AI Scheduling Assistant Demo – Design Document

Overview

- **Goal:** Demonstrate an AI-driven scheduling assistant that can parse incoming emails requesting product shipments and automatically plan a fulfillment schedule and reply with a confirmation.
- **Context:** This project (Scenario-3 by Salami Hamza) is a proof-of-concept demo, not a production system. It showcases how GPT-4, automation workflows, and a custom backend can handle supply chain scheduling requests.
- **Scope:** The demo focuses on a single use case: a Distribution Center (DC) sends an email requesting an urgent shipment of products by a certain date. The system parses the request, "simulates" scheduling the order in a production plan, and drafts a response email. All components run on a small AWS EC2 instance for demonstration purposes.

The AI Scheduling Assistant **automates email handling for urgent requests**. A GPT-4 agent extracts structured data from the email (like product SKU, quantity, needed date), a backend service computes a fulfillment plan, and an automated reply email is generated. The **emphasis is on integration** of AI (for language understanding) with workflow automation and a database-driven scheduling engine. *This document provides a detailed design of the system's architecture, workflow, and setup.* Each section includes a brief summary (TL;DR) for clarity.

Functional Architecture

- **TL;DR:** The system is composed of a Streamlit web UI for monitoring, a FastAPI backend for scheduling logic, an n8n workflow acting as the AI "agent" and process orchestrator, and a Postgres database for storing requests and plans. These components communicate on an AWS EC2 instance (via HTTP and the database). The n8n workflow triggers on new emails, uses GPT-4 to parse content, calls the FastAPI service for simulation logic, and then sends an automated email reply.

Components & Roles:

- **Streamlit UI:** A front-end dashboard (`app.py`) that allows users to **monitor requests and outcomes**. It can display incoming DC requests, the system's planned scenarios, and reply status. The UI uses Plotly for any charts (e.g. showing volumes or fulfillment timelines) and connects to the Postgres database (via `psycopg2` and pandas) to fetch data. In this demo, the Streamlit app is primarily read-only, providing insight into the AI assistant's decisions and allowing basic validation. *(For example, it might list all DC requests with their status – NEW, FULLY PLANNED, PARTIALLY PLANNED – and show a summary chart of how requests are being fulfilled.)*

- **FastAPI Backend:** A Python FastAPI application (`main.py`) that encapsulates the **business logic for scheduling**. It exposes endpoints (notably `/simulate_request`) that the workflow calls to simulate a plan for the requested shipment. The FastAPI service interacts with the Postgres DB: it reads relevant data (product info, inventory levels, production capacity) and writes results (proposed schedule scenarios and a draft email response). Essentially, FastAPI is the "brain" that, given a parsed request, decides **if and how the order can be fulfilled by the requested date**. It then prepares a human-friendly response message.

- **n8n Workflow (AI Agent):** n8n (a workflow automation tool) orchestrates the end-to-end **email processing pipeline**. The Scenario-3 workflow is configured to trigger on a new email arrival (via a Gmail node watching a designated inbox). It performs the following:

- **GPT-4 Parsing:** It uses an OpenAI GPT-4 node with a prompt to extract structured fields from the email (e.g., `dc_name`, `sku_id`, `requested_qty`, `requested_ship_date`, etc.).
- **DB Logging:** It inserts a new record into the `dc_requests` table (via a Postgres node) to log the incoming request details. Initially, this record might be marked as "NEW" status.
- **API Call:** It makes an HTTP POST call to the FastAPI `/simulate_request` endpoint, sending the structured request JSON.
- **Receive Plan:** It waits for the FastAPI response, which contains a draft reply subject and body (often along with scenario metadata).

- **Send Email:** Finally, it uses a Gmail node to automatically send an email reply. The reply's subject and HTML body come from the FastAPI response (e.g. "RE: [Original Subject]" and a message explaining the shipment plan). After sending, the workflow can update the request's status in the DB (e.g. marking it as replied/sent).

- **PostgreSQL Database:** A central **data store** for all persistent information. The database holds:

- **Product Catalog** – list of products/SKUs and details (used by GPT-4 prompt to map described items to SKU IDs).
- **Inventory Levels** – current stock for each product or raw materials (to know what is immediately available).
- **Production Lines & Capacities** – information on manufacturing lines and how much they can produce (to simulate production if needed).
- **Schedule** – a table for production schedule or shipments (the simulation may create entries here to fulfill the request by certain dates).
- **DC Requests** – each incoming request email parsed by the AI, including the DC name, product SKU, quantity, requested ship date, original email text, subject, status, and timestamp.
- **Reply Drafts** – records of the email replies the system prepares/sends, including the reply content and whether it was sent. Each reply draft typically links to a DC request.
- **Scenarios** – details of the simulation for a request (e.g. scenario name, planned production or shipping date, quantity to be produced/shipped, whether the request can be fully met, etc.). In this demo, usually one scenario ("Scenario 1") is auto-generated per request.

All components are loosely coupled: **n8n acts as the glue**, using HTTP and database operations to connect the email source, the AI parsing, the planning logic, and email output. The Streamlit UI is an optional observer that also connects to the same database for monitoring. This modular design means each part can be developed and tested in isolation (for instance, the FastAPI logic can be tested with sample inputs without involving the email workflow).

Workflow: End-to-End Logic

- **TL;DR:** When a new email arrives from a Distribution Center, the n8n workflow kicks off. The email text is fed to GPT-4 which **extracts structured data** (DC name, SKU, quantity, date). The structured request is saved to the DB and sent to the FastAPI service for simulation. FastAPI **simulates a fulfillment plan** (possibly scheduling production or using inventory) and returns a draft response (with subject and body text). n8n then sends an automated reply email to the DC, and the cycle is complete. The entire process is logged in the database, and can be reviewed via the Streamlit UI or logs.

