ChatGPT

Academic Timetable Generator – System Design & Implementation Guide

Development Roadmap

1. **Requirements & Planning:** Conduct stakeholder interviews (admins, department heads, teachers, students) to capture needs: e.g. automatic conflict-free schedule generation, multi-campus support, role-based views, and OCR-based data input. Define core goals like supporting schools vs colleges differently, and drafting an agile milestone plan (MVP, iterative features). Clarify the tech approach (e.g. consider a Backend-as-a-Service like Supabase, given its built-in Postgres database, auth, and storage [1] [2]). Outline differentiators (AI assistance, mobile PWA, analytics) and sketch user stories.

2. **Technology Stack Selection:** Choose modern, scalable tools. For the frontend, use a component-based framework (e.g. React or Angular) with a responsive UI library (Bootstrap or Material UI). On the backend, options include Node.js/Express or Python/Django, but consider Supabase as a Backend-as-a-Service to accelerate development: Supabase provides a hosted PostgreSQL with real-time data, built-in authentication, and file storage [1] [2]. This lets you bypass much boilerplate for user management and data persistence. For OCR, plan either an open-source engine like Tesseract or a cloud OCR API (Google Vision/Azure) to parse uploaded timetable images. Use a scheduling/constraint solver library (e.g. Google OR-Tools for constraint programming) or build a custom backtracking/heuristic algorithm for initial timetable generation.

3. **System & Architecture Design:** Adopt a multi-tier, multi-tenant architecture. For example, a three-tier design:

4. **Presentation Tier:** A responsive web UI (React/Angular) that adapts to desktop and mobile. Different dashboards and editors render based on user role (Admin, Department Head, Faculty, Student).
5. **Application Tier:** Server-side logic (Node.js/Express or Supabase Edge Functions) exposing RESTful (or GraphQL) APIs. This layer handles authentication, business rules (schedule constraints, conflict checking), OCR processing, and real-time updates. If using Supabase, leverage its instant APIs and Edge Functions for custom logic. All endpoints must be secured (HTTPS + JWT auth) with role-based permissions.
6. **Data Tier:** A PostgreSQL database (for example, a Supabase-managed Postgres instance [1]) storing all entities. Use Supabase Storage for handling file uploads (images, documents) [2]. Tag each record with an `institution_id` or tenant key to isolate data between institutions. Enforce referential integrity with foreign keys. Employ Supabase's Row-Level Security (RLS) or custom middleware to ensure multi-tenant isolation and RBAC.

Multi-Tenant Strategy: Implement a **shared-database, shared-schema** approach by default (one Postgres DB with an `institution_id` column on each table). This simplifies management and allows Supabase's

authentication data and policies to naturally tie to tenant IDs [1]. For enhanced isolation, consider separate schemas per tenant or separate databases; Supabase even supports migrating existing apps to its platform if needed [1]. For multi-campus, include a `Campus` table linking to `Institution`; each timetable entry then belongs to a campus, enabling synchronized schedules across sites. Ensure every data query is scoped by the current user's institution/campus.

Scalability & Security: Containerize services (e.g. using Docker). Plan deployment on a cloud host (Render, AWS, Azure, etc.) with load balancing and auto-scaling. Implement CI/CD pipelines for automated testing and deployment. Use HTTPS everywhere, strong password hashing (bcrypt), and input validation to guard against common vulnerabilities.

