ChatGPT

Kitchen Dashboard Code Review and OAuth Integration Guide

Overview

This report presents a comprehensive review of the **kitchen_dashboard** project's codebase and Docker configuration, along with guidance for integrating OAuth 2.0 credentials to access Google services (Calendar, Gmail) and a weather API. We examined the Python backend, authentication modules, Dockerfile, and related scripts (excluding the virtual environment) to identify bugs, logic errors, security issues, and general bad practices. Recommendations are provided to fix these issues and improve code quality. Finally, we outline how to use the provided Google OAuth 2.0 credentials file (`auth/google_credentials_joeltimm.json`) to obtain access tokens for Google Calendar, Gmail, and a weather service (e.g. OpenWeather).

Code Review: Issues and Recommendations

After a top-down review of the code across all directories, several issues were identified. Table 1 summarizes the key findings, including bugs, logical errors, and security or best-practice concerns, along with recommendations for each. Subsequent paragraphs provide additional context for these issues.

Key Issues Identified in Code

Issue	Location	Recommendation
Flask app created after blueprint registration – The `app` is used before it's defined, causing a runtime error.	`backend/app.py` (top of file)	Instantiate the Flask app **before** registe For example, do `app = Flask(__nam` `app.register_blueprint(widget_`
Undefined function for Calendar credentials – Code calls `load_calendar_credentials_joeltimm()` which isn't defined in the auth module.	`backend/calendar_widget.py`	Use the existing credential loader (e.g. `load_calendar_credentials()` or `load_calendar_credentials_joel` `credentials.py`). Ensure the functio matches an actual function.
Incorrect module import paths – Several imports reference a `common` package that doesn't exist in the project structure (e.g. `from common.credentials import ...`).	`backend/weather.py`, `backend/onedrive_widget.py`, tests	Fix import paths to use the correct mod replace `common.credentials` with `auth.credentials` (since `auth/cr` the correct location) and `common.auth.onedrive_credentia` proper module or unify into one auth m tests accordingly.

Issue	Location	Recommendation
Weather API call parameter bug – The `/api/weather` endpoint calls `get_weather()` with no city, but the function requires a `city` argument.	`backend/app.py` and `backend/weather.py`	Define a default city or pass a city paran For example, load a city name from an e variable or config file and call `get_wea` Alternatively, modify `get_weather()` none provided.
Weather service logic incomplete – The weather integration code is a placeholder mixing Google Maps API and OpenWeather logic (uses an OAuth token header *and* `appid` query).	`backend/weather.py`	Clarify the approach: if using OpenWea Google OAuth usage and rely solely on using a Google weather API (if available token properly. In practice, use one me (likely OpenWeather with API key, see C below).
Background thread not started in production – The calendar polling thread starts only when running `app.py` directly (inside `if __name__=="__main__":` with `app.run()`), but under Gunicorn this block won't execute. This means scheduled event fetching won't run in the Docker container.	`backend/app.py` (bottom of file)	Ensure background tasks start in all de example, move `start_polling()` ou `__main__` block (perhaps at module in Gunicorn configuration) or use a sched APScheduler) to run periodic tasks rega server.
Missing dependency for OneDrive auth – The code uses the `msal` library in `auth/credentials.py` for OneDrive, but **msal** is not listed in requirements.	`auth/credentials.py`, `requirements.txt`	Add `msal` to `requirements.txt`. K in sync with code usage to avoid `ModuleNotFoundError` in the contai
Duplicate OneDrive auth implementation – There are two different methods for OneDrive OAuth (one using `msal` in `credentials.py`, and one via device code flow in `auth/onedrive_credentials.py`), and the code calls an undefined `get_onedrive_token`.	`backend/onedrive_widget.py`, `auth/onedrive_credentials.py`	Consolidate to one OneDrive auth appr device-code flow in `load_onedrive_c` works but lacks token refresh; the MSAI refresh silently but needs proper usage method and update `onedrive_widge` correct function (e.g. use `auth.credentials.load_onedrive` and have it return a valid token). Remov unused code.
No use of `widget_profiles` blueprint due to registration bug – The widget settings API blueprint exists but wasn't properly registered (because of the app init order bug).	`backend/widget_profiles.py`	After fixing the app/blueprint order, ens widget settings use the `/api/widget` endpoints. No code change needed bey registration fix, but test that this functio reachable.

