Nexus — Detailed Technical Documentation

Tech stack: Next.js (frontend + server APIs), PostgreSQL (Supabase), Drizzle ORM, Redis (caching & queueing)

Table of contents

---

1. Purpose & Scope

Purpose. This document describes the functional and technical design for **Nexus** — a unified CRM for TPOs and a Preparation Intelligence Hub for students. The goal is to provide a single source of truth for placement operations and structured student preparation data.

Scope. The scope of this document covers system requirements, architecture, data model, caching & queueing strategies, security controls, deployment guidance, and operational considerations for a production-capable system built with the chosen stack.

---

2. Product Overview

Nexus provides two main portals: - **TPO / Admin Portal (Operations Hub)** — company CRM, scheduling, communication engine, content governance, audit logs. - **Student Portal (Intelligence**

Hub) — placement calendar, company archive, verified experience blogs, personalized dashboards, wishlist and notifications.

Key value props: - Single master database for transactional integrity - Fast-read analytics & dashboards via caching - Smart communication to reduce email bounces and blacklisting - Verify & moderate student-contributed content

---

3. Stakeholders & Roles

- **Primary Admin (TPO Head)** — full control, system-level reporting, bulk actions.
- **Secondary Admin (Coordinators)** — restricted actions (call, update, schedule), cannot delete critical master data.
- **Students** — read-only access to intelligence features, follow companies, view verified content.
- **System Engineers / DevOps** — maintain infra, observability, backup and recovery.

Role-based access controls (RBAC) will be enforced at the API layer and supported by database row-level security where needed.

---

4. Functional Requirements (by module)

A. Admin / TPO Portal

- **FR-ADM-001:** CRUD company master records (name, domain, HR contacts, last visited, historical notes).
- **FR-ADM-002:** Kanban-style status board: To Call → In Discussion → Confirmed → Visited.
- **FR-ADM-003:** Record and view audit logs for calling & emailing activity.
- **FR-ADM-004:** Template manager for email/JD/Invite templates.
- **FR-ADM-005:** Timeline manager to set tentative/confirmed visit dates and sync to student calendar.
- **FR-ADM-006:** Conflict detection for date/venue collisions.
- **FR-ADM-007:** Content governance for student blogs: review, approve, reject.
- **FR-ADM-008:** Bulk email scheduler with domain-aware rate-limiting.

B. Student Portal

- **FR-STU-001:** Read-only live calendar of confirmed and tentative dates.
- **FR-STU-002:** Company intelligence archive with historical hiring stats and alumni link.
- **FR-STU-003:** Experience blog browsing with filters (company, role, branch, year).
- **FR-STU-004:** Personalized dashboard: notifications, wishlist, progress counters.
- **FR-STU-005:** Alumni connect list (limited contact details controlled by privacy settings).

C. Shared/Platform

- **FR-PLT-001:** Authentication & authorization (TPO staff vs students).
- **FR-PLT-002:** Notifications (email + in-app push) for schedule changes.
- **FR-PLT-003:** Export reports (CSV, XLSX) for drives and year-wise placements.

All functional requirements must be testable and traceable to acceptance criteria.

---

5. Non-functional Requirements

- **NFR-1 (Performance):** Dashboard page loads <= 2s for dataset sizes up to 100k rows (with proper caching).
- **NFR-2 (Scalability):** Support 5k concurrent students in peak windows; system should degrade gracefully by serving cached content.
- **NFR-3 (Availability):** 99.9% uptime (dependent on managed services SLAs).
- **NFR-4 (Security):** All sensitive data encrypted at rest & transit; RBAC implemented.
- **NFR-5 (Maintainability):** Migrations via Drizzle, tests, modular codebase.
- **NFR-6 (Privacy):** PII handling and opt-in consent for alumni contact details.

