Sprint Report

PORTFOLIO TASK

Unit code: COS40005
Unit Name: Computing Technology Project A
Submission date: 25/11/2025

Student Name	Student Id	Statement of contribution to the report
Nguyen Xuan Bach	SWS00538	
Nguyen Huu Thang	SWD00100	
Ngo Nguyen Phuc	SWD00212	
Nguyen Hung Thinh	SWD00177	
Hoang Nhat Thanh	SWD00133	

1. SPRINT PLAN

Sprint Goal

Full-featured CMS with working AI content generation

Core Module Development

- Event management with AI content generation
- Ticket/helpdesk system
- Form builder and submissions
- Notification system
- Resource and page management

n8n Integration (Production-Ready)

- n8n_client.py with real webhook triggering
- Execution logging and error handling
- Secure callback endpoints with authentication

Advanced Frontend Features

- Events page with "Generate Content" buttons
- Calendar integration using FullCalendar
- Real-time AI content display

API Expansion

- 8 new API endpoints via /api/core/
- Event content generation endpoint
- Webhook callback handling

2. SPRINT PROGRESS

Updated Architecture

The architecture for this sprint was developed and built upon the previous sprint, with major revisions and presentation. The system is modified in a way that the backend is containerized, which saves time for deployment. N8N webhook and gateway have been integrated into the backend service, which is responsible for student/staff AI chatbot conversing the topics of recommendations for semester enrolment (under construction, based on querying the student's performance and preferences), and an automated event/notification creation.

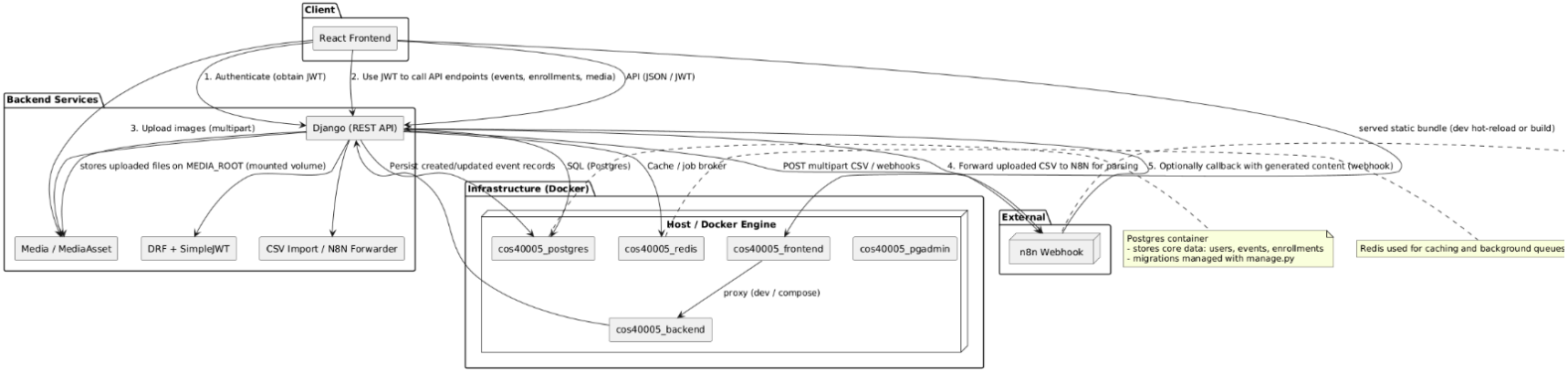


Figure : Updated architecture design

The architectural diagram illustrates a modern, decoupled web application designed around a microservices-inspired, containerized infrastructure. The system is fundamentally divided into four distinct operational zones: the Client interaction layer, the Backend Service core, the underlying Docker Infrastructure, and External Workflow integrations. This separation of concerns allows for a modular development environment where frontend interfaces, business logic, data persistence, and complex data processing workflows operate independently yet cohesively.

The Client Layer and Frontend Interaction. At the entry point of the architecture lies the Client, represented by a React Frontend. This layer is responsible for the user interface and interactivity. It functions as a single-page application (SPA) that communicates with the server primarily through asynchronous API calls. In a development environment, this serves as a hot-reloading module, whereas in production, it would be delivered as a static bundle.

The communication protocol between the client and the backend is strictly defined. The process begins with an authentication handshake; the React client sends credentials to the backend and receives a JSON Web Token (JWT) in return. Once authenticated, the client attaches this JWT to subsequent requests to access protected API endpoints, such as those for events, enrollments, and media. This stateless authentication mechanism ensures that the backend remains scalable, as it does not need to store session data in memory.

The Backend Core: Django and REST API The heart of the application logic is the Backend Services layer, powered by the Django web framework and the Django REST Framework (DRF). This container acts as the central orchestrator for the system. It is equipped with specific sub-components to handle distinct tasks: SimpleJWT manages the authentication token lifecycle, while a dedicated CSV Import/Forwarder module handles data ingestion.

The backend handles complex media management through a specific pipeline. When the client initiates a multipart image upload, Django captures these files and stores them in a designated MEDIA_ROOT. This directory is crucial because it represents a mounted volume, allowing files to persist outside the ephemeral life of the application container. The backend also acts as a proxy and interface for data retrieval, serving JSON responses to the frontend based on SQL queries executed against the database.

Underpinning the entire application is a Docker infrastructure hosted on a Docker Engine. The diagram highlights a customized container ecosystem, likely configured via Docker Compose, where each service runs in isolation but shares a network. The specific containers identified—cos40005_postgres, cos40005_redis, cos40005_frontend, cos40005_backend, and cos40005_pgadmin—suggest a structured academic or enterprise deployment (potentially referencing a specific course or project code "cos40005").

Within this virtualized network, the frontend container (in development mode) proxies requests to the backend container to avoid Cross-Origin Resource Sharing (CORS) issues. This internal networking allows the services to communicate securely without exposing internal ports to the public internet, creating a secure "host" environment.

Data management is handled by two distinct technologies residing within the infrastructure layer. PostgreSQL (cos40005_postgres) serves as the primary relational database, responsible for persisting core entities such as users, events, and enrollments. The architecture also includes pgAdmin (cos40005_pgadmin), providing a web-based GUI for database administration and inspection.

Complementing the SQL database is Redis (cos40005_redis). While PostgreSQL handles long-term storage, Redis is utilized for high-speed caching and functioning as a message broker. This is particularly relevant for background task queues, allowing the main application thread to remain responsive while heavy tasks are processed asynchronously.

A unique feature of this architecture is its integration with n8n, an external workflow automation tool. The system employs an offloading pattern for heavy data processing. When a user uploads a CSV file via the React frontend, the Django backend receives the multipart request but does not parse the file itself. Instead, it forwards the file to an n8n Webhook.

This external workflow engine acts as a specialized worker. It receives the CSV, parses the rows, and executes complex logic such as creating events or sending notifications. Once the processing is complete, n8n can optionally trigger a callback webhook to the Django backend to update the system state with the generated content. This delegation ensures that the main web server is not blocked by CPU-intensive parsing operations, maintaining high availability for user interactions.

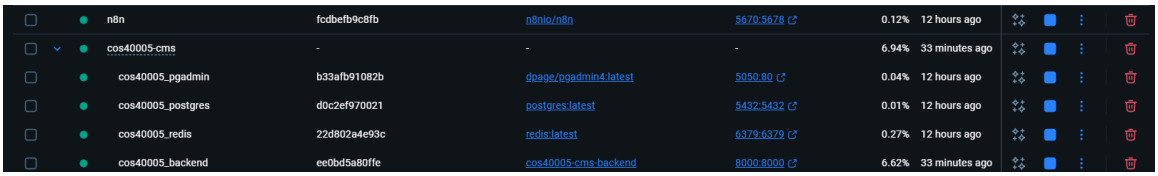


Figure: Deploying using Docker

Sprint 3 comes with major refactors. All backend services are containerized and managed using Docker: n8n, postgres, pgadmin, redis, and the uvicorn Django server.