Issue	Location	Recommendation
Potential file write collisions – Concurrent writes to JSON files (calendar cache, widget profiles, image cache) are not guarded. E.g. the calendar polling thread and an API request might write the cache simultaneously.	`backend/calendar_cache.py`, `backend/widget_profiles.py`, `backend/onedrive_widget.py`	In this small app, risk is low, but for rob locking or using thread-safe writes. Alte API serve from memory cache while the thread updates the file, to avoid simulta
Logging and error handling – Some parts print errors to console (e.g. cache failures) instead of using a logger, and broad exceptions are caught without distinguishing error types.	Multiple files (`calendar_cache.py`, `calendar_widget.py`, etc.)	Use the Python `logging` module cons (as done in some places) instead of `pr` errors go to `logs/error.log` or stdo consider catching specific exceptions fo network errors vs. auth errors) and logg for debugging.
Hard-coded secrets in repository – OAuth client secrets and tokens are present in the repo (`auth/google_credentials_joeltimm.json`, `auth/calendar_token_*.json`). Committing these is a security risk.	`auth/` directory	**Do not commit sensitive credentials c source control.** Instead, store them ou use environment variables/secret mana example, the Google client JSON can be runtime or its contents provided via env tokens should be generated on deploym secure location. Revoke and replace any have been exposed.

Table 1: Summary of identified issues in the codebase and recommended fixes.

Additional Context and Notable Issues:

- **Module Structure and Imports:** The project may have been restructured (from a `common` module to a local `auth` package) without updating all import references. To avoid such issues, ensure the Python packages are properly defined. For example, include an `__init__.py` in the `backend` directory and use relative imports or adjust the PYTHONPATH so that `auth.credentials` can be imported consistently. This will prevent `ModuleNotFoundError` problems when running the app or tests. Similarly, the test file `tests/test_onedrive_widget.py` has import mistakes (`from onedrive_widget.py import ...` is invalid syntax). Fixing the package imports will allow tests to run and mimic real usage.

- **Google Calendar Polling Thread:** The design uses a daemon thread to periodically fetch Google Calendar events (every 5 minutes) and cache them. This is a good approach for keeping data fresh, but remember that when using a WSGI server (Gunicorn), the main script isn't executed directly. As noted, the polling won't start unless explicitly invoked. A solution is to start such threads on Flask app startup. For example, you could initiate `start_polling()` at the bottom of `backend/app.py` unconditionally (with safeguards to not start twice in debug mode), or use Gunicorn's `--preload` or an `on_starting` hook to launch background tasks. Another approach is to use APScheduler or Flask extensions to schedule jobs, which might integrate more cleanly with Flask's lifecycle.

- **Token Management and Refresh:** The `auth/credentials.py` utility functions for Google APIs are designed to automatically refresh tokens if expired (`creds.refresh(Request())`) or run a local OAuth flow if no valid token. This follows Google's recommended pattern for installed apps [1]. The OneDrive token logic, however, is not as robust – the device code implementation writes the token JSON but never refreshes it, and the MSAL-based function simply errors out if no cached token is present. It's recommended to unify token management so that tokens refresh seamlessly. For example, using MSAL's `acquire_token_silent` and `acquire_token_by_device_flow` could allow refresh tokens to be used. At minimum, handle the case where the saved OneDrive token is expired: detect expiry and trigger the device code flow again or prompt the user to re-auth.

- **Use of Configuration and Environment Variables:** Currently, certain values are hard-coded (e.g., the OneDrive folder name `"Space Background photos"`, a placeholder OpenWeather API key in code, etc.). Consider moving such configuration to the `.env` file or a config section. For instance, a `CITY` for weather, `OPENWEATHER_API_KEY`, or the OneDrive folder ID could be environment variables. The project already loads environment variables via `python-dotenv`, so extending this to other settings would improve flexibility. This also avoids editing the code when deploying to a different environment – only the `.env` or actual environment variables need to change.

- **Security Considerations:** In addition to not storing secrets in git, ensure file permissions on credential files in deployment are restricted (non-root user in container, or at least only accessible by the application user). Moreover, if this dashboard is accessible on a network, consider restricting access to the Flask endpoints (currently, there's no authentication or authorization on the REST API). If it's meant for a local kiosk only, that might be acceptable. But if deployed on a home network, you might want to at least put the app behind a firewall or require a simple auth token for the API to prevent unauthorized use.