Step-by-Step Workflow:

1. **Email Trigger:** A Distribution Center sends an email to the designated inbox (e.g., *plant_requests@example.com*). For the demo, a Gmail account is used. Example email content might be: *Subject: "URGENT Promo Support - Need 12k cases Citrus Shampoo"*; Body: "Hi Plant Team, We have an urgent promotion... need 12,000 cases of Citrus Shampoo (200ml) by Nov 2, 2025... Can you confirm availability?"

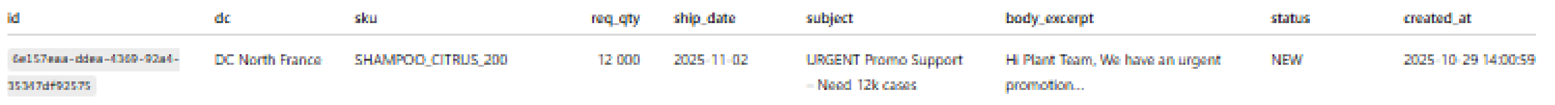
2. **n8n Triggers the Flow:** The n8n workflow is set up with an **email trigger node** (using Gmail's API). It periodically checks the inbox for new messages matching certain criteria (or uses a push if configured). When the new request email is detected, the workflow starts. The email's subject, body, sender, etc., become available as JSON fields within n8n.

3. **GPT-4 Parsing (AI Agent):** The first major action is a **GPT-4 node** in n8n. It sends the email subject and body to OpenAI GPT-4 with a crafted prompt. The prompt instructs GPT-4 to extract specific fields and return a JSON object. For example, the prompt defines keys like `dc_name`, `sku_id`, `requested_qty`, `requested_ship_date`, etc., and provides a product catalog for reference. GPT-4 then returns something like:

   ```
   {
     "dc_name": "DC North France",
     "sku_id": "SHAMPOO_CITRUS_200",
     "requested_qty": 12000,
     "requested_ship_date": "2025-11-02",
     "email_subject": "URGENT Promo Support - Need 12k cases Citrus Shampoo",
     "email_body": "Hi Plant Team, ... (full email text) ... DC North France"
   }
   ```

   This step **transforms unstructured text into structured data** that the rest of the pipeline can work with. *(Note: If any field is missing or ambiguous, the GPT-4 prompt is designed to output* `null` *for that field. For instance, if no date is mentioned in the email,* `requested_ship_date` *would be null.)*

4. **Log Request to Database:** The workflow next uses a **Postgres node** to insert a new record into the `dc_requests` table. This log includes the parsed fields and perhaps an initial status (e.g. "NEW"). For example, a row in `dc_requests` might be:



id	dc	sku	req_qty	ship_date	subject	body_excerpt	status	created_at
6e157eaa-ddea-4369-92a4-35347df92575	DC North France	SHAMPOO_CITRUS_200	12 000	2025-11-02	URGENT Promo Support – Need 12k cases	Hi Plant Team, We have an urgent promotion...	NEW	2025-10-29 14:00:59

This ensures a permanent record of the request is kept even if the rest of the workflow fails. It also allows the Streamlit UI or analysts to see incoming requests. (In a production scenario, this DB record could trigger other processes or at least serve as an audit log.)

1. **Call Simulation API:** Next, an **HTTP Request node** in n8n sends the structured data to the FastAPI backend. This is a POST request to the endpoint `/simulate_request` (for example, `http://<EC2-IP>:8000/simulate_request`). The JSON payload contains the key fields extracted by GPT-4:

```
{
  "dc_name": "DC North France",
  "sku": "SHAMPOO_CITRUS_200",
  "qty_requested": 12000,
  "requested_date": "2025-11-02"
}
```

The Content-Type is set to `application/json`. This call hands off the task to the **scheduling logic** implemented in the FastAPI service. (Note: The use of the EC2 public IP suggests that n8n might be running externally or in a way that it needs to reach FastAPI over the internet. If all components are on one server, this could just be `localhost:8000` in a real deployment. In the demo, it was configured with a specific IP and port.)

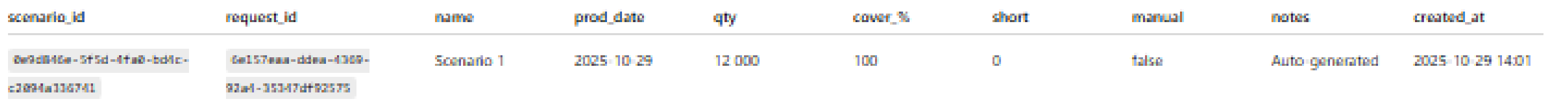
1. **Simulation & Planning (FastAPI):** Upon receiving the request, the FastAPI endpoint handler (in `main.py`) performs the **core logic** of the scheduling assistant:

2. It likely loads relevant data from the database: for example, check the **current inventory** for the requested SKU (from `inventory_materials` table) and the production capacity on manufacturing lines (from `lines` and `line_capability` tables) for the timeframe.

3. It then determines if the requested quantity can be **fully met by the requested date**. If there is sufficient inventory on hand and/or enough production time before the deadline, it plans to fulfill 100% of the request by or before the requested ship date. If not, it might plan a **partial fulfillment** (or a delayed fulfillment) – e.g., it can only supply a portion by that date, with the rest coming later.
4. The logic might create one or multiple “scenarios”. In this simplified demo, it automatically creates **Scenario 1** as the recommended plan. For example, *Scenario 1* might be: *"Produce 12,000 units of Citrus Shampoo by Oct 29, 2025, utilizing available capacity, to ship by Nov 2, 2025."* If inventory was

available, perhaps *"Use 100 cases from stock and produce 11,900 more."* If the request is too large to complete on time, the scenario might note a shortfall.