N8N

On-site Implementation of N8N with Google Cloud Console

The core requirement for the SwinCMS automation suite was to bridge the gap between static planning documents (CSV Semester Plans) and dynamic, time-sensitive execution (Notifications and CMS posts). We selected n8n (nodemation) as the orchestration engine over custom microservices for three primary reasons:

- Containerized Autonomy: Running n8n as a Docker container allows the system to operate entirely within the university's local network infrastructure while maintaining isolation from the host OS.

- Hybrid Logic Capabilities: Unlike standard low-code tools, n8n's "Code Node" architecture allowed us to inject custom JavaScript for complex date parsing (e.g., converting natural language rules like "1 week before" into ISO-8601 timestamps), which was essential for interpreting the Academic Director's input.
- State Abstraction: By utilizing n8n, we avoided the need to build a custom backend interface for event management. Instead, we leveraged the Google Calendar API as a "Headless CMS," storing automation metadata directly within event descriptions.

A significant technical hurdle involved authenticating the Dockerized n8n instance with the Google Cloud Platform (GCP).

The Problem: Standard OAuth2 flows require a public callback URL to return the access token. However, our n8n instance runs inside a Docker container on localhost:5678. When initiating the handshake, Google rejected the request with redirect_uri_mismatch because localhost is not a verifiable public origin.

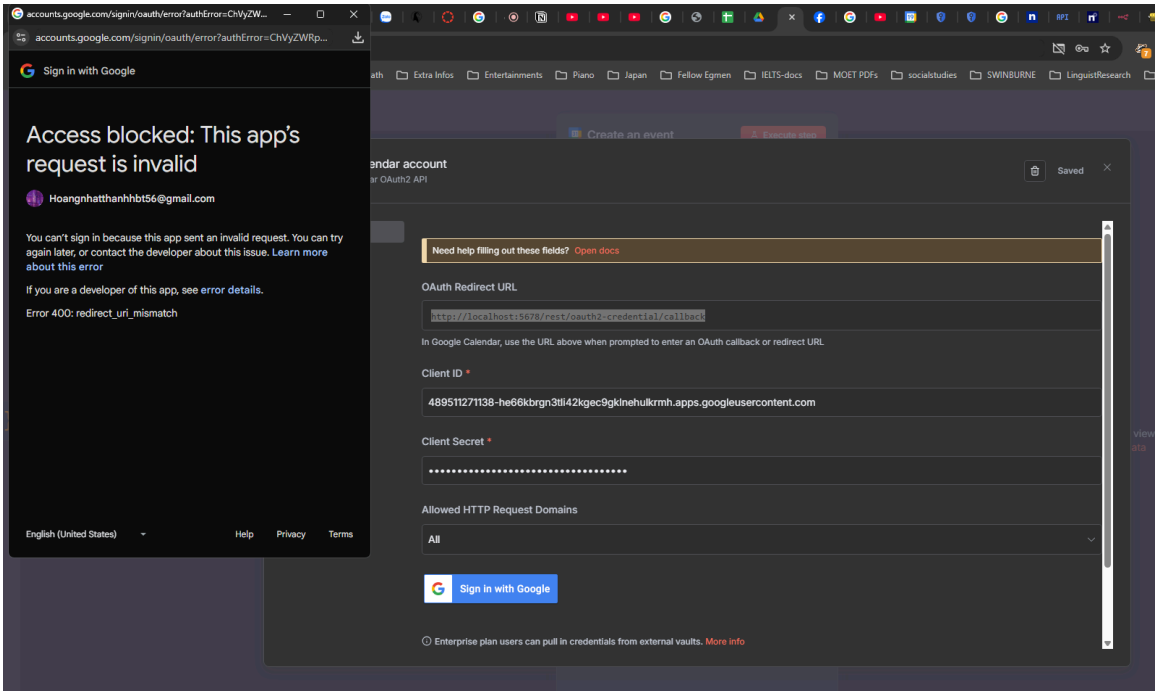


Figure: Using Ngrok for n8n implementation

For this, we have to resort to bring n8n into the internet, we implemented a secure tunneling strategy using *Ngrok* to expose the local container to the public internet solely for the authentication handshake.

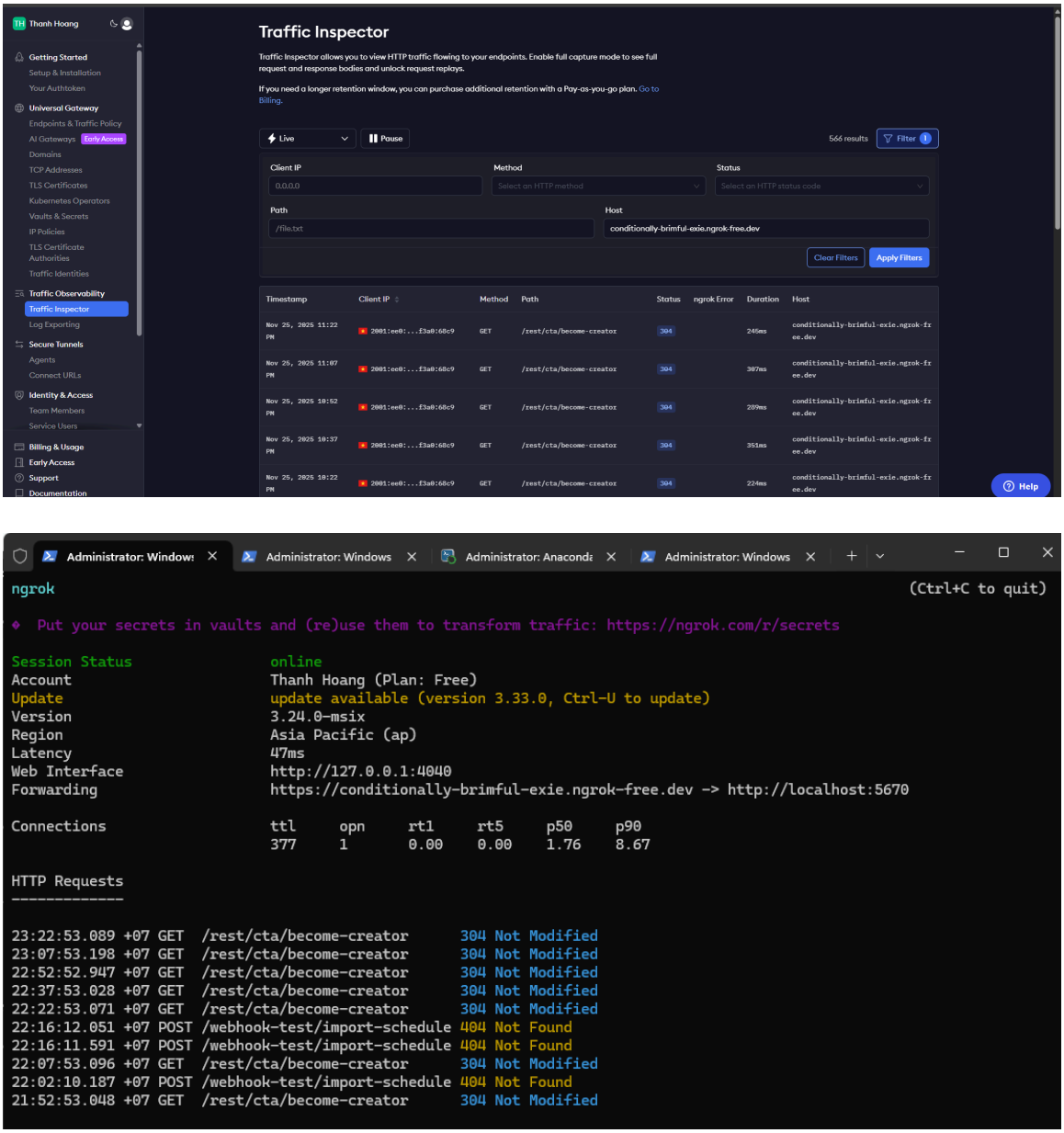


Figure: traffic being forwarded out from port 5670 (which is forwarded from port 5678 from the container itself) to the internet.

```
docker run -it --rm --name n8n -p 5670:5678 -e
WEBHOOK_URL=https://conditionally-brimful-exie.ngrok-free.dev -v
C:\Users\ADMIN\n8n-data:/home/node/.n8n n8nio/n8n
```

The command can be destructured as follows:

- Environment Configuration: the previous container was deleted and a new one was created, hence the -it flag, with the WEBHOOK_URL environment variable pointing to the Ngrok developer domain (https://[id].ngrok-free.app).
- GCP Whitelisting: The Google Cloud Console was configured to strictly accept redirects to the specific path /rest/oauth2-credential/callback.