Dockerfile Review

The provided Dockerfile is straightforward and generally on the right track for a single-container deployment. It sets up a Python 3.12 slim environment, installs dependencies, and runs the Flask app with Gunicorn. Below are observations and best-practice recommendations for the Docker setup:

- **Base Image and Dependencies:** Using `python:3.12-slim` is a reasonable choice for a lightweight image. All required Python packages are installed via `pip install -r requirements.txt`. Since the code uses Google APIs and requests, the chosen packages cover that. However, one missing library is **msal** (as noted) – if OneDrive integration is needed in the container, include it in requirements and reinstall. Also, if any Google APIs require additional system libraries (for example, if using Pillow for images or numpy for certain tasks, which might need JPEG or BLAS libraries), those should be installed via `apt`. According to the project overview, libraries like `libatlas-base-dev`, `libjpeg-dev`, etc., were installed on the Raspberry Pi. In the container, none of those are installed. If the Python libs in use (Flask, google-api-python-client, etc.) work without those, that's fine. Just be aware that if later you add image processing or scientific computing features, you may need to add corresponding `apt-get` steps in the Dockerfile to install system dependencies. Currently, for the listed requirements, no additional system packages are strictly needed.

- **Copying Files and Context:** The Dockerfile uses `COPY . .` which copies the entire project directory into the image. This ensures all code and data files are included, but it can also send unnecessary files to the build context. For example, it appears to have copied the entire `.git` folder and possibly the `venv` folder into the image (if those exist in the build context). This increases image size and could expose sensitive info. It's a best practice to use a `.dockerignore` file to **exclude** files not needed in the container (e.g., `.git/`, `venv/`, local logs, credential files if you plan to inject them differently, etc.) [2]. By adding a `.dockerignore`, you keep the image lean and reduce risk of leaking info. In this project's case, if you intend to bundle the OAuth credentials in the image, that's convenient but consider the security implications. Ideally, you'd exclude actual token files and only include code. Then provide tokens at runtime (see below).

- **Running as Non-Root:** The current Dockerfile doesn't specify a user, so it will run as root by default. For better security, especially if this were ever exposed beyond a trusted LAN, consider adding a step to create a non-root user and switch to it (`RUN adduser appuser && USER appuser`) before running the Gunicorn command. This way, even if an attacker gains access to the container, they don't immediately have root privileges. On a closed Raspberry Pi setup this is less critical, but still a good practice to keep in mind for container hardening.

- **Entrypoint/Command:** The use of Gunicorn with one worker (`--workers=1`) is appropriate given the app's likely low traffic (a single dashboard client). Gunicorn will handle requests efficiently and can be configured to bind to port 5050 as done. The timeout of 120 seconds is a bit high; unless there are very long requests, you might lower that to, say, 30 seconds to free workers if something hangs. Since we rely on a background thread for calendar updates, the request handling itself should be quick. The worker-class `sync` is fine; you could also use `gthread` with a couple of threads if you expect to serve multiple simultaneous long polling requests, but that's not necessary here.

- **Static File Serving:** One thing to note is how static files and the frontend are served. In the project, there is a `frontend/` directory with `templates/index.html` and static JS/CSS, but the Flask app currently doesn't use them (the root endpoint just returns a plain string). In a single-container deployment, you may want Flask to serve the frontend. For example, you could move the `index.html` into a `templates/` directory within the Flask app context (or configure Flask to look at `frontend/templates`). The `Flask(__name__)` default static folder is `static/` and template folder is `templates/` relative to the module. Right now, `frontend/static` and `frontend/templates` might not be automatically recognized. You might serve the index by adding an endpoint like:

```
@app.route("/dashboard")
def dashboard():
    return render_template("index.html")
```

and ensure `template_folder="../frontend/templates"` and `static_folder="../frontend/static"` when creating the Flask app (or move those folders). This way, hitting the app (maybe at `/dashboard` or `/`) will return the actual UI. Alternatively, if the front-end is a separate static site (opened directly in browser), then the back-end API should have CORS enabled. Consider your deployment strategy and update the Docker image accordingly. If serving via Flask, everything is contained in one container; if separate, you might need two containers (one for a static file server or a simple HTTP server serving the front-end files). For simplicity, integrating it into this Flask app container is easiest.

- **Persistent Data:** The container writes cache and log files to the `/app/backend/logs` and `/app/backend/static_data` directories. In a Docker environment, these changes won't persist unless you use volumes. If you want to retain logs or avoid re-fetching OneDrive images on each restart, consider mounting a volume for those directories. This way, updates remain between container restarts. If this dashboard is mostly ephemeral or can refresh itself, you might not need persistence, but logs especially could be valuable for troubleshooting.