5. The FastAPI app inserts a record into `dc_request_scenarios` table, capturing the details. For instance:

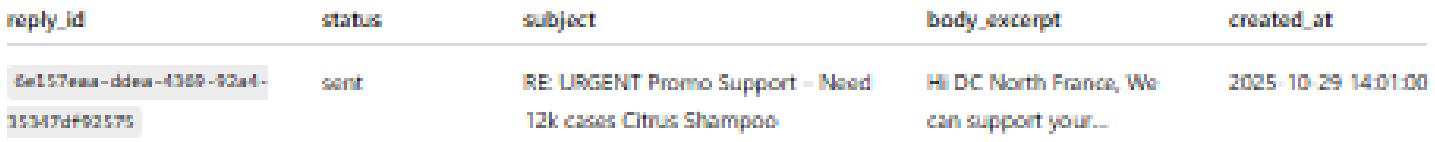


scenario_id	request_id	name	prod_date	qty	cover_%	short	manual	notes	created_at
0e9d846e-5f5d-4fa0-bd4c-c2894a336741	6e157eaa-ddea-4369-92a4-35347df92575	Scenario 1	2025-10-29	12 000	100	0	false	Auto-generated	2025-10-29 14:01

In this example, `percent_covered = 100.00` and `shortfall = 0.00` indicate the full quantity can be met (no shortfall). For another case with a large request, it might still plan production but some shortfall could be noted (and perhaps `percent_covered` would be less). The `manual_override` flag (all scenarios here false) might be a placeholder for whether a human adjusted the plan.

6. Simultaneously, the FastAPI prepares a **reply message draft**. It uses the scenario outcome to compose an email response. For example, if the request can be fully met: *“Hi DC North France, \n\nWe can support your request for 12,000 cases of Citrus Shampoo by 2025-11-02 in full. We have scheduled production to ensure the order will ship on time. \n\nThank you, \nPlant Scheduling Team”*. If only partial, it would word the email differently (e.g., acknowledging how much can be shipped by the date and what the plan is for the remainder).

7. A record is inserted into `dc_request_reply_drafts` to store this draft. For example:



reply_id	status	subject	body_excerpt	created_at
6e157eaa-ddea-4369-92a4-35347df92575	sent	RE: URGENT Promo Support – Need 12k cases Citrus Shampoo	Hi DC North France, We can support your...	2025-10-29 14:01:00

Here, the reply_id is the same as the request id (for simplicity linking them 1-to-1). The status ‘sent’ indicates this draft was sent out. In other cases, the status might be ‘draft’ (if the system chose not to auto-send, e.g., requiring human approval) or labels like 'full' / 'partial' indicating the nature of fulfillment. In this demo, **full vs partial fulfillment** were tracked in the status field for informational purposes (e.g., some replies have status “partial” meaning the response is that only part of the request can be fulfilled by the date).

8. The FastAPI returns a JSON response to n8n containing at least the **email subject and body** for the reply. For instance:

```
{
  "email_subject": "RE: URGENT Promo Support - Need 12k cases Citrus
Shampoo",
  "email_body_html": "Hi DC North France,<br><br>We can support your
request for 12,000 cases of Citrus Shampoo in full. We have scheduled
production to ensure it ships by 2025-11-02.<br><br>Regards,<br>Plant
Scheduling Team"
}
```

(It may also return other info like whether it was full/partial, which could be ignored by n8n but logged in the DB as above.)

9. **Automated Email Reply:** Back in the n8n workflow, after the HTTP request node, the **response from FastAPI is parsed**. The workflow then passes these fields to a **Gmail Send Email node**. This node is configured with the recipient (for the demo, it might send back to the original sender or to a fixed address like the demo owner's email for testing). The subject is set to the `email_subject` from the JSON, and the body to `email_body_html`. The Gmail node sends out the email via the Gmail API. In effect, the DC (or the tester) receives an email that looks like it was written by a human scheduler, confirming the plan.

Example outcome: The DC North France would get an email reply within minutes of their request, with subject "RE: URGENT Promo Support - Need 12k cases Citrus Shampoo" and a body detailing that the 12k cases will be provided by the requested date. If the system could only partially fulfill, the email might say, "we can only supply X cases by that date and Y cases will be delivered by [later date]". All of this text is generated automatically by the backend.

1. **Post-Execution Logging and UI:** At this point, the request has been logged, a scenario planned, and the reply sent. The records in the database reflect the outcome. The Streamlit UI can now display this new request and its status. For instance, the UI might show a table of recent requests, indicating that the request from DC North France for Citrus Shampoo was **fulfilled in full** and an auto-reply was sent. It could also show the production schedule (from the `schedule` table if used – e.g., showing that on 2025-10-29, a production batch of 12,000 units was added for Citrus Shampoo on a particular line).

The **workflow diagram** below summarizes the interactions among components:

(Figure 1: System Workflow — an email from DC triggers the n8n workflow, which uses GPT-4 for parsing, logs to Postgres, calls the FastAPI for planning, and then sends a reply email. The Streamlit UI concurrently reads from the database to display the results.)

<!-- (Diagram would be included in the PDF) -->

1. **Failure Handling (Demo Considerations):** As a demo, some typical robust error handling might be simplified. For example, if GPT-4 returns invalid JSON or if the FastAPI is down, the workflow could fail. In a production system, you would add retries or fallback logic (like alerting a human). In this

demo, the focus is on the **happy path**. All main steps are however logged (either in the DB or via n8n's execution logs), which aids in debugging if something goes wrong during a test.

Technical Architecture

- **TL;DR:** The system's technical architecture consists of a Postgres relational database (with tables for products, inventory, requests, scenarios, etc.), a FastAPI app providing API endpoints to simulate and plan requests (using data from the DB), a Streamlit app for UI (connecting directly to the DB for data), and the n8n workflow which ties everything together. The components communicate via HTTP calls and direct DB access. This section describes the database schema highlights, key API endpoints in FastAPI, the purpose of the Streamlit `app.py`, and how the apps are configured to interact on AWS EC2 (including environment variables and triggers).