The integration required writing to the Staff Calendar, which is managed by three parties: FPT, Swinburne, and Personal email accounts, which are enforced and used by students. Staying true to our self-host, self-serve philosophy, we will attempt to simulate this interaction using our personal email, which we will refer to as the “Test User.”.

Access Control Strategy:

Initial attempts to add the internal university email as a "Test User" failed due to Google's "External App" restrictions on enterprise accounts. We implemented a "Delegated Access" pattern to resolve this:

- A personal service account (Gmail) was registered as the primary OAuth connector in n8n.
- The official Staff Calendar was shared with this service account with "Make changes to events" permissions.
- This allowed n8n to modify the official calendar via the service account, bypassing the need for organization-wide API privileges.

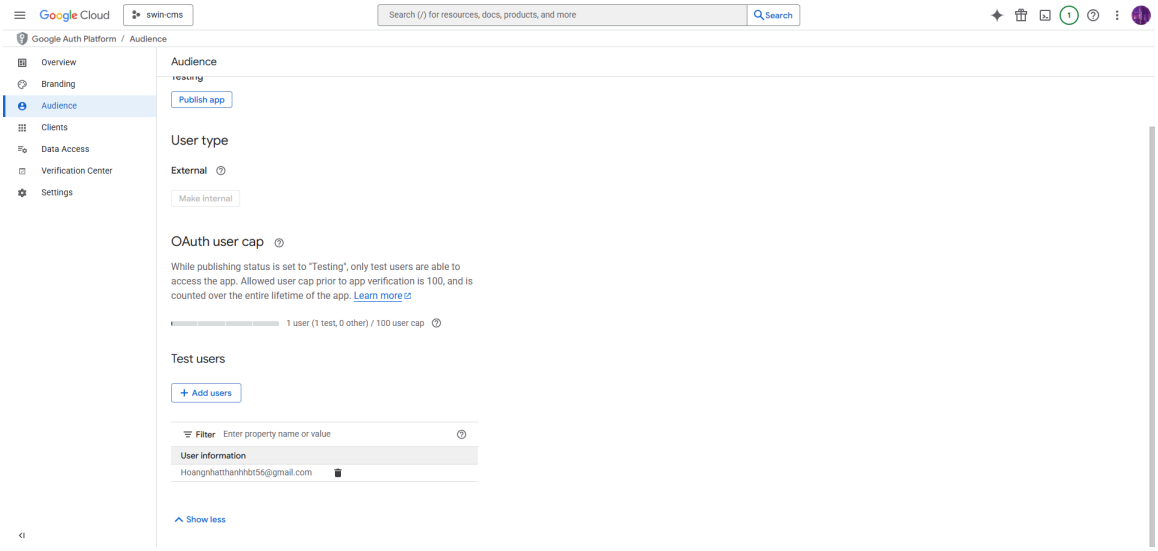


Figure: test account

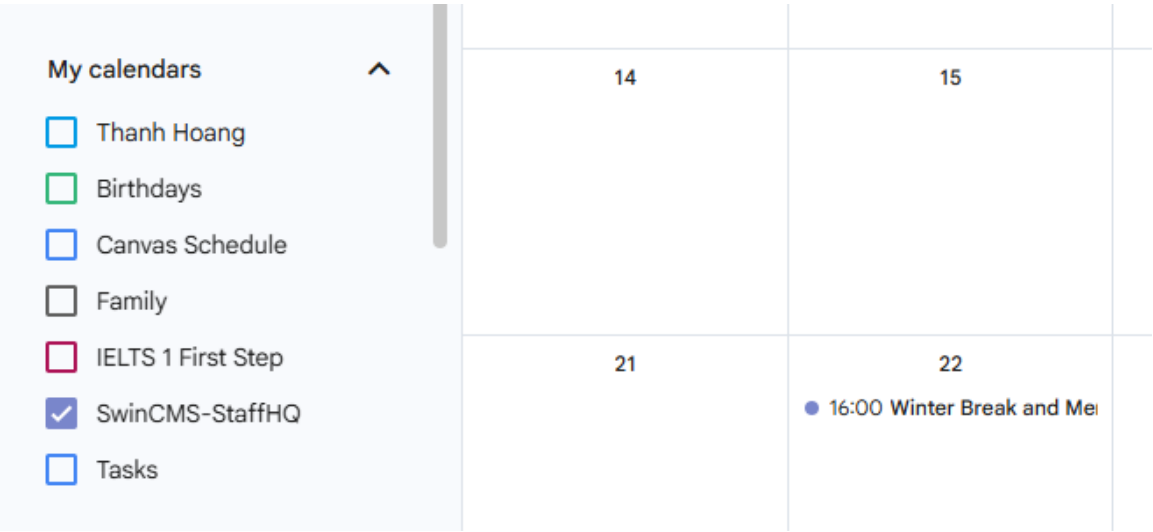


Figure: simulated Swinburne Staff Calendar Space

Container Networking: Linking N8N to Local PostgreSQL

The second phase of the workflow requires enriching event data with student contact information stored in a local PostgreSQL database. Because Docker containers utilize isolated network namespaces, the n8n container could not address the database via localhost.

To bridge this, we utilized the Docker host gateway host.docker.internal.

- **Database Configuration:** We modified postgresql.conf to listen on all interfaces (*) and updated pg_hba.conf to accept traffic from the Docker subnet.
- **Query logic:** The workflow dynamically injects the target audience (e.g., "parents, students") from the Calendar metadata into the SQL query:

```
SELECT email FROM students WHERE role = ANY(STRING_TO_ARRAY($1, ','))
```