---

6. High-level Architecture

```
[User Browser]
   |  (HTTPS)
   v
[Vercel / Next.js App (Server/Edge)] -- SSR / RSC / API Routes
   |                          |
   |                          -> [Background workers (BullMQ) using Redis]
   |
   -> Supabase Postgres (Primary transactional DB)
   -> Redis (cache + queue + rate-limits)
   -> External Email Provider (Sendgrid/Mailgun)
```

Notes: - Next.js serves UI and server API endpoints (for server-side rendering, server components, and API actions). - Supabase provides Postgres storage + Auth; Drizzle used in server code for typed queries and migrations. - Redis used for caching, short-lived counts, locking, and queueing via BullMQ / Bee-Queue.

---

7. Component Details

7.1 Next.js (Frontend, API & Auth)

Pattern: App Router with server components for data fetching and Auth managed inside Next.js using **Auth.js (NextAuth)**.

Routing & API: - `app/` for pages & server components. - `app/api/*` (or `pages/api/*`) for server API endpoints used by background workers and client requests. - Use Server Components for pages that need fresh data (e.g., admin dashboard) and ISR / caching for student-facing pages.

Authentication & Sessions (Auth.js / NextAuth): - **Auth lives in Next.js** using Auth.js (NextAuth). Next.js therefore serves as: frontend, backend API, and the auth layer. - **Web (browser):** Use NextAuth cookie sessions (HTTP-only cookies) for browser sessions and UI flows. - **Mobile (Flutter):** Use NextAuth in **JWT mode** for API-first authentication. Expose explicit mobile endpoints for token exchange, refresh, and sign-in (see Auth Architecture section below). - **User table:** Persist user metadata in Supabase Postgres (users table). NextAuth can be configured to sync user records into the same Postgres instance

so both web and backend services share one source-of-truth for user profiles and roles. - **Roles & RBAC:** Store roles (student, tpo_admin, coordinator) in Postgres and enforce authorization in API routes and at the DB layer (via custom policies). NextAuth provides JWT claims which the backend can read to enforce RBAC.

Edge vs Serverless functions: - Keep long-running jobs or background tasks in worker processes; avoid long DB connections from edge functions.

7.2 Supabase (Postgres & Storage)

- Use **Supabase Postgres** as the primary source-of-truth (users, companies, drives, blogs, emails, analytics). Supabase is used here **only as a managed Postgres + storage provider**, not as the auth provider in this architecture.
- Use Supabase Storage for blog images and attachments. Use signed URLs for uploads.

Connection pooling: For serverless Next.js on Vercel, use a pooling solution and monitor connection usage. Monitor and tune connection limits for the managed Postgres instance.

Note: By centralizing Auth inside Next.js (NextAuth) you keep a single auth implementation (cookie sessions for web, JWT for mobile) while still using Supabase Postgres for persistent user/profile data. Supabase (Postgres, Auth, Storage) - Use Supabase Postgres as primary source-of-truth. - Use Supabase Auth for user management (sign-up flows for students, staff invitations for TPOs). - Storage for blog images or attachments. Use signed URLs for uploads.

Connection pooling: For serverless Next.js on Vercel, use Supabase connection pool options or a dedicated pooling solution. Monitor connection usage and increase plan if needed.

7.3 Drizzle ORM

- Use Drizzle for type-safe queries and migrations (drizzle-kit).
- Prefer raw SQL for complex aggregation queries (analytics) and use Drizzle for typed DTOs.

Sample Drizzle advantages: - Compile-time typed queries, smaller runtime overhead than some ORMs.

7.4 Redis

Roles: - **Caching:** counts, recent activity, dashboards. - **Queueing:** job queues for bulk email using BullMQ. - **Rate-limiting & Throttling:** per-domain/batch limits for email blasts. - **Distributed locking:** ensure only one worker processes a company's schedule at a time to avoid clashes.

Providers: - Upstash (serverless Redis) or Redis Cloud (Redis Labs). Choose based on latency to your app region.

---

8. Data Model (tables + fields)

Below are the primary tables. Use snake_case for columns and explicit types.