Database Schema & Data: The Postgres database (accessible with credentials stored in environment variables on the server) has the following key tables relevant to this project (simplified overview):

- `products` **/** `materials`**:** Tables defining product information. For instance, a `products` table might list SKUs like `SHAMPOO_CITRUS_200` (Citrus Shampoo 200ml) and relate to `materials` or `bill_of_materials` if manufacturing uses raw materials. (The demo includes a small catalog of 4 SKUs in the GPT prompt and corresponding entries in these tables.)
- `inventory_materials`**:** Current inventory levels for products or raw materials. e.g., how many units of each SKU or ingredient are on hand. This is used to decide if an order can be fulfilled from stock.
- `lines` **&** `line_capability`**:** Definitions of production lines (e.g., Line1, Line2 in the factory) and their capabilities (which product categories they can produce, and how fast). For example, a line capability might say Line1 can produce up to 10,000 units of Shampoo per day. The simulation logic likely uses this to figure out how much can be produced by a date.
- `schedule`**:** A production schedule table. In a more elaborate implementation, when planning a request, the system could insert planned production runs into `schedule` (with a foreign key to a production line and date). This demo might not fill out a detailed schedule for every scenario due to simplicity, but the structure is there to support it.
- `users`**:** (If present) likely for Streamlit or future use – for example, a user table could be to manage login to the UI or to assign requests to planners. In this scenario, it's not actively used beyond possibly containing a default user.
- `dc_requests`**:** (Detailed earlier) stores each incoming request parsed from email. Key columns: `id` (UUID), `dc_name`, `sku_id`, `requested_qty` (Numeric), `requested_ship_date` (Date), `email_subject`, `email_body` (text of the email), `status` (e.g. NEW, or later updated to something like DONE), and timestamp fields. This is the main record representing what the DC asked for.
- `dc_request_scenarios`**:** Stores simulation outcomes for requests. Each scenario has an `id` (UUID), references a request (likely via `request_id` or using the same ID), a name (e.g. "Scenario 1"), a planned production or shipment date, quantity planned, percentage of the request covered, any shortfall, and perhaps flags or notes (e.g. auto-generated or if manual intervention is needed). In the demo, typically one scenario per request is created automatically. If the system were expanded, multiple scenarios (like alternatives) could be stored and a user could pick one.
- `dc_request_reply_drafts`**:** Stores the contents of reply emails generated. Key fields: `id` (UUID, often matching the request), `status` (draft/sent/full/partial as discussed),

`email_subject` (the subject line of the reply), `email_body` (the HTML or text body of the reply message), and timestamp. This allows tracking exactly what was communicated back to the DC for each request. In an enterprise setting, these drafts could be presented for approval before sending; in the demo, many are auto-sent, but the table still logs them.

Data Flow & Relationships: Typically, `dc_requests` is the primary entity. It may have a one-to-one or one-to-many relation with `dc_request_scenarios` (one request could have multiple scenario options, though in this scenario it's 1:1) and similarly one request to one reply draft. The `request_id` is used to tie them together. For instance, the FastAPI might use the request's ID as the reply draft ID and as a foreign key in scenario, simplifying lookups. This way, given a request ID, one can find the scenario and the reply that were generated.

FastAPI Endpoints (main.py): The FastAPI app provides an interface for external components (like n8n or potentially the Streamlit app) to invoke the planning logic. Key endpoints include:

- `POST /simulate_request`: **(Core Endpoint)** Accepts a JSON payload with the extracted request data (`dc_name`, `sku`, `qty_requested`, `requested_date`). Triggers the simulation and response generation as described. Returns a JSON with the reply content (and possibly some status info). This endpoint is invoked by n8n for each new request. Internally, it uses the database (via SQLAlchemy or raw SQL with `psycopg2`) to read/write records. For example, it may:
- Find the product info for the given SKU (to get details like pack size or category if needed),
- Check current inventory for that SKU,
- Determine production requirements (perhaps using the `line_capability` table to see how many days or lines are needed),
- Insert scenario and reply records,
- Return the email draft.
- `GET /requests` or `GET /requests/{id}`: **(Optional)** The design could include endpoints to retrieve requests or their status. The Streamlit app might have used direct DB queries instead of API, but for a more decoupled system, one could have an endpoint to list all DC requests and their status, or to fetch the scenario for a given request. There isn't evidence of such calls in the provided materials, but we mention it for completeness.
- `POST /confirm_send` or similar: **(Optional)** If human-in-loop was considered, an endpoint could mark a draft as sent or trigger sending. In our case, n8n handles sending via Gmail, so FastAPI doesn't send emails itself.

(In the provided code, `main.py` *likely defines the app and includes something like* `@app.post("/simulate_request")` *with a Pydantic model or dict parameter. It might instantiate a database connection (using credentials from environment variables loaded via* `python-dotenv`*) and then contain logic to perform the above steps. Given the simplicity, it might not use an ORM but directly execute SQL using* `psycopg2` *or pandas to manipulate data.)*

Streamlit App (app.py): The Streamlit UI is a lightweight web app that runs separately from the FastAPI service. Its purpose is to make the demo **interactive and visual** for demonstration purposes. Key aspects of `app.py` likely include:

- Loading environment variables (with `python-dotenv`) to connect to the Postgres database (host, port, user, password, etc., which would be configured in an `.env` file).