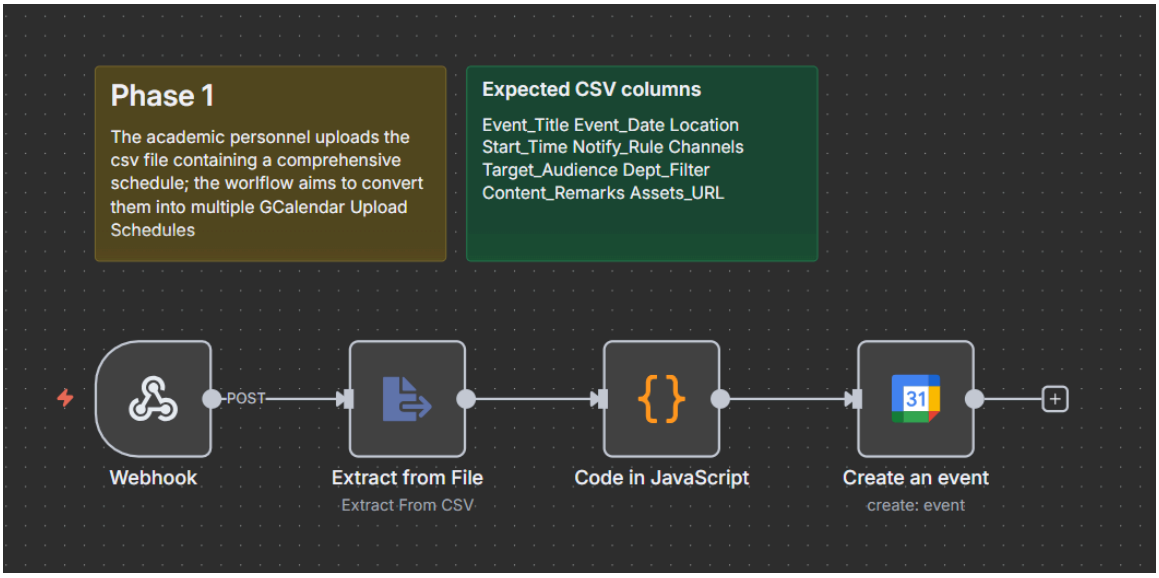
Data persistence used AI applications in N8N (Metadata pattern)

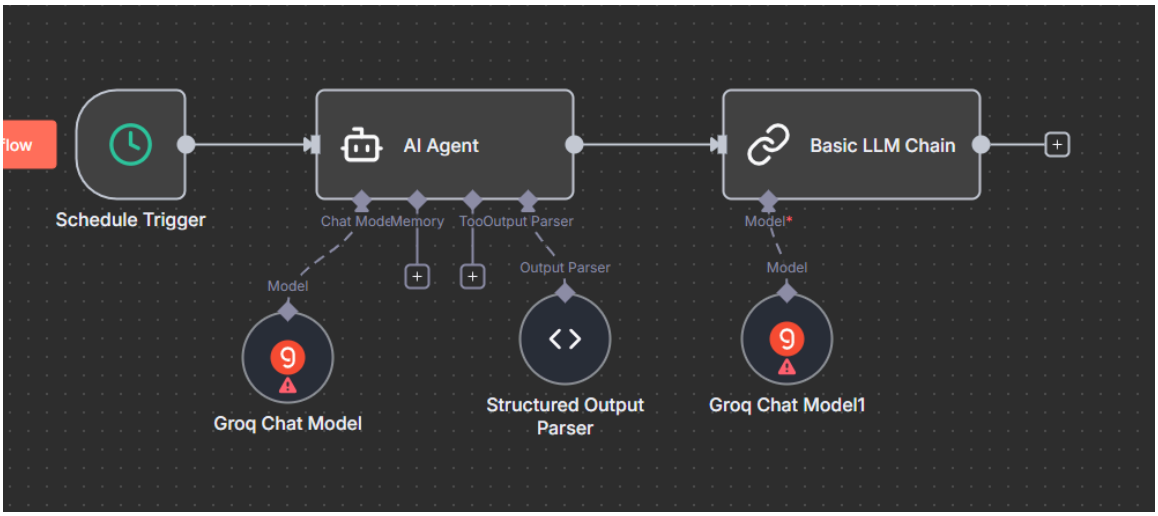
To ensure the system is stateless between runs, we utilized the Google Calendar description field as a JSON store. When the CSV is ingested (Stage 1), the system serializes the execution logic into a hidden JSON string:

```
{
  "cms_data": { "title": "Tuition Fee", "audience": "students" },
  "automation_trigger": { "trigger_date": "2025-12-05" }
}
```

The JSON format above is from example_end_of_semester.csv; please find the file as an attachment to this submission.

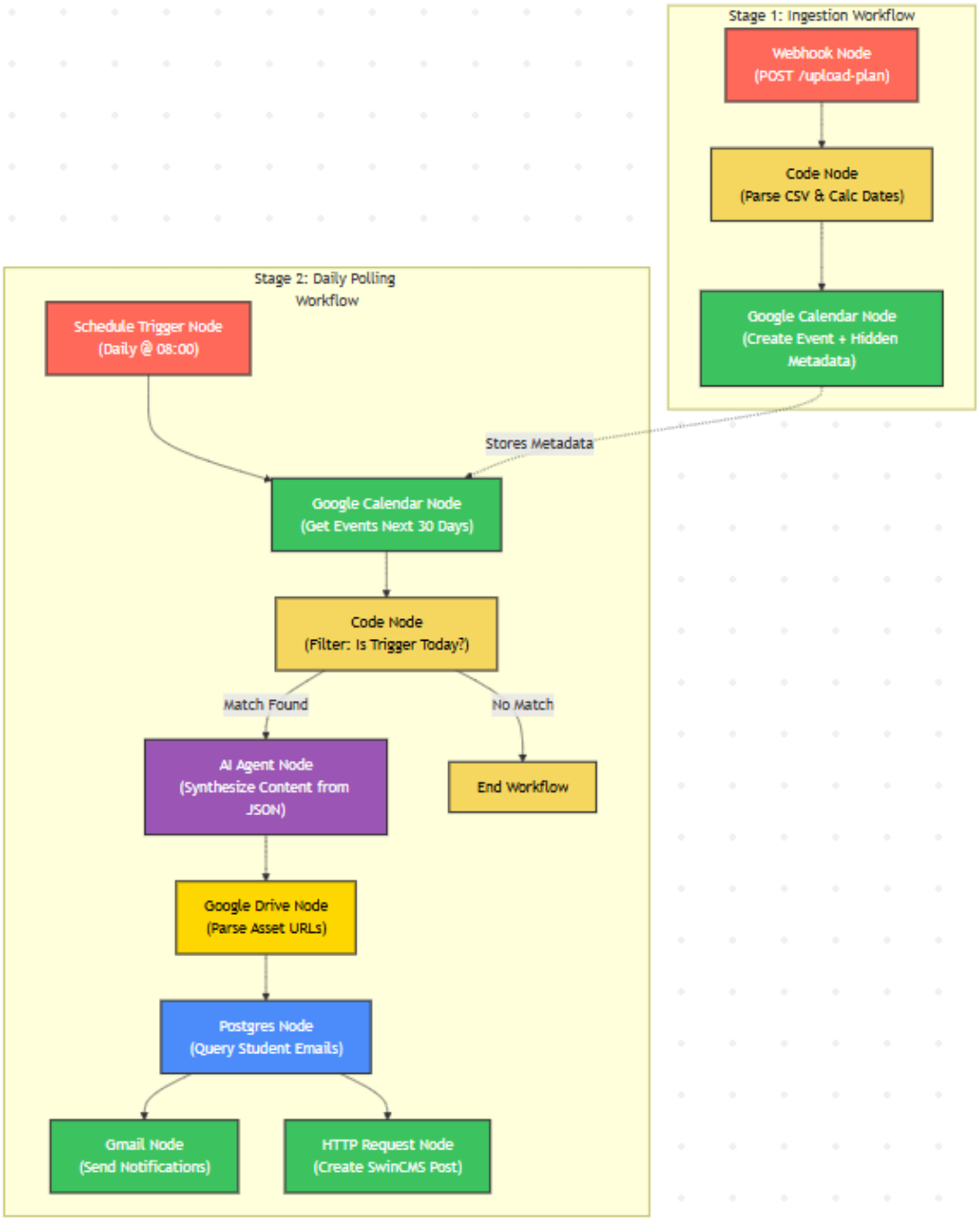
The Stage 2 workflow (Daily Poller) fetches upcoming events, parses this JSON, and executes the notification only if automation_trigger.trigger_date matches the current system date.



