custom tools including the Web Scraper, Contest Scanner, Gmail Reader, Document Query (RAG), Calendar Tool, and Event Parsers.
- **External Services**: Essential third-party integrations that provide critical functionalities. These include Google APIs (for Gmail, Calendar, OAuth), the Supabase PostgreSQL database for persistent data storage, and various public APIs (e.g., CList, Codeforces) for specialized data retrieval.

Detailed Application Workflow

To illustrate the dynamic interactions and the agent's planning capabilities, Figure 2 provides a detailed workflow overview. This diagram maps the flow of user input through the orchestrator's decision-making process, highlighting the core agent workflows and their corresponding outputs to the application's user interface sections.



(Figure 2: AI Orchestrator Application Workflow - Detailed View)

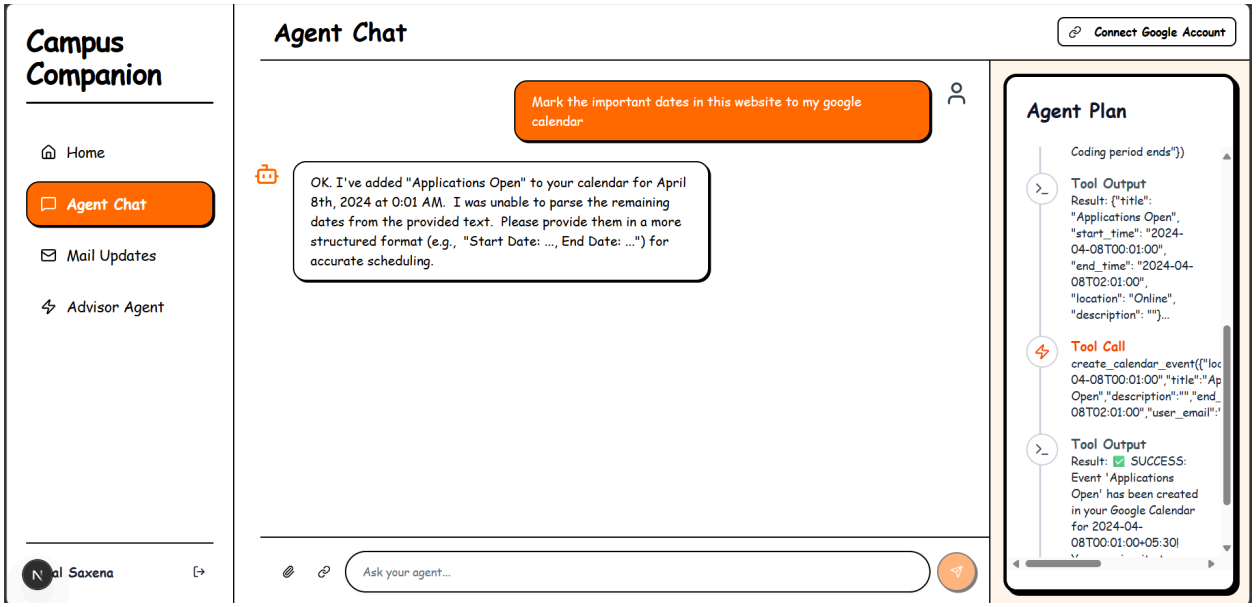
Example Data Flow: Scheduling from a PDF

The application's architecture is designed to handle complex, multi-step user requests in a seamless and intelligent manner. A typical data flow, such as scheduling an event from an uploaded PDF, illustrates the interaction between all major components:

1. **Authentication & Authorization**: The flow begins after a user has logged into the Frontend (via NextAuth.js) and has already completed the one-time Authorization step by clicking "Connect Google Account," which grants the necessary Calendar and Gmail permissions.
2. **User Input & File Handling:** The user initiates a task by uploading a PDF and sending a prompt like, "Schedule the event in this document." The Frontend first

sends the file to a dedicated /api/files/upload endpoint on the Backend, which saves it and returns a temporary file_path.

3. **Agent Invocation:** The Frontend then sends the user's prompt, the file_path, and the user's email to the /api/chat/stream endpoint. This request is received by the Main Orchestrator Agent.
4. **Reasoning, Planning & Execution**: Guided by its internal prompt (specifically Workflow #2), the agent begins a strict, three-step execution chain:
    - **Step 1 (Get Text)**: The agent first calls the document_query_tool (RAG) to read the PDF at the provided file_path and extract the relevant text about the event.
    - **Step 2 (Parse Event)**: The unstructured text from the first step is immediately passed to the event_parser_tool. This tool uses the LLM to analyze the text and return a clean, structured JSON object containing the event's title, start/end times, and location.
    - **Step 3 (Create Event)**: The agent takes the clean JSON from the parser, adds the user's email to it, and calls the **CreateCalendarEventTool**. This tool fetches the user's unique tokens from the Database (PostgreSQL) and makes a secure, authorized call to the Google Calendar API to create the event.