8.1 `users` (students + staff)

- `id` UUID PK
- `email` text unique
- `name` text
- `role` enum ('student','tpo_admin','coordinator')
- `auth_provider` varchar
- `created_at` timestamptz
- `profile_meta` jsonb (optional: branch, year, current_company)

8.2 `companies`

- `id` UUID PK
- `name` text
- `domain` text
- `industry` text
- `notes` text
- `created_at` timestamptz

8.3 `company_contacts`

- `id` UUID
- `company_id` UUID FK
- `name` text
- `email` text
- `phone` text
- `role` text

8.4 `drives` (a company visit / recruitment event)

- `id` UUID
- `company_id` UUID FK
- `status` enum('tentative','confirmed','visited','cancelled')
- `venue` text
- `start_date` date
- `end_date` date
- `notes` text
- `created_by` UUID (user)

8.5 `applications` (student -> drive registrations)

- `id` UUID
- `drive_id` UUID
- `student_id` UUID
- `status` enum('applied','shortlisted','interviewed','placed','rejected')
- `score` float

8.6 `blogs` (experience posts)

- `id` UUID
- `author_id` UUID

- `company_id` UUID
- `title` text
- `body` text
- `tags` text[]
- `is_verified` boolean
- `moderation_status` enum('pending','approved','rejected')
- `created_at` timestamptz

8.7 `audit_logs`

- `id` UUID
- `actor_id` UUID
- `action` text
- `target_type` text
- `target_id` UUID
- `meta` jsonb
- `created_at` timestamptz

8.8 `notifications`

- `id` UUID
- `user_id` UUID
- `type` text
- `payload` jsonb
- `read` boolean
- `created_at` timestamptz

---

9. Indexing, Partitioning & Scaling

- **Indexes:**
- `companies(name)` btree
- `drives(start_date)` btree
- `applications(drive_id, student_id)` composite index
- `blogs(company_id, created_at)` for archive queries
- GIN index on `blogs.tags` and `users.profile_meta` when querying JSONB
- **Partitioning:** For very large tables (audit_logs, applications), partition by date (monthly/quarterly).
- **Vacuum & Autovacuum tuning:** Monitor table bloat and tune autovacuum for high-churn tables.
- **Read replicas:** If reads dominate, configure read replicas and route analytics/long-running queries there.

---

10. Caching Strategy (Redis patterns)

What to cache: - Recent drives & calendars (student-facing) — keyed by `calendar:year:semester` - Company details & small archival stats — `company:slug` TTL 1h - Dashboard aggregates (counts, top companies) — TTL 5m; refresh on write via invalidation hooks - Rate-limit counters for email sending per domain — sliding window counters

Cache invalidation patterns: - Write-through for company updates (update DB then invalidate `company:*` keys) - Event-based invalidation: on drive create/update → invalidate `calendar:*` and user wishlists related to company

Locking: - Use Redis distributed locks (`SET key NX PX`) when scheduling/confirming drives to avoid double-book collisions.

---

11. Communication Engine

10A. Email Template Management (Database-driven)

Purpose. Store, version, render, and govern reusable email templates (single/bulk) with safe variables, approvals, and auditability.

10A.1 Why templates live in the DB (not code)

- TPO can edit content without redeploys
- Versioning & rollback
- Approval workflow (avoid accidental bad sends)
- Per-template rate limits & policies

10A.2 Template data model

```
email_templates (
  id uuid primary key,
  name text not null,                -- e.g. "Company Invitation"
  slug text unique not null,         -- company-invite
  subject text not null,
  body_html text not null,
  body_text text,                    -- optional fallback
  variables text[] not null,         -- ["company_name","drive_date"]
  status text check (status in ('draft','approved','archived')) default
'draft',
  created_by uuid references users(id),
  approved_by uuid references users(id),
  created_at timestamptz default now(),
  updated_at timestamptz default now()
)
```