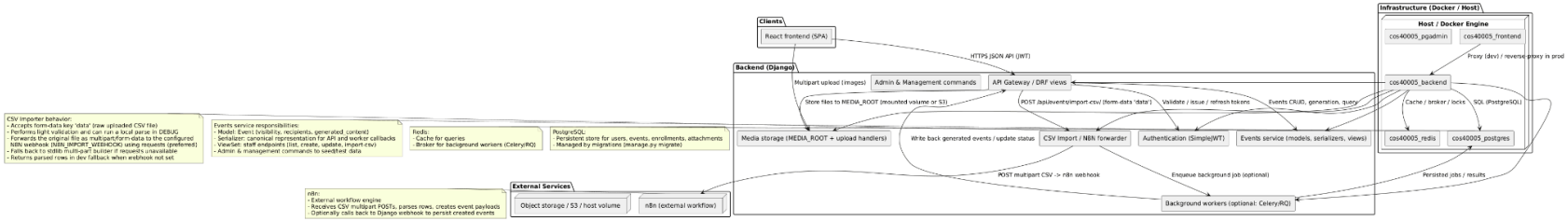


N8N Use Cases

Calendar-based CMS (Content Management System)



Backend



Figure

Architectural Overview of the Event-Driven Django System

From the figure, it can be seen the architecture depicts a sophisticated, decoupled web application designed to handle event management, media handling, and complex data ingestion workflows. The system is built upon a separation of concerns between a reactive client interface, a monolithic but modular Django backend, specialized infrastructure containers, and external workflow automation services.

Client-Side Interaction and Security The user experience is driven by a React Frontend, structured as a Single Page Application (SPA). This client communicates exclusively with the backend via an HTTPS JSON API. Security is enforced through a stateless mechanism where the client authenticates once to obtain a JSON Web Token (JWT). This token is then attached to the header of subsequent requests, allowing the client to interact with protected endpoints. The diagram specifically highlights the API Gateway / DRF Views component within the backend, which serves as the primary entry point for these requests, routing them to the appropriate logic handlers whether they be for fetching event data or initiating complex file uploads.

The Backend Core and Service Logic The application logic resides within the Backend (Django) container. This layer is highly structured, utilizing the Django REST Framework (DRF) to manage data serialization and view logic. A core component here is the Events Service, which encapsulates the business logic for event management. As detailed in the architecture's notes, this service manages the Event model (handling visibility and recipients), uses Serializers to create canonical representations for the API, and employs ViewSets to expose endpoints for listing, creating, and updating records.

A distinct feature of this architecture is its sophisticated handling of CSV data. The system employs a dedicated CSV Import / n8n Forwarder. When the client posts a multipart form containing a raw CSV file to the /api/events/import-csv/ endpoint, the backend acts as an intelligent router. It performs light validation on the file before attempting to forward it to an external n8n Webhook for processing. Crucially, the system is designed with fault tolerance: if the external webhook is unavailable, the backend logic falls back to a standard library implementation to parse the rows locally. This ensures that the application remains functional even if external workflow services are disrupted.

Data Persistence, Media, and Caching The system's data layer is bifurcated to handle different types of persistence. Structured relational data—such as users, events, enrollments, and attachments—is stored in PostgreSQL (cos40005_postgres). This database is managed via standard Django migrations. For performance optimization and asynchronous task management, the system utilizes Redis (cos40005_redis). Redis serves multiple roles: it acts as a high-speed cache for queries, manages distributed locks, and functions as a message broker for background workers (using Celery or RQ) to handle long-running jobs off the main thread.

Media handling is flexible in this design. While the previous iteration emphasized local volume mounting, this architecture introduces the option for Object Storage (S3) alongside the local

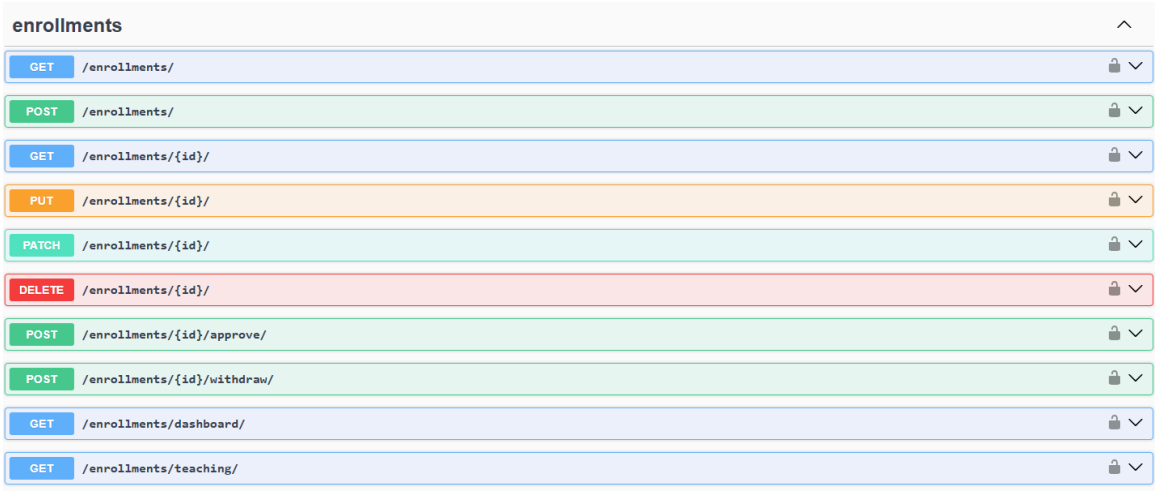
MEDIA_ROOT. This suggests a cloud-ready design where uploaded images and attachments can be offloaded to scalable cloud storage, managed by backend upload handlers.

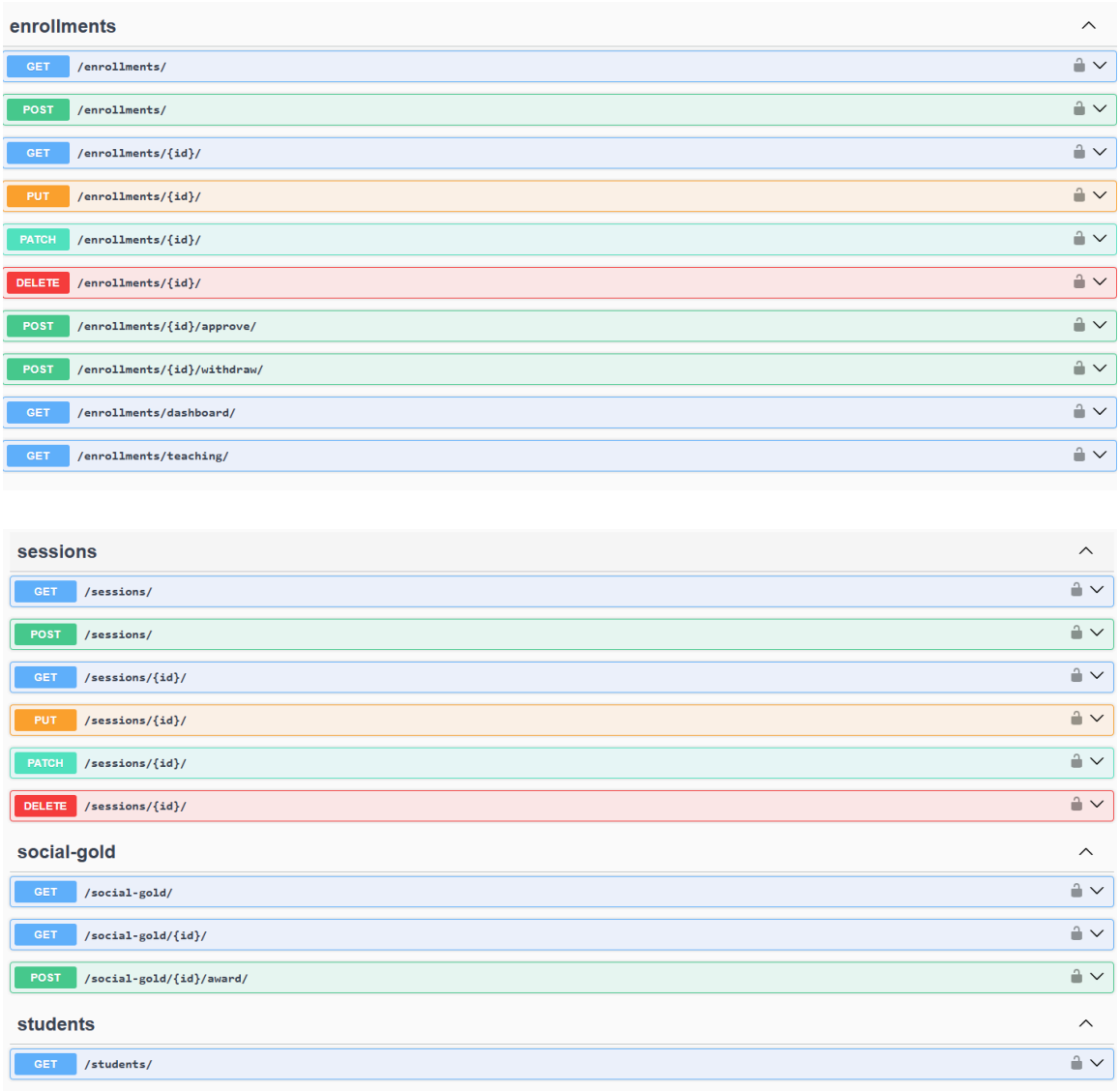
Infrastructure and External Integration The entire application stack is containerized using Docker, orchestrated largely within a host engine. The specific containers (cos40005_backend, cos40005_frontend, etc.) interact through an internal network. In a development setting, the frontend container proxies requests to the backend, while in production, this would likely be handled by a reverse proxy configuration to ensure secure traffic flow.