Recommended Tech Stack

- **Frontend:** React.js (with React Router for navigation) or Angular. Use a UI kit like Bootstrap or Material-UI for responsive design. Optionally implement as a PWA for mobile access. Manage global state with Redux or Context API.
- **Backend:** Node.js/Express (or Python/Django) paired with Supabase. If using Supabase, you can use its RESTful APIs or PostgREST, plus Edge Functions for custom logic, instead of building a full custom server. Protect endpoints with JWT/OAuth; Supabase Auth handles user sign-up/sign-in and integrates with Postgres for RBAC [1].
- **Database:** PostgreSQL (Supabase's hosted DB) for structured data. Use Supabase's built-in Postgres for automatic backups, extensions, and real-time updates [1]. Implement indexes on frequent query fields (e.g. `teacher_id`, `day_of_week`). Define all relationships (e.g. `users.role_id` → `Role.id`). If not using Supabase, a managed Postgres/MySQL is recommended.
- **File Storage & OCR:** Use Supabase Storage (cloud bucket) for uploading timetable images and other documents [2]. Supabase Storage supports uploading/serving large files with access controls. When an image is uploaded, trigger OCR: either call Tesseract (an open-source OCR engine) or a cloud OCR API. *Tesseract* is a free, Apache-licensed OCR that extracts text from images [3]; it works well on clean scans but may need image preprocessing. Alternatively, use **Google Cloud Vision** or **Document AI** to extract text/layout from photos (Google Cloud's OCR tools give access to powerful ML models for images [4]). The OCR output (class names, room IDs, times) is parsed and inserted into the schedule database.
- **Scheduling / AI:** Start with a constraint-satisfaction approach: treat each class as a variable to assign (time slot, room, teacher), with constraints (no double-booking, availability, capacity, breaks). For complex cases, integrate Google's OR-Tools CP-SAT solver to optimize assignments (minimize conflicts, gaps) or use a genetic algorithm. Build hooks so that in future an AI/ML model (possibly a Python microservice) can refine or suggest schedules.
- **Real-Time & Notifications:** Use Supabase's Realtime subscriptions to push database changes (e.g. timetable updates) to connected clients [2]. Alternatively, integrate WebSockets (Socket.io). For notifications, use email (with NodeMailer or a service like SendGrid) and in-app alerts. Prepare for future mobile push (e.g. Firebase Cloud Messaging) for critical updates.
- **Export & Reports:** For PDF timetables, use a library like jsPDF (client-side) or PDFKit (server-side). For Excel reports, use SheetJS (xlsx) or ExcelJS. Visualize analytics (room usage, conflict charts) with Chart.js or D3.js on the dashboard. Ensure users can download or email exports.
- **DevOps & Security:** Maintain infrastructure-as-code (Dockerfiles, Kubernetes/Render configs). Set up automated testing via CI (GitHub Actions, Jenkins). For logs and monitoring, use tools like

Prometheus/Grafana or hosted services (Datadog). Document all APIs with Swagger or Postman. Always run behind HTTPS and apply best practices (CORS, helmet, sanitization).

System Architecture & Multi-Tenancy

The system follows a typical three-layer web architecture with multi-tenant considerations:

- **Presentation Layer:** A responsive web frontend (React/Angular) accessed in-browser or on mobile. The UI offers **role-specific views**: e.g. Admins manage institutions and global settings; Department Heads edit department timetables; Faculty see personal schedules and submit change requests; Students see enrolled class schedules. Navigation and components (login forms, dashboards, timetable grids, forms) should adapt to user roles.
- **Application Layer:** The server-side (Node/Express or Supabase functions) implements business logic. Authentication is handled by Supabase Auth (backed by Postgres) or a similar JWT/OAuth system [1]. Each API endpoint enforces role-based access (RBAC) – for instance, only Admins can add new institutions or users in other roles. The layer also coordinates the scheduling engine, constraint management, OCR processing, and notifications. It uses Supabase's instant REST APIs for CRUD on database tables or custom SQL/Edge Functions for complex operations.
- **Data Layer:** All data resides in a SQL database (e.g. Supabase's PostgreSQL). Tables include institutions, users (linked to roles and departments), courses, classes, rooms, timeslots, and timetables. Each record carries a tenant identifier (`institution_id`) so that queries can be scoped per organization. Supabase Storage holds uploaded files (images, CSVs) [2]. Use **Row-Level Security (RLS)** policies or WHERE clauses to ensure each tenant only accesses its own data.

Multi-tenant Strategy: A **Shared Database/Shared Schema** model is simplest: one database for all institutions, with a tenant ID column on each table to segregate data. Supabase supports this pattern well via RLS and request-time context. For stricter isolation, one could use separate schemas per tenant or even separate DB instances, but this adds maintenance. The architecture should allow easy onboarding of new institutions by inserting a new tenant record and assigning resources (campuses, depts, etc.) under it.

Role-Based Access: Upon login, Supabase Auth provides user info (including role). The frontend and backend enforce that, for example, only Admin roles see global settings pages, only Department Heads can run the scheduling algorithm for their department, etc. Conditional rendering hides UI elements (like "Institution Settings") from unauthorized roles.