- **Secrets Management:** As mentioned, **do not bake real secrets into the image** if it can be helped. A safer practice is to use Docker secrets or environment variables at runtime to supply API keys and credentials [3]. For example, you can build the image without the `auth/*.json` token files, then at runtime mount a volume or use `-e` variables that the container reads to load credentials. The OAuth client secrets JSON could be provided via an env var (and then written to a file inside the container before running, if the libraries need a file). Gmail and Calendar tokens, once obtained, could be mounted from the host (so if you re-create the container you don't need to re-auth). This ensures that if someone pulls the container image from a registry, they won't get your keys. Since this is a personal project, you might not distribute the image, but it's good to design with security in mind.

In summary, the Dockerfile is functional for development, but for a production-like deployment: minimize the build context using `.dockerignore`, don't include dev artifacts, run as non-root, and handle secrets carefully. These changes will make the container more secure and maintainable.

OAuth 2.0 Credentials Usage Guide

The project includes an OAuth 2.0 client credentials file (`google_credentials_joeltimm.json`) intended for accessing Google APIs (Calendar, Gmail, etc.). To utilize this file for obtaining access tokens, you will typically follow Google's OAuth flow for installed applications. The file contains the client ID and client secret that identify your application to Google. Below we describe how to generate and use tokens for **Google Calendar** and **Gmail**, and how to handle a third-party **weather API** (OpenWeather as an example).

Google Calendar API Integration

To access Google Calendar, the application must have an access token for the Google Calendar API scope. The general process is:

1. **Enable the Calendar API in Google Cloud**: Ensure that the Google Cloud project associated with `google_credentials_joeltimm.json` has the Calendar API enabled and that you've configured an OAuth consent screen for external access (if not already done).

2. **Run the OAuth flow to get user credentials**: Using the OAuth client JSON, your app will request authorization from the Google user (your account that has the calendar). In code, you can use `google_auth_oauthlib.flow.InstalledAppFlow` to simplify this. For example:

```
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials

# Define the scope for Google Calendar read/write access
SCOPES = ['https://www.googleapis.com/auth/calendar']  # or auth/
calendar.readonly for read-only

creds = None
token_path = 'auth/calendar_token.json'  # where we'll store the token
# Load existing token if present
if os.path.exists(token_path):
    creds = Credentials.from_authorized_user_file(token_path, SCOPES)
# If no valid credentials, perform OAuth sign-in
if not creds or not creds.valid:
    if creds and creds.expired and creds.refresh_token:
        creds.refresh(Request())
    else:
        flow = InstalledAppFlow.from_client_secrets_file('auth/
google_credentials_joeltimm.json', SCOPES)
        creds = flow.run_local_server(port=0)  # starts a local web server for
the OAuth callback
    # Save the credentials for next time
    with open(token_path, 'w') as token_file:
        token_file.write(creds.to_json())
```

This code will open a browser (or console link) for the Google account to grant the specified scope. Once authorized, it saves the tokens to `calendar_token.json`. The logic also checks if a token already exists and is valid or refreshable – if so, it uses that instead of prompting again. This approach mirrors Google's recommended quickstart pattern [1]. In the provided project, functions like `load_calendar_credentials()` in `auth/credentials.py` implement this logic. You can call that function directly to get a `Credentials` object.

1. **Use the Calendar API**: Once you have `creds` (a `Credentials` object), you can access Google Calendar. The Google API Python client makes this easy:

```
from googleapiclient.discovery import build

service = build('calendar', 'v3', credentials=creds)
# Example: fetch upcoming events
events_result = service.events().list(
    calendarId='primary', timeMin='<ISO8601 timestamp>', maxResults=10,
    singleEvents=True, orderBy='startTime'
).execute()
events = events_result.get('items', [])
for event in events:
    start = event['start'].get('dateTime', event['start'].get('date'))
    summary = event.get('summary', '')
    print(start, summary)
```

In the context of the dashboard, the `backend/calendar_widget.py` already uses this approach: `build('calendar', 'v3', credentials=creds)` and then queries events. You just need to ensure the credentials `creds` passed in are authorized as above. After obtaining events, the code caches them to avoid frequent API calls. If you follow the above steps, your Calendar widget will display current events and refresh the token automatically when it expires (using the refresh token stored in the JSON).