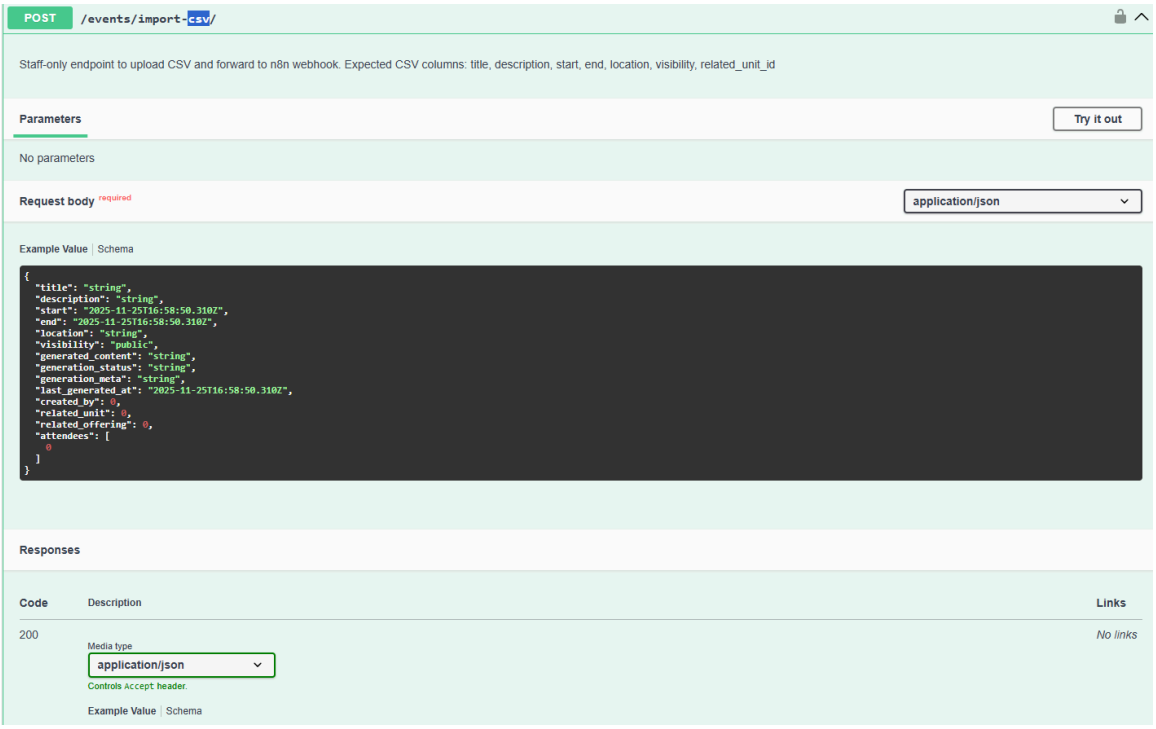
Finally, the architecture heavily leverages External Services to offload complexity. The n8n external workflow engine is pivotal for processing heavy CSV payloads. Instead of the Django backend consuming CPU cycles to parse thousands of rows, n8n receives the file, processes the data, and optionally calls back to the Django API to write the generated events or update status. This "fire-and-forget" or "callback" pattern allows the primary web server to remain responsive to user interactions while heavy data processing occurs in parallel externally.

APIs

So far, the backend has incurred 40 more API gateways; for the sake of brevity, the functionalities of each API are not discussed here; instead, they can be found at http://localhost:8000/api/docs/#/ after launching backend services. API services are neatly divided into multiple components.




enrollments
GET /enrollments/
POST /enrollments/
GET /enrollments/{id}/
PUT /enrollments/{id}/
PATCH /enrollments/{id}/
DELETE /enrollments/{id}/
POST /enrollments/{id}/approve/
POST /enrollments/{id}/withdraw/
GET /enrollments/dashboard/
GET /enrollments/teaching/
sessions
GET /sessions/
POST /sessions/
GET /sessions/{id}/
PUT /sessions/{id}/
PATCH /sessions/{id}/
DELETE /sessions/{id}/
social-gold
GET /social-gold/
GET /social-gold/{id}/
POST /social-gold/{id}/award/
students
GET /students/



Completed (New Features)

Backend Expansion:

Core module: 9 new models (Event, Session, AttendanceRecord, Ticket, Form, Notification, etc.)

n8n Integration:

- src/backend/core/n8n_client.py - Production webhook client
- trigger_workflow() function with API key auth
- N8NExecutionLog tracking with status management

API Endpoints: /api/core/events/, /api/core/tickets/, /api/core/forms/, etc.

Event Content Generation:

- POST /api/core/events/{id}/generate_content/
- POST /api/core/events/{id}/generation_callback/ (webhook endpoint)

Frontend Enhancement:

Events Page (Events.jsx):

- Lists events from /api/core/events/
- "Generate Content" buttons trigger n8n workflows
- Displays AI-generated social posts, brand scores, tone analysis

Calendar Page (Calendar.jsx):

- FullCalendar integration showing student enrollments
- 12-week semester blocks starting 2025-01-05
- Color-coded by course category (core/major/elective)

Database Migrations

- Migration 0002_course_role_scholarship_alter_user_options_and_more.py
- Added all core models and n8n workflow tables

Evidence (Updated)