Integration & Scalability: All services expose clear APIs, making it easy to plug in additional modules or microservices (e.g. an AI optimization service). Using Supabase's Edge Functions (serverless) can offload OCR or ML jobs without running a separate server. Deploying on a cloud platform (Render, AWS, etc.) with container orchestration ensures the app can scale horizontally. In the future, a serverless function could be triggered on each image upload (Supabase Storage Event Trigger) to invoke OCR, demonstrating how the architecture is ready for growth and third-party integrations.

Database Schema (Key Tables)

The relational schema (in Postgres/Supabase) should include, at minimum:

- **Institution:** `(id, name, type, address, contact_info, settings…)` – records each school/college and its default rules.
- **Campus:** `(id, institution_id, name, location, contact…)` – supports multi-site institutions. Linked to Institution.
- **Department:** `(id, campus_id, name, head_user_id…)` – academic department (e.g. Math Dept). Linked to a Campus.
- **Role:** `(id, name)` – defines roles (Admin, DeptHead, Faculty, Student).
- **User:** `(id, institution_id, campus_id, dept_id, role_id, name, email, password_hash, etc.)` – all users of the system. Each user belongs to one institution (and optionally a campus/department).
- **Course:** `(id, institution_id, dept_id, code, title, hours_per_week, ...)` – academic course definition.
- **Class (Course Offering):** `(id, course_id, academic_term, section, elective_group_id, ...)` – specific class sections (e.g. "Math 101 – Section A").
- **Room:** `(id, campus_id, name, type (lecture/lab), capacity, ...)` – physical rooms.
- **TimeSlot:** `(id, campus_id, day_of_week, period_number, start_time, end_time)` – defined slots (possibly with breaks) per campus.
- **TimetableEntry:** `(id, class_id, timeslot_id, room_id, teacher_id)` – a scheduled instance linking a class to a time, room, and instructor.
- **FacultyAvailability:** `(id, teacher_id, timeslot_id, is_available)` – indicates when a teacher can (or cannot) teach, used by the scheduler.
- **ElectiveGroup:** `(id, name, description)` – groups of classes (e.g. electives) to ensure students don't have overlapping choices.
- **InstitutionRule:** `(id, institution_id, rule_type, value)` – configurable rules (e.g. "max_classes_per_day", "lunch_start", etc.).

All foreign keys should be enforced (e.g. `User.role_id → Role.id`) and indexed (e.g. on `teacher_id`, `day_of_week`) for performance. If using Supabase, this schema lives in its Postgres DB [1], and Supabase's dashboard or CLI can be used to manage migrations. This design fully supports CRUD on each entity, constraint editing, and querying timetables by institution, campus, or department.

Component & Module Breakdown

- **Authentication & RBAC Module:** Use Supabase Auth (or a similar service) for user sign-up/login. It supports email/password, OAuth, etc., with data stored in Postgres [1]. Upon login, issue JWTs. Implement middleware to check roles on every API request (e.g. only admins access institution-level APIs).
- **Institution & Profile Module:** Manages onboarding of new institutions (forms to enter school name/type, etc.) and default settings. Includes CRUD for Institution, Campus, Department, with templates for default rules. The Admin user for a new institution is created here.
- **User Management Module:** CRUD operations for users. Admins can add/edit teachers, students, and department heads, assigning roles and linking to campuses/departments. Integrate with

Supabase Auth for secure user creation. Ensure password hashing (e.g. bcrypt) and unique email constraints.