- Querying tables like `dc_requests`, `dc_request_scenarios`, and `dc_request_reply_drafts` to get the latest data. This might be done via `pandas.read_sql()` or using `psycopg2` directly, then converting to pandas DataFrames for easy manipulation.
- Displaying the data: possibly as tables or charts. For example, it may show a table of all requests with their status and a summary of whether they were fully or partially fulfilled. Using Plotly, it could visualize something like "Requested vs Planned quantities over time" or capacity utilization. E.g., a bar chart where each request's quantity is color-coded by whether it was fulfilled fully or partially. Or a timeline chart showing requested ship dates and planned production dates.
- Possibly, interactive elements: The UI could allow filtering by status or selecting a specific request to view details (like the email text, and the reply that was sent). Streamlit makes it easy to show text, so they might display the original email and the AI's draft side by side for a selected request.
- **Triggering actions:** In some demos, the UI might allow triggering the workflow manually. For example, a button "Simulate New Request" that calls the FastAPI or mimics an email coming in. However, since GPT-4 parsing is done in n8n (which the Streamlit app doesn't directly control), the UI probably doesn't try to re-run that. More likely, if any triggers exist, they could be to re-run the simulation on an existing request (in case one wants to tweak something). Given no direct mention in requirements, the Streamlit app is likely read-only, serving as a **monitor and visualization tool**.

Because the UI runs on the same EC2, it can directly connect to Postgres. It might not interact with the FastAPI at all for simplicity, instead reading the tables that FastAPI/n8n have populated.

Inter-Component Communication & AWS Setup: All components are deployed on an AWS EC2 instance (or within the same VPC). Communication happens as follows:

- n8n to FastAPI: over HTTP. The workflow used the EC2's public IP and port 8000. In deployment, this required opening port 8000 in the EC2's security group (accessible to the n8n server or the internet). If n8n was running on the same machine, it could use localhost to avoid external exposure; but using the public IP means the workflow could be run from n8n cloud or another host. It's important to secure this in real deployments (e.g., with authentication on the FastAPI, which wasn't described here). For the demo, the endpoint was likely left open or had a simple security measure since it's not production.
- FastAPI to Postgres: over the local network. The DB likely runs on the same EC2 (on default port 5432). Credentials are set via environment variables (`DB_HOST=localhost`, etc.). Security group allows the EC2 itself to connect to the DB port. (If Postgres were on RDS or another host, network rules would need to allow the EC2 or FastAPI to reach it.)
- Streamlit to Postgres: also local connection on EC2. Streamlit uses the same `.env` for DB creds.
- n8n to Postgres: n8n writes to Postgres using its Postgres node. In n8n, a "Postgres account" credential was configured (we saw it referenced). That contains the DB connection details. So n8n also connects to the database (meaning the DB had to accept connections from wherever n8n is running). If n8n is on EC2 as well, it's local; if external, the DB port might be opened to that external service, which is a consideration.
- n8n to Gmail/Email: The Gmail node uses OAuth credentials to access the Gmail API over the internet. So the EC2 needs internet access to contact Gmail and OpenAI endpoints. (Ensure outbound internet is enabled for the instance, which is usually the case by default).

Environment Variables and Config: Several **configuration values** are needed to run this system, which should be kept out of code. These include:

- **Database credentials:** e.g., `DB_HOST`, `DB_NAME`, `DB_USER`, `DB_PASSWORD`, possibly `DB_PORT`. Both FastAPI and Streamlit will load these (via `python-dotenv` reading an `.env` file) to connect to Postgres.
- **OpenAI API Key:** n8n uses the OpenAI GPT-4 node. In n8n, this is typically set up through Credentials in the n8n UI (not via our code). The OpenAI API key would be stored securely in n8n and referenced by the GPT-4 node. It's not directly in our codebase, but for completeness, it's a secret that needs to be configured for the GPT step to work.
- **Gmail Credentials:** Similarly, the Gmail sending node in n8n requires OAuth2 credentials (client ID/secret and a refresh token or service account) to send emails. These are configured in n8n's credentials and not stored in our code repository.
- **FastAPI server config:** If needed, environment variables might specify the FastAPI host/port. In our case, we run it on port 8000. On EC2, running FastAPI with `host=0.0.0.0` ensures it's accessible externally (per security group rules).
- **Streamlit config:** Streamlit might read some env vars for configuration. For example, `STREAMLIT_SERVER_PORT=8501` (default) or using CLI flags to set it. The `.env` could also hold things like `N8N_URL` if the UI wanted to link to n8n, but that's speculative.

This distributed architecture is container-friendly (each component could run in a Docker container). In the demo, it might be all running in separate processes on one VM. The **AWS EC2 deployment** likely involved the following steps (high-level): installing Python and dependencies, setting up Postgres, running `uvicorn` for FastAPI, running `streamlit` for the UI, and running n8n (either via Docker or npm).

Security Note: Because this is a demo, certain shortcuts were taken (hard-coding IPs, lacking auth on the API). For an open-source release, we recommend securing the FastAPI endpoint (e.g., with an API key or at least limiting it to internal calls) and not exposing the database publicly. The `.env.sample` file should be provided for users to fill in their own secrets (not committing actual credentials).

Testing & Validation

- **TL;DR:** To test the system, you can either use the n8n workflow with a sample email or simulate the process via API calls. The recommended approach is to import the provided n8n workflow, configure the Gmail and OpenAI credentials, then send a test email as a Distribution Center request and observe the automated response and database entries. Alternatively, you can run the FastAPI and Streamlit apps, and manually call the `/simulate_request` endpoint with sample data (bypassing actual email parsing) to verify that a scenario is generated and logged. The Streamlit UI can be used to **validate** that the request was recorded and see the scenario and reply details. Always check the console logs of FastAPI and n8n for any errors during testing.