1. **Multiple Calendars (optional)**: If you want to integrate multiple Google Calendar accounts (as the presence of `calendar_token_tsouthworth.json` suggests), you would repeat the auth flow for each account (each user needs to grant access). You can either use separate client IDs for each or use the same client but store different token files. The function `load_calendar_credentials_for(email_suffix)` in `credentials.py` shows an approach: it picks a token file name based on the user's email prefix. If you have separate credential JSONs for each user, you'd call the flow with each respective file. Otherwise, with one client ID, just authenticate each user sequentially and store their tokens.

Gmail API Integration (Sending Email)

Using the same OAuth client file, you can obtain credentials for Gmail API. The Gmail API allows sending emails, reading emails, etc., depending on the scope. The project indicates a plan to send a daily summary email via Gmail, which would require the `'https://www.googleapis.com/auth/gmail.send'` scope (to send mail on your behalf).

To set up Gmail API access:

1. **Enable Gmail API** in the Google Cloud project (if not already enabled alongside Calendar).

2. **Obtain Gmail authorization** similar to Calendar, but with Gmail-specific scope. You can reuse much of the code, just change the scope and token filename:

```
SCOPES = ['https://www.googleapis.com/auth/gmail.send']
token_path = 'auth/gmail_token.json'
# (then same logic as above: load creds if exist, otherwise run flow with
InstalledAppFlow using the same 'auth/google_credentials_joeltimm.json')
```

This will prompt you (the Google account) to grant permission to send email. Save the token to `gmail_token.json`. The provided script `generate_google_tokens.py` actually automates this – it defines a Gmail service with that scope and will generate the token file for you if you run it. You can use that, or integrate directly in code as above.

1. **Using the Gmail API**: With the obtained `creds`, build a Gmail service:

```
service = build('gmail', 'v1', credentials=creds)
```

To send an email, you need to create a MIME message and then use the API to send it. The Gmail API expects a raw message in base64 encoded format. For example:

```
import base64
from email.message import EmailMessage

# Create an email message
message = EmailMessage()
message['Subject'] = "Daily Dashboard Summary"
message['From'] = "your_email@gmail.com"
message['To'] = "recipient@example.com"
message.set_content("Hello, this is the daily summary from Kitchen
Dashboard...")

# Encode message to base64
raw_message = base64.urlsafe_b64encode(message.as_bytes()).decode('utf-8')
body = {'raw': raw_message}
# Send the email
send_result = service.users().messages().send(userId='me', body=body).execute()
print(f"Message sent, ID: {send_result.get('id')}")
```

This would send an email from your Gmail account. In a cron job or scheduled context, you could automate this daily. The token with Gmail.send scope allows the app to send email without needing your password or manual intervention after the initial consent. Just ensure the token is kept safe (as it allows sending email as you).

Note: If using Gmail API feels too complex for sending a simple email, an alternative is to use SMTP (via `smtplib` or a library) with an app password. But since the OAuth setup is already there, using the API is fine and more secure (no storing email credentials). The code above is a sample; in practice, you would probably compile some content (e.g., “X new events today, weather is Y, etc.”) and include that in the email body.

Weather API Integration

For weather information, the project’s plan is a bit unclear. The code snippet in `backend/weather.py` shows an attempt to use a Google API endpoint for weather, protected by Google OAuth (hence the use of a Cloud Platform scope). However, Google’s Maps or Weather API would typically use API keys, not OAuth tokens, or require a different service enabling. Given the instruction, we’ll assume integration with an external weather service like **OpenWeatherMap**, which uses a straightforward API key (no user OAuth).

Using OpenWeather API: OpenWeatherMap provides current weather data via simple HTTP calls. You need to sign up for a free API key. Once you have the API key, do the following:

1. **Store the API Key securely**: Instead of hard-coding the key, put it in the `.env` file or an environment variable (e.g., `OPENWEATHER_API_KEY=your_key_here`). Load it in your config or directly via `os.getenv`. This keeps the key out of source code.

2. **API Call**: Using the `requests` library, call the OpenWeather API. For example, to get current weather for a city by name:

```python
import os, requests

API_KEY = os.getenv('OPENWEATHER_API_KEY')
city = "Chicago"  # or from a config setting
url = f"http://api.openweathermap.org/data/2.5/weather?q={city}
&units=imperial&appid={API_KEY}"
response = requests.get(url)
if response.status_code == 200:
    data = response.json()
    temp = data['main']['temp']
    description = data['weather'][0]['description']
    print(f"{city} Weather: {temp}°F, {description}")
else:
    print("Error fetching weather data:", response.text)
```

This will fetch the weather (temperature, description, etc.). You can parse whatever fields you need. The code in the project was expecting a JSON with `main.temp` and `weather[0].description`, which aligns with OpenWeather's response structure. So adapting the existing `get_weather(city)` function to use this approach is easy: construct the URL with the city and API key, make the GET request, and return a dictionary of the needed fields (or an error message if the request fails).