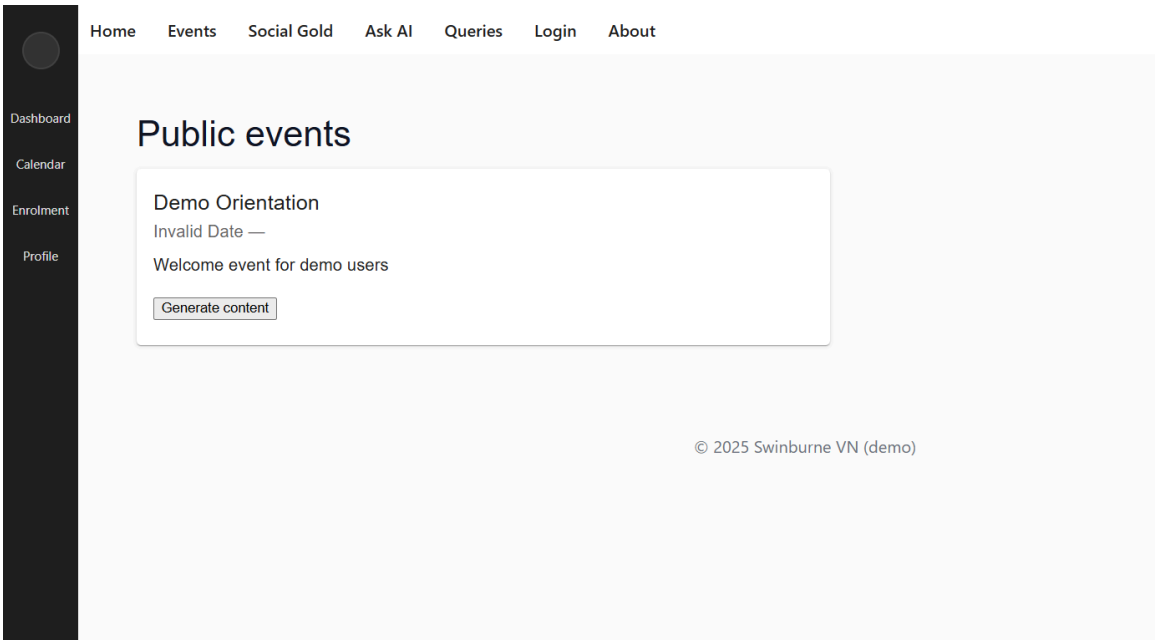


Figure 3:

Figure: Events page showing "Generate Content" functionality

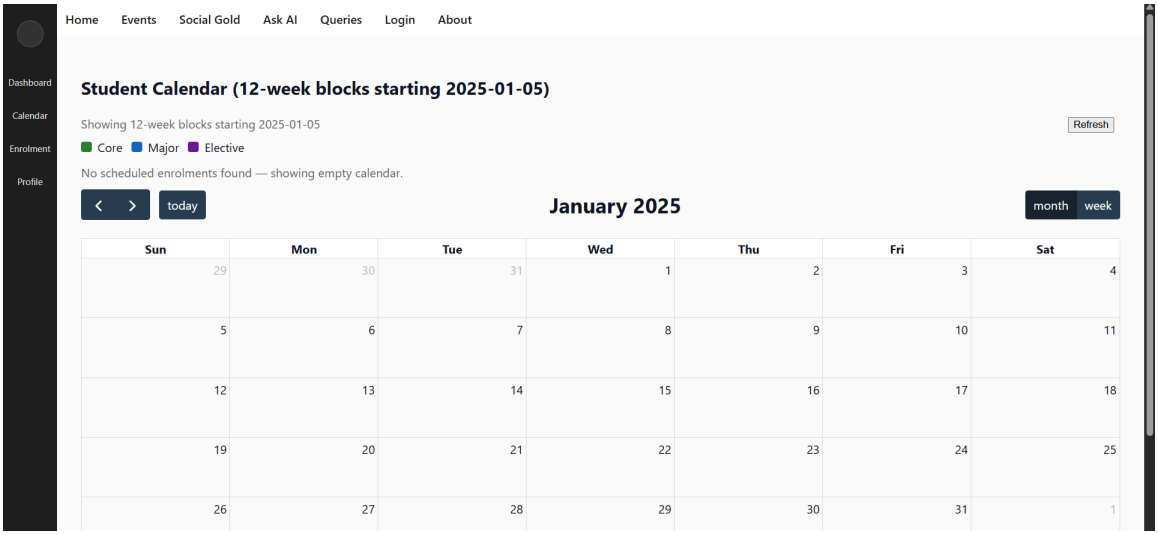


Figure 4:

Figure: Calendar view with color-coded enrollments

AI N8N testing

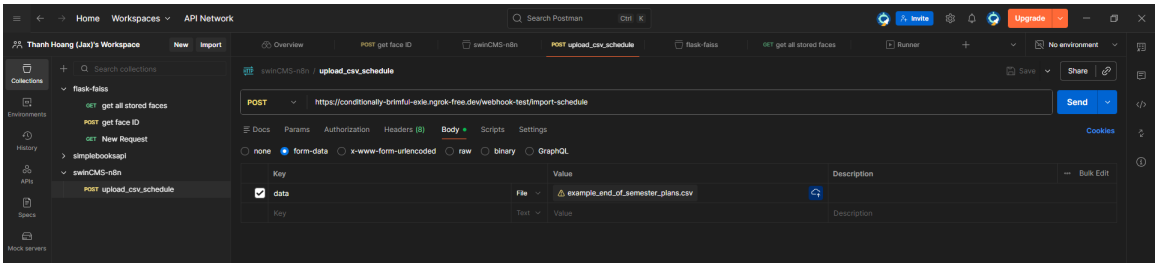


Figure: a request to N8N to parse a planned csv schedule can be made using POST request.

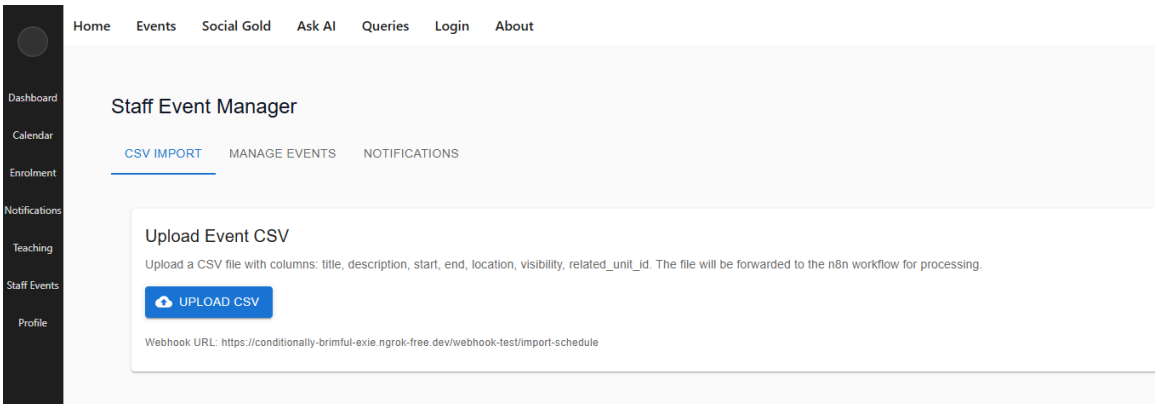


Figure: or interactively on the web

Figure: the staff can interactively (with the help of AI) can refine the content from the JSON format.

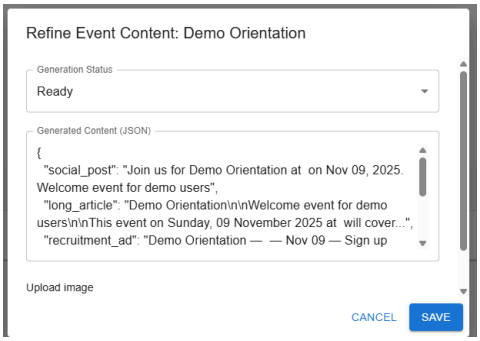


Figure: event content refinement.