Using n8n Workflow (End-to-End Test):
1. **Import Workflow:** Load the `FlowKa-Lab_Scenario-3.json` workflow into n8n (either your self-hosted n8n or n8n cloud). This workflow has all nodes configured (GPT-4, Postgres, HTTP, Gmail). Update the credentials: - Set up the **OpenAI** credential in n8n with your API key (if not already present). Ensure the GPT-4 node references the correct credential and model (GPT-4). - Set up the **Gmail** credential (OAuth2 credentials and allow n8n to send as your test email). Alternatively, modify the Gmail node to use SMTP or another email service if Gmail isn't available – or even disable sending and just log the output for now. -

Update the **Postgres** credential in n8n to point to your database (hostname (likely localhost if n8n is local), DB name, user, password). This should match the Postgres where your FastAPI and UI also connect. - If your FastAPI isn't at the exact IP:8000 as in the workflow, update the HTTP Request node URL to wherever your FastAPI is accessible (e.g., `http://localhost:8000/simulate_request` if running on the same machine as n8n). 2. **Triggering the Workflow:** If using a Gmail trigger node, you may simulate a new email by actually sending an email to the connected Gmail account. For testing, you might change the trigger to a manual trigger or webhook. For instance, replace the Gmail trigger with an **Inject node** that supplies a sample subject and body (copied from one of the example emails in the DB) to mimic an incoming email. Then execute the workflow manually in n8n. 3. **Observe the Execution:** In n8n's execution preview, you can watch each step. Verify that: - GPT-4 returns a sensible JSON (check the output of the GPT node). - The Postgres node successfully inserts the request (no error; you can connect to the DB and SELECT from `dc_requests` to confirm). - The HTTP node returns a 200 OK with the `email_subject` and `email_body_html`. - The Gmail node sends the email (if configured). If not actually sending, you can copy the content from the HTTP node output as the "draft". 4. **Check Outcomes:** Look at your email inbox to see if a reply arrived (if you tested with a real email). Also, open the Postgres DB (using a client or psql) and verify that: - The `dc_request_reply_drafts` table has a new entry with the reply content. - The `dc_request_scenarios` has an entry for the request. - The `dc_requests` status might be updated (if the workflow or FastAPI set it to something like 'DONE' or left as 'NEW'). Using the Streamlit UI at this point is helpful: launch `streamlit run app.py` and see if the new request is listed. The UI should reflect the scenario (e.g., show that the request was fulfilled fully or partially).

This end-to-end test validates the entire integration from email to email. **Success criteria:** The AI correctly parsed the email (fields make sense), the plan from FastAPI is logical (e.g., does not promise more than capacity), and the reply email content is coherent and accurate (e.g., it addresses the right DC, mentions the right product and quantities).

Testing FastAPI in Isolation:
For validating the scheduling logic without involving GPT-4 or n8n, you can call the FastAPI endpoint directly: - Run the FastAPI app (`uvicorn main:app --reload`). Ensure it's connected to a database loaded with base data (product catalog, inventory, etc. from the provided SQL). - Use a tool like **curl or HTTPie** or **Postman** to POST a JSON to `http://localhost:8000/simulate_request`. For example:

```
curl -X POST -H "Content-Type: application/json" -d '{
    "dc_name": "DC Test",
    "sku": "SHAMPOO_CITRUS_200",
    "qty_requested": 5000,
    "requested_date": "2025-12-01"
}' http://localhost:8000/simulate_request
```

Adjust the data to various scenarios (small qty vs very large qty, etc.). - Check the response. It should be JSON with an email draft. Also check the database: a new request entry, scenario, and draft should be created. This helps ensure the FastAPI logic is working as expected. - You can then manually mark those replies as sent or even call a hypothetical `send_email` if it existed. In our system, sending is outside FastAPI, so you'd manually verify the content.

Validation Criteria for FastAPI: The simulation should respect inventory and capacity. You can test edge cases like requesting far beyond capacity to see if the response indicates inability to fully meet (e.g., perhaps the reply says "we can only partially fulfill"). Also test missing fields: if you omit `requested_date` (simulate GPT returning null), does it handle it gracefully (perhaps scheduling as soon as possible or returning an error)? In a demo, likely the date is expected.

Streamlit UI Testing:
- Launch the UI and ensure it can connect to the database. The initial view should load existing data (the provided SQL includes sample requests and scenarios). Check if the UI elements (tables/charts) render without error. - If interactive, try selecting different requests or date ranges if those controls exist. - After running a new test (via n8n or manual FastAPI call), refresh the Streamlit app (or it may auto-refresh if coded that way). Verify the new data appears. For example, if you just simulated a request, you should see DC "Test" or whatever in the list with appropriate status. This confirms the UI and backend are in sync.

Logging and Monitoring: For testing, open console logs for both FastAPI and Streamlit (and n8n if self-hosted): - **FastAPI logs:** should show the incoming requests to `/simulate_request` and any printouts (e.g., it might log what scenario it decided or any exceptions). By running uvicorn with `--reload` or in debug, you get error stack traces in the console if something fails in logic. - **Streamlit:** runs as a script; any errors (like DB connection issues) will show in that terminal. Streamlit also provides a browser console for errors in rendering components. - **n8n:** if using n8n desktop or cloud, check the execution logs. If self-hosted, n8n logs to console or files as configured. This can help catch issues like inability to connect to DB (bad credentials) or GPT-4 errors (e.g., hitting rate limits or parsing issues).

By following these testing steps, you ensure each part of the system works and the integration as a whole achieves the desired outcome: automatically reading an email, scheduling the request, and responding appropriately.

Performance and Logging Notes

- **TL;DR:** Performance is not a primary concern for this demo given the low volume (a few requests) and reliance on an external GPT-4 API (which has inherent latency ~ a few seconds per request). The system is designed for clarity over efficiency. Logging is done mainly through the database (every request, scenario, and reply is recorded) and through console outputs of each service. There is minimal optimization or asynchronous processing, but this is acceptable for the demonstration scope. In a real deployment, one would consider concurrency, error handling, and scaling, but here the focus is on a clear, working concept.