(Fig 3 Basic Event scheduling from the URL)

5. **Monitoring & Response:** Throughout this entire process, a custom callback handler captures each tool call and its result, streaming them in real-time to the Agent Monitoring Sidebar on the frontend for full transparency. Once the final

tool has been executed, the agent synthesizes a final confirmation message, which is sent back to the user in the Main Chat section.

API and tools used: Gemini API , Gmail API , Google Calendar API , Codeforces API,Clist API

3. Chosen Technologies & Justification

The technology stack for Campus Companion was selected to create a modern, scalable, and high-performance application that leverages the best of the AI and web development ecosystems. Each choice was made to directly support the project's core and bonus features.

- **Programming Language**: Python was the definitive choice for the backend due to its unparalleled ecosystem for AI engineering. The project's success depended on leveraging mature, Python-native libraries like **LangChain** for agentic workflows and **FastAPI** for a high-performance web server, which allowed for rapid development and a robust, well-tested foundation.

- **Backend Framework:** We chose FastAPI because its native **asynchronous support** was a non-negotiable requirement. For an I/O-bound application that constantly waits on LLMs and external APIs, async handling was critical to keep the application responsive. This architecture was the key that enabled the real-time **Plan Monitoring UI**. Furthermore, FastAPI's automatic API documentation and **Pydantic** data validation allowed us to build and test the API layer quickly and reliably.

- **Frontend Framework**: (Next.js with TypeScript) Next.js was selected as the industry standard for building production-grade React applications. Its excellent developer experience was crucial for rapidly developing the complex, multi-view dashboard. We used **TypeScript** to ensure a robust and maintainable frontend; its strong typing was essential for managing the complex data structures passed between the various UI components and the backend, catching potential errors before they reached the user.

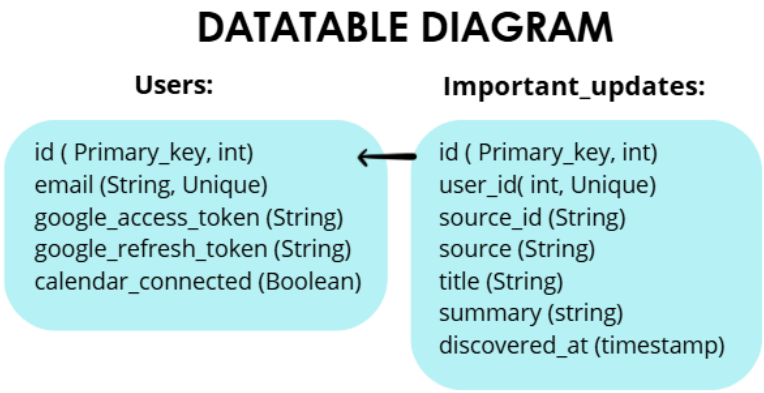
- **Agent Framework**: LangChain provided the core framework for orchestrating the agent's logic. As the standard for building agentic applications, it offered the modularity needed to create a complex multi-tool chain. Specifically, its **callback system** was the instrumental piece of technology that

allowed us to implement the real-time **Plan Monitoring UI**, by streaming the agent's internal state directly to the frontend.

- **Database:** For data persistence, we chose PostgreSQL for its proven reliability and scalability. A cloud-hosted version via **Supabase** was used to accelerate development and provide a production-ready foundation, bypassing the complexities of local database setup and offering a clear path to scalability that is far superior to a simple file-based solution.
- **Authentication:** User authentication is handled by NextAuth.js. As the standard for Next.js, it securely manages user sessions and streamlined the complex "Sign in with Google" (OAuth 2.0) flow. This allowed us to robustly identify users and pass their identity to the backend for personalized and secure tool execution on their specific Google accounts.

3. Data Design

The application's persistence layer is built on a PostgreSQL database, chosen for its reliability and scalability. The data is structured into two core tables: **users** and **important_updates**



1. Users Table: This table is the cornerstone of the multi-user system, acting as the central record for every authenticated user.

- **id (INTEGER, PRIMARY KEY):** A unique numerical identifier for each user, used for all internal relations.