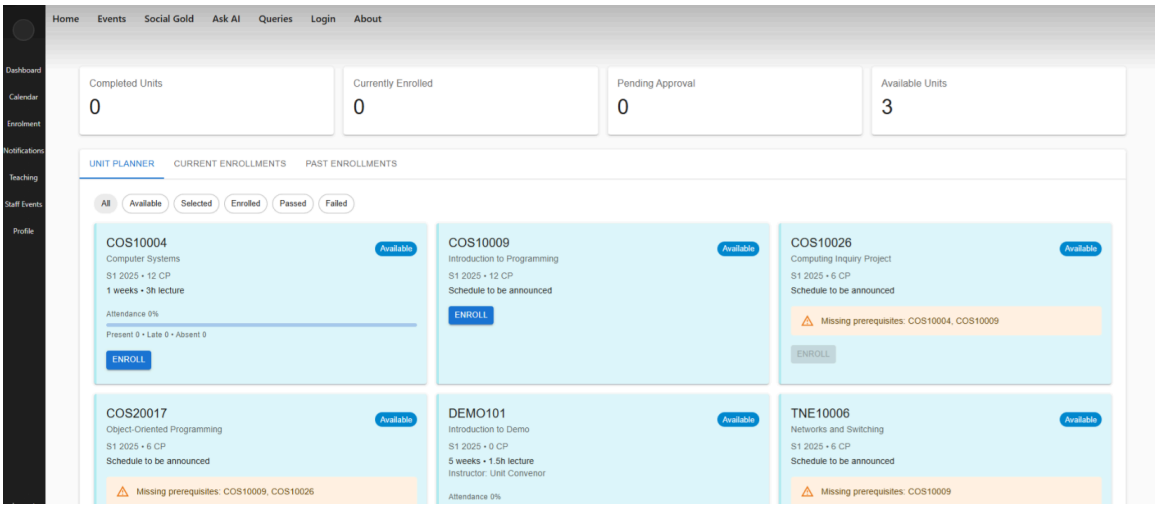


Figure : Unit for Enrollment

Figure : Notification and events

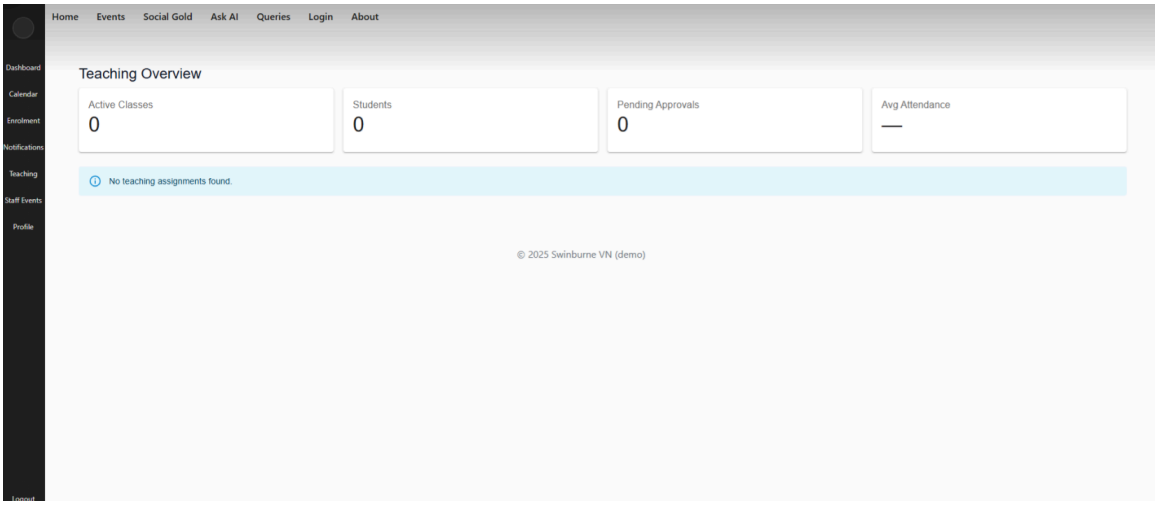


Figure: Teacher's dashboard

3. Sprint Demonstration

Demo Flow (Updated)

Basic Stack (same as before)

- Docker Compose services running
- Frontend/backend connectivity

New Core Features

- Navigate to Events page: show event listing
- Click "Generate Content" → demonstrate n8n workflow trigger
- Show generated social posts with metadata (tone, brand score)
- Navigate to Calendar → show FullCalendar with enrollment data

API Demonstration

- GET /api/core/events/ - event listing
- POST /api/core/events/1/generate_content/ - AI generation
- Show Django admin with N8N execution logs

n8n Integration

- Show webhook URL configuration in N8NWorkflow admin
- Demonstrate callback endpoint security
- Show execution logging and error handling

Technical Highlights

- Real AI Integration: Not just UI mockups, actual workflow triggering
- Production Security: Webhook secret validation, API key auth

- Comprehensive Logging: Full audit trail of n8n executions
- Modern UI: FullCalendar, MUI components, real-time updates

4. Sprint review (Critical review of the product)

Plan vs Actual (Major Improvements)

- Exceeded expectations: Delivered full CMS functionality beyond original scope
- n8n Integration: Moved from scaffolding to production-ready implementation
- Frontend UX: Significant upgrade from "too basic" to feature-rich interface
- API Completeness: Comprehensive CRUD operations for all core entities

Key Achievements

- Functional AI Workflows: Events can trigger real n8n workflows for content generation
- Calendar Integration: Students can visualize their semester schedule
- Ticket System: Complete helpdesk functionality for student support
- Form Builder: Dynamic form creation and submission handling

Challenges Addressed

- Previous feedback "too basic": Now has comprehensive CMS features
- n8n integration: Moved from models-only to full workflow execution
- Frontend richness: Added calendar, AI generation UI, better UX

Client/Instructor Feedback Response

5. Retrospect (Critical review of the process)

[This section should present a summary on how the overall team process was followed. Challenges, bottlenecks found during the sprint, how dealt with etc.]

What Went Exceptionally Well

- Feature Velocity: Delivered comprehensive CMS in one sprint
- Integration Success: n8n workflows working end-to-end
- Code Quality: Clean separation of concerns, proper error handling
- Documentation: Added README_EVENTS_GENERATION.md for n8n patterns

Technical Decisions

- Modular Architecture: Core module separate from users/academic/enrollment
- Security-First: Webhook authentication, API key management
- Frontend Libraries: FullCalendar for professional calendar UX

- Error Handling: Comprehensive logging and fallback mechanisms

Process Improvements Made

- Better Planning: Clear separation between scaffolding and implementation
- Documentation: Inline code docs and architectural decisions
- Testing: Added test files for core functionality

6. Lessons learned (Critical review of SPRINT one experience and future plan)

[Add a list of lessons learned from working together on the particular project in the group. List a clear set of to-do/recommendations for improvement in the future. It is strongly recommended that you review and reflect (based on experience) on your team plan, communication plan, quality plan, and risk registry. Teams should be referring to the ethical considerations for any data collection, and/or any development that took place in the sprint.]

Technical Learnings

n8n Integration Patterns:

- Webhook-first design for async processing
- Execution logging crucial for debugging
- Fallback to local generation when workflows fail

Frontend State Management: Real-time updates after AI generation

Calendar Complexity: Parsing enrollment schedules into recurring events

Architecture Decisions

- Core Module Design: Generic models (Event, Ticket, Form) serve multiple use cases
- API Security: Separate authentication for user APIs vs webhook callbacks
- Database Schema: JSON fields for flexible content storage

Next Sprint Priorities

- Production Deployment: CI/CD pipeline, environment configs
- Advanced AI Features: Multi-channel content (email, articles, ads)
- User Experience: Form builder UI, ticket management interface
- Performance: Caching, database optimization
- Testing: Comprehensive test suite for core functionality

Risk Mitigation

- n8n Dependency: Implemented fallback generation when workflows unavailable
- Data Privacy: Secure webhook endpoints, no PII in logs
- Scalability: Async workflow processing, proper error handling