- **Course/Class Management Module:** CRUD for courses and class offerings. Allows bulk import via CSV/Excel (e.g. using SheetJS) to add many courses or sections at once. Elective groups can be managed here. Each course links to a department; classes link to courses. Total hours and elective flags are editable.
- **Room & Timeslot Module:** Admin defines room inventory (with types: lab, lecture hall, etc.) and scheduling slots (periods in each day). The module allows setting break periods or special times. Use a UI to add timeslots and tag rooms by campus. Manage capacity and features (e.g. projector).
- **Constraint Editor Module:** Interfaces and APIs for all scheduling constraints. For example, a calendar UI for each faculty to mark unavailable slots; forms to set max classes per day or mandatory free periods; grouping courses into elective clusters. All constraints are stored in the DB (FacultyAvailability, InstitutionRule, ElectiveGroup tables) for use by the scheduler.
- **Scheduling Engine Module:** Core logic that generates the timetable. When "Generate Timetable" is triggered, this module takes classes and attempts to assign them to timeslots and rooms without conflicts. Start with a heuristic/backtracking algorithm in Node.js: for each class, try to assign an available teacher, room, and time that meets constraints. Integrate Google OR-Tools (CP-SAT solver) for optimized results if needed. After generating, it checks for conflicts (no teacher/room double-booking, capacities met). Any unresolved issues are logged for review. Provide endpoints to trigger generation or fetch its status.
- **OCR & Data Import Module:** Handles uploads of existing timetable data. For spreadsheets, parse with SheetJS or ExcelJS. For images, leverage **Supabase Storage**: when an admin uploads a timetable image, it's saved to a storage bucket [2]. An OCR service (either a dedicated microservice or a Supabase Edge Function) reads the image using Tesseract OCR [3] or calls a cloud OCR API (Google Cloud Vision) to extract text and table structure. The parsed output (list of class-time-room assignments) is then validated and inserted as TimetableEntries. Also allow CSV/Excel uploads of teacher lists, course catalogs, etc.
- **Notification & Real-Time Module:** Use Supabase Realtime subscriptions to push updates (e.g. timetable changes) to connected clients [2]. When schedules are updated, broadcast the new entries. For push notifications, integrate a service (Firebase, or email). For example, if a teacher requests a swap, send an email via NodeMailer or a service API. The UI should display in-app alerts for recent activity.
- **Export & Reporting Module:** Provides exports of timetables and analytics. Implement endpoints or UI actions to generate a PDF (using jsPDF on client or a server-side lib) and Excel file of the current timetable. For reports, query usage statistics (e.g. room utilization rates, number of conflicts) and feed into chart components (Chart.js or D3). Offer CSV/Excel download for raw data as well.

- **Analytics & Monitoring Module:** Collect metrics (like scheduling success rates, average teacher load) and display on admin dashboards. Use charts (e.g. heatmaps of busiest periods) to help with planning. This module might run periodic analytics queries or use a separate reporting database. Also include server monitoring (CPU, DB performance) via Prometheus/Grafana or a cloud monitoring service.

- **Frontend Modules:** Structure the UI into role-based sections. For example:

- *Admin UI:* Institution settings, global analytics, user/course/room management screens.
- *Department Head UI:* Schedule editor (grid view of days vs periods for their departments), constraint editor.

- *Faculty UI:* Read-only timetable of their classes, with a “Request Change” form.
- *Student UI:* Personalized class schedule and iCal export.

- *Shared Components:* Navbar, notifications panel, common forms (login, profile edit), and utilities (e.g. date pickers).

- **DevOps Module:** Infrastructure code and pipelines. Write Dockerfiles for the app (and separate one for any worker or OCR service). Configure Kubernetes manifests or Render/Azure Web App settings. Set up CI (e.g. GitHub Actions) to lint, test, and deploy on each commit. Use the Supabase CLI to manage the database schema and migrations. For logging, use a library like Winston and ship logs to a central service (or use Supabase’s own logging if available).

Each module should be developed independently and communicate via the defined APIs, allowing teams to work in parallel. Use agile sprints to implement and test each module, ensuring a modular, maintainable codebase.

Sample UI Wireframe Descriptions

- **Onboarding Screen:** A multi-step form where the super-admin of a new institution enters details (institution name, type – school/college, address) and creates the first Admin user. Next steps include adding campus names/locations. Use progress indicators. After completion, redirect to the Admin Dashboard.

- **Admin Dashboard:** Presents an overview (e.g. number of campuses, departments, pending tasks). A sidebar or top menu links to key sections (Institution Settings, User Management, Master Data [Courses, Rooms, Timeslots], Reports). The main pane might show summary charts (e.g. pie chart of room usage) and a notification feed (e.g. “2 teacher swap requests pending”). Use cards for quick stats. Ensure the layout is responsive and works on tablets.

- **Department Head View:** Displays a timetable grid for the semester (days vs periods) with classes populated. Each cell shows “Course – Teacher”. Above or beside the grid are action buttons: “Generate Timetable (Manual)” and “Import from Image”. Clicking **Import from Image** opens a file uploader: after selecting an image (JPEG/PNG of an existing timetable), the system runs OCR and fills the grid automatically (this is the “OCR from images file upload” feature). When generating, show a spinner; afterwards, highlight any conflicts in red. A sidebar lists constraint toggles (e.g. a mini-calendar to mark Teacher availability). Admins can drag-and-drop classes in the grid to manually adjust.