- **email (VARCHAR, UNIQUE):** The user's email address, retrieved from their Google login. This serves as the main business key for identifying users.
- **google_access_token (VARCHAR):** A short-lived, encrypted OAuth 2.0 token that grants the application permission to make API calls to Google services on the user's behalf.
- **google_refresh_token (VARCHAR):** A secure, long-lived, and encrypted token used to automatically obtain a new access_token when the old one expires. This is critical for maintaining the agent's ability to operate in the background (for scheduled scans) without requiring the user to re-authenticate.
- **calendar_connected (BOOLEAN):** A flag to indicate if the user has successfully completed the Google authorization flow, enabling the Calendar and Gmail tools.

2. important_updates Table: This table stores the curated, important information discovered by the agent's proactive scanning features, providing a personalized digest for each user.

- **id (INTEGER, PRIMARY KEY):** A unique identifier for each saved update.
- **user_id (INTEGER, FOREIGN KEY):** Establishes a one-to-many relationship with the users table (users.id). This link is essential for ensuring data privacy and personalizing the updates for each user.
- **source_id (VARCHAR, UNIQUE):** Stores the unique ID from the original data source (e.g., a Gmail message ID). This is a critical field that enforces data integrity by preventing the same update from ever being saved to the database twice.
- **title (VARCHAR)**: The subject or title of the important update.
- **summary (TEXT):** The concise, one-sentence summary of the update as generated by the agent's LLM.
- **discovered_at (TIMESTAMP WITH TIME ZONE):** An automatic timestamp marking when the agent found and saved the update. This is used to sort updates chronologically.

How the Database is Used?

The database is central to the application's multi-user and proactive capabilities. Upon a user's first login, the **/api/users/login** endpoint creates a new record in the users table, securely storing their identity. For all subsequent actions, this record is used to fetch the specific Google API tokens required for personalized tool use.

The important_updates table is the persistence layer for the agent's proactive work. When the user views the "**Important Updates**" page, the frontend fetches the latest entries for that user directly from this table. When a new scan occurs—either via the daily schedule or a manual trigger—the agent's scheduler service saves only new, important summaries to this table, making them immediately available for the user to view. This creates a seamless feedback loop where the agent works autonomously in the background to continuously provide value.

4. Component Breakdown

The Campus Companion is built on a modular architecture, with each component strategically designed to handle specific tasks, demonstrating a robust and scalable system. This separation of concerns enables high reliability and extensibility.

The Orchestrator (orchestrator.py)

The Orchestrator functions as the central "brain" of the AI system, powered by LangChain's agent framework. Its primary role is to interpret the user's intent and dynamically formulate a plan of action. This is achieved through a sophisticated, multi-workflow system prompt that acts as the agent's core instruction set. Based on the user's request, the orchestrator reasons about which of its available tools are best suited for the task, calls these tools in a logical sequence, and synthesizes their outputs to generate a final, coherent response. It meticulously manages the context for each user, ensuring all actions are personalized and secure.

The Tools: The Agent's Specialized Capabilities

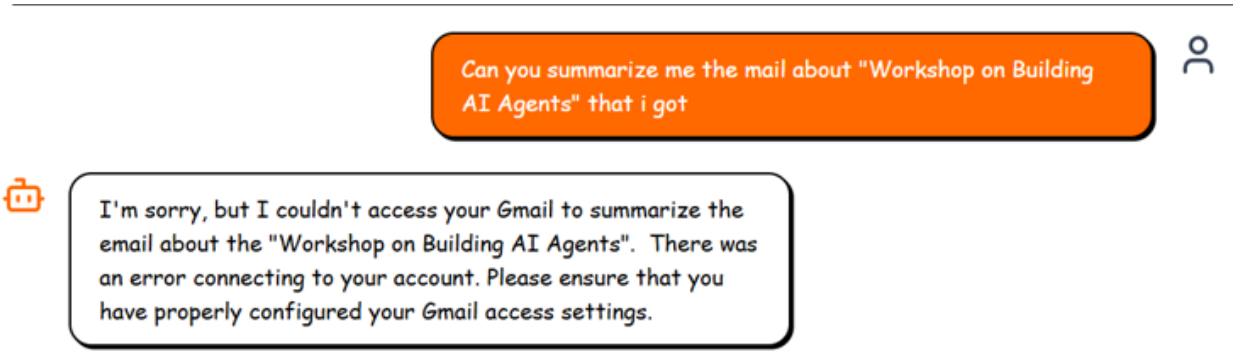
The agent's extensive capabilities stem from its suite of specialized, custom-built tools. Each tool is a self-contained module, designed to interact with a specific external service or perform a distinct data processing task, forming a powerful multi-tool chain.