Optional versioning table (recommended):

```
email_template_versions (
  id uuid primary key,
  template_id uuid references email_templates(id),
  subject text,
  body_html text,
  body_text text,
  variables text[],
  version int,
  created_by uuid,
  created_at timestamptz default now()
)
```

10A.3 Variable system (safe rendering)

- Templates use **named placeholders** only:

```
{{company_name}}
{{drive_date}}
{{college_name}}
```

- Variables must be declared in `email_templates.variables`.
- Rendering step validates:
- All required variables are provided
- No undeclared variables exist (prevents injection)

Rendering pipeline:

```
Fetch template → validate variables → render → enqueue send job
```

Use a logic-less renderer (e.g., Mustache / Handlebars without helpers).

10A.4 Approval & governance flow

- Draft → Approved → Archived lifecycle
- Only **approved** templates can be used for bulk sends
- Editing an approved template creates a **new version** and resets status to `draft`
- All sends store `template_id` + `template_version`

10A.5 Per-template sending policies

Add a JSON policy field (optional):

```
send_policy jsonb
```

Example:

```
{
  "max_per_domain_per_hour": 50,
  "requires_tpo_approval": true,
  "allow_bulk": true
}
```

Workers enforce these policies before dispatch.

10A.6 Audit & traceability

Each outbound email stores: - `template_id` - `template_version` - rendered variables (JSON)

This enables: - Full replay/debug of what was sent - Compliance and post-incident audits

10A.7 Caching strategy

- Cache **approved templates** in Redis:

```
email:template:{slug} → JSON
```

- TTL: long (6–12 hours)
- Invalidate cache on template approval or archive

10A.8 UI expectations

- Rich text editor (HTML) + plain text preview
- Variable sidebar with auto-insert
- Validation warnings for missing variables
- Approval button for TPO head
- Preview send (send to self)

---

11. Communication Engine

Overview. The communication engine handles outbound (single + bulk) emails, inbound mail ingestion, and classification/segregation of incoming mail into company-specific timelines or miscellaneous buckets. It combines a transactional email provider, a Redis-backed queueing system, webhook-based inbound processing, and a domain-intelligence classifier.

11.1 Components

- **Template Manager** (DB + UI) — store email templates, variables, and per-template sending policies.
- **Send Queue (Redis + BullMQ)** — reliable, retryable, rate-limited queue for all outbound email jobs.
- **Worker Processes** — dequeue jobs, enforce per-domain and global rate limits, call the email provider (SendGrid/Mailgun/SES), and persist send logs.

- **Inbound Webhook Processor** — HTTP endpoint(s) that receive parsed incoming email payloads from the provider (or Gmail push events), normalize the payload, and enqueue classification jobs.
- **Classifier & Processor** — attaches inbound emails to companies, threads, or the miscellaneous inbox using domain mappings, headers, and thread metadata. Also handles suppression lists and bounce processing.
- **Email Store** — persistent table that stores the normalized inbound/outbound messages, headers, threading metadata, and classification.
- **Admin Inbox UI** — unassigned/suggested mappings view for human verification and quick reclassification.

11.2 Outbound Flow (Single & Bulk)

Flow:

```
Admin UI -> Next.js API -> Redis/BullMQ enqueue -> Worker -> Email Provider -
> Recipient
```

Worker responsibilities: - Render template with variables and track `message_id`. - Add metadata headers such as `X-Nexus-Company-ID`, `X-Nexus-Drive-ID`, and `Reply-To` that map to a company-specific mailbox (if applicable). - Apply domain-aware rate limiting: Redis counters per recipient domain (`email:domain:{domain}:count`) with sliding-window semantics. - Respect suppression lists and do not send to bounced addresses. - Persist an outbound log row in `emails` table with `direction='outbound'` and recorded headers.

Headers to include on outbound messages (example):

```
X-Nexus-Company-ID: <company_id>
X-Nexus-Drive-ID: <drive_id>
X-Nexus-Sent-By: <user_id>
Reply-To: hr+<company_slug>@nexus.college.edu
```

Using a `reply-to` alias like `hr+oracle@nexus.college.edu` helps attribute inbound mails to the correct company and drive.