Performance considerations:
- The **GPT-4 call** is the slowest step (could be ~5-10 seconds or more), and it happens synchronously in the n8n workflow. This means processing each email takes at least a few seconds. For a demo or small scale this is fine. If we expected high volume, we might use a faster model (GPT-3.5 or a local model) or queue requests.
- The FastAPI simulation logic likely runs very quickly (sub-second) for one request, since it's doing simple queries and perhaps a small calculation. The data set (few products and one day's plan) is tiny. As more requests accumulate in the DB, queries filtering by SKU or date are still trivial. No performance bottlenecks are expected in the Python logic itself.
- **Parallelism:** n8n processes one trigger at a time per workflow (unless scaled out). If multiple emails

arrived simultaneously, n8n would queue them or run in parallel depending on settings. Our workflow touches shared resources (the DB). The DB operations are simple inserts/selects, which can handle multiple connections easily at this scale. The FastAPI (if using Uvicorn with default single worker) would handle one request at a time. In a heavy load scenario, adding more Uvicorn workers or async capabilities might be needed, but for the demo, one worker is fine.
- **Scaling**: The architecture could scale by separating components onto different servers or containers: e.g., host n8n in the cloud, FastAPI on its own server behind an API gateway, Postgres as a managed service, etc. Streamlit could be optional for end-users. However, since it's an integrated demo, all on one EC2 is the simplest.

Logging:
- The primary "log" of activity is the **database** itself. Each key action leaves a record: a new row in `dc_requests` means an email was processed; a new row in `dc_request_reply_drafts` means a reply was generated (and possibly sent if status=sent). This provides a persistent history. One can run SQL queries to see all activity (e.g., count of requests processed, how many were partial vs full).
- **n8n Logging:** n8n provides an execution log for each workflow run (including step-by-step data). This is useful during development. For instance, if GPT-4 output is malformed and the HTTP node fails, the n8n log would show it. For production-like monitoring, n8n can send errors to its internal logs or even notify via email/Slack if a workflow errors out (not set up here explicitly).
- **FastAPI Logging:** By default, Uvicorn logs each request (with status code and time). Additional logging could be inside the simulation logic – e.g., logging whether it found enough inventory or how it decided full vs partial. In the design phase, one might add `print()` statements or use Python's logging module to record these decisions. These logs would appear in the console or a log file if redirected. They are helpful for debugging the algorithm's behavior.
- **Streamlit Logging:** Streamlit doesn't have a complex log; it will output to console if there are errors (like a DB connection failure or exception in code). Since the UI is mostly reading data, not much can go wrong except connectivity issues. However, if someone tries to run the UI while FastAPI or DB is down, they'll see errors which guide the troubleshooting.

Error Handling: In this demo, error handling is minimal: - If GPT-4 fails or returns nonsense, the workflow might stop. (We could enhance the GPT prompt to be very strict or add a fallback to catch JSON parse errors). - If the DB insert fails (e.g., DB is down), n8n will log an error; the email won't be processed fully. There is no automatic retry in the current setup, but one could re-run the workflow once the issue is resolved. - If FastAPI endpoint is unreachable, the n8n HTTP node will error and the email reply won't be sent. In testing, this is a common point to check (correct URL, server running). - If Gmail send fails (due to auth or rate-limit), n8n logs it. The draft is still in the DB, so one could manually send it later.

Overall, the system is intended to handle a **handful of requests in a controlled demo environment**. As such, performance is acceptable and logging (mostly via data records) is sufficient for tracing what happened. For demonstration purposes, the clarity of each step was prioritized over heavy-duty engineering concerns.

Deployment Notes (AWS EC2)

- **TL;DR:** The demo is deployed on a single AWS EC2 instance (Linux) running all components. Key deployment steps include installing Python and required packages (see requirements.txt), setting up a PostgreSQL database, running the FastAPI app (e.g., with Uvicorn on port 8000) and the Streamlit app (default port 8501), and running n8n (which could be via Docker or npm) to handle the workflow. Environment variables are configured on the EC2 (through an `.env` file and in n8n's credential settings) for database connections and API keys. Make sure to open the necessary ports in the EC2 security group (e.g., 8000 for API if needed externally, 8501 for the Streamlit web interface, 5678 if exposing n8n's editor, etc.) and to secure credentials. Below are some tips for deployment and environment configuration:

System Setup: - **EC2 Instance:** Use an Ubuntu 22.04 LTS (for example) or similar distribution. Ensure it has internet connectivity for reaching the OpenAI and Gmail APIs. For a smooth experience, use at least a t2.medium or similar if running all services, although t2.small could suffice since load is low. - **Python & Dependencies:** Install Python 3.10+ on the server. Transfer the project files (or git clone the repository if it's public on GitHub). Create a Python virtual environment and install the packages from `requirements.txt`: - `streamlit`, `fastapi` (although FastAPI wasn't in the requirements file we saw, it should be added along with `uvicorn`), `psycopg2-binary` (for Postgres), `pandas`, `plotly`, `python-dotenv`. - Also, ensure `openai` Python library if it were needed (here n8n calls the API, so the FastAPI doesn't directly use the OpenAI library). - **Note:** If editing the repository, update `requirements.txt` to include `fastapi` and `uvicorn` for completeness, since the backend depends on them. - **PostgreSQL:** Install Postgres on the EC2 (e.g., via `apt-get`). Create a database and a user. For example, create a database named `flowkalab` and a user `flowka_user` with a password. Update the `.env` with these details. Use the provided SQL script (`data/SQL_Data_backup.sql` or a cleaned `demo_data.sql`) to initialize the schema and seed the data. You can copy the SQL file to the server and run `psql -U flowka_user -d flowkalab -f demo_data.sql`. This will create all tables and insert the example products, inventory, and some sample requests and scenarios (so the system has context to work with).