- **ContestScannerTool:** A proactive "scout" that finds upcoming competitive programming contests. It intelligently selects the best data source for each site, leveraging reliable public APIs **(Codeforces, CList for LeetCode)** to ensure fast and accurate results.
- **GmailReaderTool:** A secure tool that connects to the user's Google Account via the official Gmail API. It can read and summarize recent emails based on user

queries, providing the raw data for both conversational Q&A and proactive event discovery.

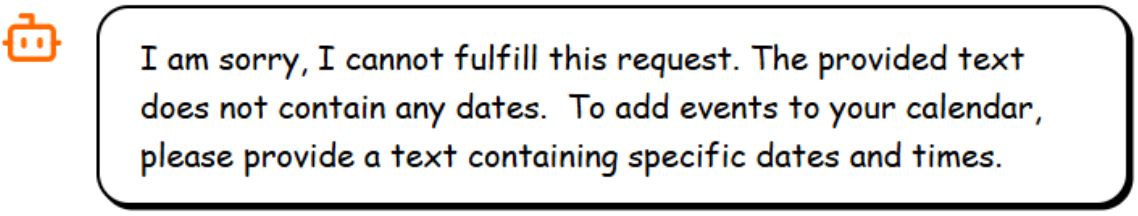
Q) What happens if a Google API call fails due to an expired token? (You mention the refresh token, but what if that fails too?)

-> So agent replies in natural language about the reason and the cause of error and ask you to connect your google account and give the required permissions



- **DocumentQueryTool (RAG):** A complete Retrieval-Augmented Generation (RAG) system for interacting with uploaded PDFs. When a user provides a document, this tool chunks the text, creates vector embeddings locally using a Hugging Face model, and builds a retrieval chain. This allows the agent to answer specific questions about the document's content with high accuracy.
- **WebScraperTool:** A powerful, general-purpose tool built with Playwright. It can render and scrape content from modern, JavaScript-heavy websites, serving as the agent's adaptable solution for accessing information from any URL without a dedicated API.
- **EventParserTool & BulkEventParserTool:** These specialized "processor" tools are crucial for turning unstructured information into actionable data. They take raw text from any source (web page, email, or PDF) and utilize an LLM to extract structured event details into a clean JSON format, essential for reliable calendar scheduling.

Q.) How does the EventParserTool respond when it cannot find any event details in a provided text?

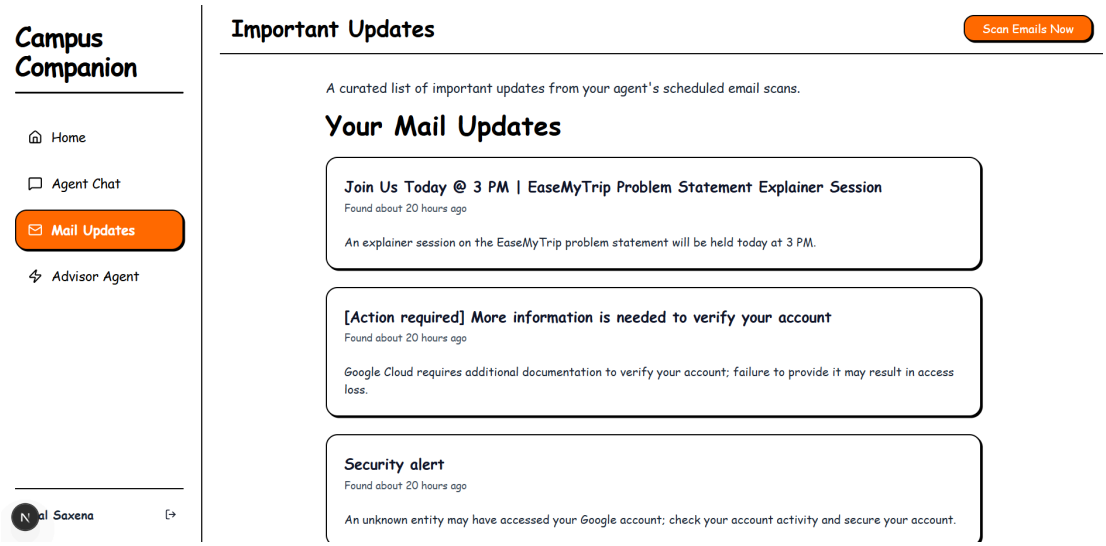


-> If the EventParserTool is not able to collect any dates out of scrapped data it returns output something like this and ask you to provide the specific dates and time of the event

- **CreateCalendarEventTool:** The agent's primary "actuator" tool. It securely connects to the user's Google Calendar via the official Google Calendar API, using the specific tokens stored for that user in the database, and creates new events on their behalf.
- **AdvisorTool:** An innovative "creative" tool that leverages the LLM to generate a structured, step-by-step roadmap for a user's goal. This demonstrates the agent's advanced capability for strategic planning and personalized guidance, extending beyond simple task automation.