- **Faculty View:** Shows only that teacher’s weekly schedule in a grid. Below it is a “Request Change” button. Clicking it opens a form: the teacher picks a class period they want to swap, selects an alternative slot (from available slots), and enters a reason. They submit and get a notification when an admin approves or rejects. The current period is highlighted (e.g. green) in real-time using a clock.

- **Student View:** Provides a simple list or grid of their enrolled classes with times/rooms. Optionally offer an “Export to Calendar (iCal)” link. The UI is clean and mobile-friendly (students often use phones). It also reflects any changes live (if a class is moved).

- **Schedule Generation Interface:** For admins or heads, a page that lets the user choose between **Manual** (interactive editing) or **AI-Assisted** generation. In AI mode, present parameters (e.g. “Prioritize core courses in the morning”, “Enforce 1-hour lunch after 5th period”). On submit, call the scheduler/LLM. Display a progress indicator, then show the resulting timetable with metrics (“Conflicts: 0”, “Average teacher load: X”). Include an “Export” button (PDF/Excel) and an “Email Schedule” option.

- **Constraint Settings Screen:** A form-based interface listing all editable rules. For example: dropdowns or toggles for “Break Periods” (e.g. lunch 12–1pm), numeric inputs for “Max classes per day” per teacher, checkboxes for “No classes after 4pm on Wednesdays”. A table of **Elective Groups** lets the admin name groups and assign classes to them (preventing student conflicts). Each teacher has a mini weekly calendar where they click timeslots to toggle availability (stored in `FacultyAvailability`). All inputs have save buttons that call the backend APIs.

- **Analytics Dashboard:** Only visible to Admins. Includes charts: a bar graph of **Classes per Department**, a pie chart of **Room Utilization** (occupied vs free time), and a heatmap showing **Conflicts by Day/Period**. Each chart has filters (select campus or term) and an export link. Below the charts are summary cards (e.g. “Total Teachers: 47”, “Percent of periods filled: 85%”). This helps admins identify bottlenecks (e.g. an overbooked lab) for future planning.

Reusable AI Prompt for Timetable Generation

```
You are an AI scheduling assistant. The user will provide:
- A list of classes to schedule (each with course name, student group, and
number of periods needed),
- A list of teachers (with their available time slots),
- A list of rooms (with capacities and types),
- Time slot definitions (days of week and periods per day),
- Specific constraints (e.g. “no classes on Wednesdays after 3pm”, “max 3
classes per teacher per day”, elective groups, break times).

Generate an optimal timetable that assigns each class to a day, period, room,
and teacher. The timetable must respect all constraints: no teacher or room is
double-booked, class size ≤ room capacity, and all institutional rules are
followed. Output the schedule as structured JSON, for example:

[
  {"class": "Math101-A", "day": "Monday", "period": 1, "room": "Room 201",
"teacher": "Alice"},
  {"class": "Physics102-B", "day": "Monday", "period": 2, "room": "Lab1",
"teacher": "Bob"},
  ...
]

Also include a summary of any potential conflicts or unfilled requirements.
Prioritize balancing teacher workloads and minimizing gaps in student schedules.
```

```
If using AI generation, clarify any assumptions made.

This prompt can be reused with varying inputs. It guides an AI/LLM to propose a
structured timetable or highlight conflicts. A system could fill in this prompt
from user data and use an LLM to suggest schedules, then verify the output
against actual constraints.
```

This comprehensive design leverages Supabase's integrated backend services (Postgres database, Auth, Storage) to streamline development [1] [2]. OCR is handled via image uploads to storage and processing by Tesseract or cloud APIs [3] [4]. All components are modular, enabling future extensions (AI optimization, push notifications, analytics). With thorough testing (unit, integration, load) and a robust DevOps setup, the platform will deliver conflict-free timetables across multiple institutions and campuses at scale.

Sources: Official documentation on Supabase (Postgres DB, Auth, Storage, Realtime) [1] [2]; Tesseract OCR overview [3]; Google Cloud Vision OCR recommendation [4]; standard practices in scheduling CSP and web architecture.

---

[1] [2] Supabase Docs
https://supabase.com/docs

[3] Tesseract (software) - Wikipedia
https://en.wikipedia.org/wiki/Tesseract_(software)

[4] OCR With Google AI | Google Cloud
https://cloud.google.com/use-cases/ocr