1. **No OAuth needed**: Note that for OpenWeather, we do **not** use the Google OAuth credentials at all. The `google_credentials_joeltimm.json` is irrelevant to OpenWeather; it's only needed if you were using a Google-provided weather API (which is not common). Therefore, you can remove any Google OAuth token usage from `weather.py`. The `load_weather_credentials()` in `auth/credentials.py` that uses Cloud Platform scope can be deprecated unless you have a specific Google Cloud service for weather. It's simpler to use a direct API key for weather services. This keeps Google OAuth for Google services (Calendar/Gmail) and non-Google APIs with their own keys.

2. **Future Google Weather integration (optional)**: If you intended to use a Google service (like Google Maps API's weather data or a Cloud Function), you would indeed use an OAuth token with a broad scope like Cloud Platform. In that case, you'd call that Google endpoint with the `Authorization: Bearer <token>` header as the code is doing. Ensure that the token has the right scope and the Cloud project has the API enabled. But again, for simplicity and cost, OpenWeather or similar public APIs are easier for a hobby project.

Using the Credentials in Practice

Once you have the tokens and keys set up as described, integrate them into the app:

- **Google Calendar**: The background thread will use the token from `calendar_token.json` to fetch events. Make sure this file is present (after the initial auth). If you're deploying via Docker and didn't include the token file, you might run the auth flow on the device once (perhaps by running the container interactively to perform the OAuth dance, or by copying the token file from your dev machine after generating it). After that, the background job can refresh the token automatically without user intervention.

- **Gmail**: The app doesn't yet have code to send emails, but you can create a scheduled job (perhaps also using APScheduler or even a simple thread + sleep loop separate from the calendar loop) that sends an email daily. Use the `gmail_token.json` in a similar way to get credentials and call the Gmail API. This could be invoked, for example, every morning at 8am to send an update. The scheduling can be done with Python's `schedule` library (which is in requirements) – you could set up `schedule.every().day.at("08:00").do(send_summary_email)` and then run a loop in a background thread to execute pending `schedule.run_pending()` calls. This might be more elegant than writing your own sleep loop.

- **Weather**: Update the `/api/weather` route to call `get_weather(city)` with a configured city. You could, for instance, add an environment variable for `CITY` (or multiple cities). If you only care about one location, hard-coding is fine but putting it in config makes it easier to change. The route could also accept a query parameter for city if you want it dynamic. For now, simplest is to decide on one city (your locale) and use that. Ensure that your OpenWeather API key is available to the container (via .env or docker run `-e OPENWEATHER_API_KEY=...`). The response from `get_weather` can be passed through directly as JSON. The front-end can then display the temperature and condition accordingly.

By following the above guidance, the OAuth 2.0 client credentials file will be effectively used to grant access to Google Calendar and Gmail, and a separate API key will handle weather data.

Conclusion

In this analysis, we identified various code issues in the **kitchen_dashboard** project and provided recommendations to resolve them. Key fixes include correcting import paths and initialization order, eliminating hard-coded secrets, and improving the robustness of background tasks and token management. The Dockerfile, while mostly sound, can be improved with best practices around context management and security. Guidance was also given on how to properly leverage the OAuth 2.0 credentials to integrate Google Calendar and Gmail features, as well as how to retrieve weather information securely via an external API.

By implementing these changes, the project will be more stable, secure, and maintainable. The integration steps will allow the dashboard to display calendar events, send summary emails, and show current weather, providing a complete picture on the kitchen display. It's crucial to test each component after making changes – for example, run the container and ensure that Google API calls succeed (watch the logs for any authentication prompts or errors), and that the weather endpoint returns real data. With the bugs addressed and OAuth flow set up, the **Kitchen Dashboard** will be well on its way to achieving its goal as a modular, smart home information center.

---

[1] Python quickstart | Google Calendar | Google for Developers
https://developers.google.com/workspace/calendar/api/quickstart/python

[2] Best practices | Docker Docs
https://docs.docker.com/build/building/best-practices/

[3] environment variables - Docker container - what's the best practice to inject to it a sensitive data - Stack Overflow
https://stackoverflow.com/questions/54873288/docker-container-whats-the-best-practice-to-inject-to-it-a-sensitive-data