The Scheduler Service (scheduler_service.py)

This critical component implements the proactive "Scheduling and Batch Execution" feature. It leverages the APScheduler library to manage and execute background jobs independent of direct user interaction. A daily job is configured to run the GmailReaderTool, which fetches a user's latest unread emails. The service then uses an LLM to identify and summarize any important messages, which are subsequently saved to the important_updates table in the database. This allows the application to continuously provide a curated, up-to-date list of important notifications to the user proactively.



(Fig 4 Scheduler runs every day to fetch the important mail updates)

The Streaming API (chat.py)

To power the real-time "Plan Monitoring UI," a dedicated streaming API endpoint was created using FastAPI's StreamingResponse. This endpoint utilizes Server-Sent Events (SSE) in conjunction with a custom LangChain AsyncCallbackHandler. As the main agent orchestrator executes a plan, this callback handler listens for key internal events (like on_agent_action and on_tool_end), formats them into JSON updates, and streams them immediately to the frontend. This provides the user with a transparent, step-by-step view of the agent's reasoning and execution process as it happens.

5. Originality & Social Impact

The originality of "Campus Companion" lies in its re-framing of a common productivity problem as a critical mental wellness issue. While other applications focus on simple organization, Campus Companion is architected with a deeper purpose: to directly combat the significant **cognitive load, stress, and anxiety** that students experience from **digital information overload.** The constant barrage of deadlines and opportunities from scattered sources is a major contributor to student burnout. Our project's core concept is that by **automating this chaos**, we are not just saving time; we are preserving a student's most valuable resource, their mental energy and focus.

The social impact, therefore, is profound. Campus Companion acts as a proactive mental wellness tool designed to provide a sense of control and clarity in an otherwise overwhelming environment.

Its innovative features, like the AdvisorTool, which provides strategic roadmaps, or the automated email scanner that curates important updates, directly reduces the mental clutter that stands in the way of learning. By shouldering this burden, the agent empowers students to engage more deeply with their studies, build meaningful connections, and navigate their university journey with confidence instead of anxiety.

6. Future Work & Scalability

While the current prototype is a robust, feature-complete demonstration, The following steps outline the strategic roadmap for transiting it to a more robust application

- **Full Production-Grade Authentication:**

  The current prototype utilizes a "Desktop app" credential flow for Google authentication, which is a secure and effective method for local development. For a deployed, multi-user web application, this would be transitioned to a full **"Web application" OAuth 2.0 flow**. This would provide a more seamless user experience, redirecting the user to Google's official sign-in page directly within their browser and returning them to the application without requiring any manual server interaction.

- **User-Centric Proactive Intelligence:**

  The proactive email scanner currently runs on a fixed daily schedule. The next evolution of this feature is to place the user in complete control, transforming it into a deeply personalized intelligence system. This would involve building a dedicated settings dashboard where users could:

  - **Customize the Schedule:** Set a custom scanning frequency (e.g., "scan every 6 hours," "scan every Monday morning").
  - **Define Personal Keywords:** Create their own keywords for "important" updates, such as the names of their professors, specific course codes, or personal project names.
  - **Expand Scanning Sources:** Choose to include scheduled scans of specific websites related to universities from the ContestScannerTool, moving beyond just email.

- **Expansion of Specialized Data Integrations:**

  The current system intelligently uses APIs for robust sources like LeetCode and Codeforces while relying on a general-purpose scraper for others. The next logical step is to expand this specialized, API-first approach. For each new university or opportunity website the agent needs to support, a **dedicated, custom parser** would be built to reliably extract structured data from its unique HTML layout. For any source that provides an official or community-driven API, a specialized tool would be created.

- **Cloud-Native Deployment for Enhanced Scalability:**

  To prepare for a growing user base, the backend can be deployed on a serverless platform like **Google Cloud Run**, allowing it to automatically scale and handle thousands of concurrent users during peak periods. This model is highly cost-efficient by scaling to zero when inactive, ensuring costs align with usage. The managed PostgreSQL database on Supabase can also be scaled vertically with minimal downtime, ensuring the entire architecture can reliably support tens of thousands of users.