Starting the FastAPI Service: - Before starting, double-check the environment variables for DB connection are loaded. The FastAPI `main.py` likely reads them using `dotenv.load_dotenv()` and then uses `os.getenv("DB_HOST")`, etc. - Launch the app using Uvicorn. For development, you might do:

```
uvicorn main:app --reload --host 0.0.0.0 --port 8000
```

The `--reload` is useful during development to auto-restart on code changes; you might omit it in a stable environment. `--host 0.0.0.0` makes it accessible externally (ensure your security group allows port 8000 inbound if testing from your local machine; for production, you might restrict this or use an SSH tunnel). - You should see Uvicorn startup logs. Test quickly by visiting `http://your-ec2-dns:8000/docs` in a browser – FastAPI's automatic docs. This confirms the API is up. The docs should list the `/simulate_request` endpoint. (Note: If you included `fastapi` in requirements, the docs will show nicely. If not, ensure to install FastAPI; otherwise, this won't run at all.)

Starting the Streamlit App: - In a separate shell (or as a background process/screen session), run:

```
streamlit run app.py --server.port 8501 --server.address 0.0.0.0
```

Streamlit will start and by default open a browser on local machine (which you won't see on a headless EC2). The `0.0.0.0` binding allows you to view the UI via the EC2's public IP and port 8501. Make sure port 8501 is allowed in the security group if you want to access the UI from your PC. Alternatively, you can SSH port-forward if you prefer not to open it publicly. - Once running, open a browser to `http://<EC2-public-IP>:8501`. You should see the dashboard. If it fails to load, check Streamlit logs; likely causes are DB connection issues (check that the .env is in the same directory as app.py and contains correct creds). You can also print some debug info in app.py (like current working directory or env var values) to ensure it's picking up the file.

Running n8n: - **Option 1: n8n Cloud or Local from your machine:** Since n8n just needs to reach the EC2's API and database, you can run n8n on your own machine for testing. Import the workflow, update credentials to point to the EC2's Postgres (`DB_HOST` would be EC2's IP, ensure EC2 allows inbound Postgres or set up an SSH tunnel). Also update the HTTP node to point to EC2's FastAPI. Then trigger the workflow. This avoids installing n8n on the server. - **Option 2: Install n8n on EC2:** You can use Docker: `docker run -p 5678:5678 -v ~/.n8n:/home/node/.n8n n8nio/n8n` (this starts n8n server accessible on port 5678). Or `npm install -g n8n` then `n8n start`. If running on EC2, open port 5678 only to your IP or use SSH tunneling for security, because n8n has an open editor interface. - Once n8n is up, import the workflow JSON as before, set credentials (OpenAI key, Gmail OAuth, Postgres as localhost if n8n is on EC2 and Postgres is local). Update any IP references (if all on EC2 now, the HTTP node URL can be `http://localhost:8000/simulate_request`). Now you have a fully self-contained setup. - Consider running n8n, FastAPI, and Streamlit each in a `screen` or `tmux` session or configure as systemd services for resilience. For a quick demo, manually starting them is fine.

Environment & File Configuration: - Use an `.env` file placed in the project directory (where `main.py` and `app.py` reside) with content like:

```
DB_HOST=localhost
DB_NAME=flowkalab
DB_USER=flowka_user
DB_PASSWORD=yourpassword
```

(plus any other config you might need, e.g., if we externalize FastAPI's host or port, or Streamlit secrets). Both `main.py` and `app.py` load this to get DB settings. This way, you don't hard-code credentials. The repository should include an `.env.sample` with dummy values for guidance. - The n8n credentials are stored within n8n (not in our files). For deployment, just ensure you configure them in the n8n UI or via environment if using n8n's file-based credential loading.

Deployment Verification: - After everything is up, do a full run-through: Send a test email (or simulate one) and watch it go through. Monitor logs. This checks that all pieces that were started are communicating: e.g., n8n on EC2 calling FastAPI on EC2 (should be fast, no firewall issues if using localhost internally), n8n writing to Postgres (should succeed if creds correct), etc. - One common issue on EC2 could be **firewall**: If using any external connections (like if n8n is off-box or you want to access UI externally), ensure the AWS Security Groups are configured appropriately. For instance, open port 8000 to your IP if you want to hit the API externally, open 8501 to your IP for the UI. Keep these restricted or turn them off when not in use, as the demo doesn't implement auth on those interfaces.

Environment Cleanup/Stop: - Since this is a demo, remember to shut down external exposure when not needed (stop the EC2 or at least close the ports) to avoid unwanted access. The Gmail and OpenAI keys especially should be kept safe (n8n does a good job not exposing the key in logs, but don't share the n8n workflow with those credentials embedded).

Attribution

This project is an open-source **early version** of an AI Scheduling Assistant by *Salami Hamza*. It was developed as part of the FlowKa Lab series (Scenario-3) to illustrate how modern AI (GPT-4) can integrate with automation tools and custom logic to streamline business workflows like supply chain scheduling.

Author: Salami Hamza – who designed and implemented the initial concept and demo. The project's code and workflow are made available for educational and development use. We gratefully acknowledge the use of open-source tools including **n8n** (for the workflow automation), **OpenAI GPT-4** (for the language understanding), and various Python libraries (FastAPI, Streamlit, etc.) that power the solution.

License: The project is released under the MIT License, which means it can be freely used, modified, and distributed. (A full copy of the MIT License will be included in the repository.) Please attribute the original author in derivative works.

This documentation and demo aim to accelerate learning and further innovation in AI-driven process automation. Community contributions are welcome – see the README and GitHub repository for how to get involved, report issues, or suggest enhancements.