11.3 Inbound Flow (Webhooks preferred)

Preferred: Configure the transactional provider's inbound parsing/webhook (Mailgun routes, SendGrid Inbound Parse, SES+SNS+Lambda) to POST normalized JSON to `/api/v1/email/inbound`.

Inbound webhook flow:

```
Email Provider -> POST /api/v1/email/inbound -> normalize -> dedupe ->
enqueue classification job -> store raw payload + minimal normalized row
```

Normalization step extracts: `from`, `to`, `cc`, `bcc`, `subject`, `text_body`, `html_body`, `attachments` (metadata), `message_id`, `in_reply_to`, `references`, and full headers.

Deduplication: Use `message_id` and a Redis dedupe key `email:inbound:dedupe:{message_id}` with short TTL to prevent double-processing.

Alternative: If provider webhooks are unavailable, implement Gmail API watch + push notifications and pull new messages. This is more complex; prefer webhooks.

11.4 Classification & Domain Intelligence

Priority-based classification algorithm: 1. **Explicit metadata** — look for `X-Nexus-Company-ID`, `Reply-To` aliases that match `hr+{company_slug}@nexus.college.edu`, or a special `To` address bound to a company mailbox. 2. **Thread matching** — match `in_reply_to` or `references` to existing `emails` threads in DB; inherit the `company_id` and `drive_id`. 3. **Domain mapping lookup** — extract the sender domain (normalize subdomains) and lookup `company_domains`. 4. **Heuristics & keywords** — subject/body matching for company name, signatures, or job descriptions. 5. **Fallback** — classify as `miscellaneous` and surface in Admin Inbox for manual tagging.

SQL table: `company_domains` (added to schema):

```
company_domains (
  id uuid primary key,
  company_id uuid references companies(id) not null,
  domain text not null,
  confidence text check (confidence in ('manual','verified','learned'))
default 'learned',
  created_by uuid,
  created_at timestamptz default now()
)
```

Blocked domains: Add a `blocked_domains` table to prevent auto-mapping for public email providers:

```
blocked_domains (
  domain text primary key
)
```

Seeded with `gmail.com`, `outlook.com`, `yahoo.com`, etc.

Auto-learning & verification: - When an inbound email is classified via thread or explicit metadata, extract the sender domain and insert a `company_domains` row with `confidence='learned'` if not present. - Maintain counters for learned domains. When a learned domain reaches N successful matches (configurable, e.g., 5), auto-promote it to `verified` or flag it for TPO review. - Manual confirmation UI allows TPO to mark domains as `manual` and remove incorrect learned domains.

Normalization rules: - Normalize `careers.oracle.com` -> `oracle.com` by taking the registered domain (use public suffix list library). - Lowercase domains and trim sub-subdomains.

11.5 Email Storage Schema (additions)

Add or extend the `emails` table defined earlier with fields for threading and classification:

```
emails (
  id uuid primary key,
  direction text check (direction in ('inbound','outbound')),
  message_id text unique,
  in_reply_to text,
  references text[],
  thread_id text,
  company_id uuid null references companies(id),
  drive_id uuid null references drives(id),
  from_email text,
  to_emails text[],
  cc_emails text[],
  subject text,
  text_body text,
  html_body text,
  headers jsonb,
  classification jsonb, -- stores details about why it was classified
  created_at timestamptz default now()
)
```

11.6 Redis keys & rate-limiting

Example keys: - `email:send:queue` (BullMQ queue) - `email:domain:{domain}:count` (rolling counter for rate-limit) - `email:inbound:dedupe:{message_id}` (dedupe lock) - `domain:cache:{domain}` → cached `company_id` with TTL

Rate limiting strategy: - Per-domain sliding window counters (e.g., 100 emails / 10 minutes for specific domains), enforced in worker before dispatch. - Global concurrency controls (max workers sending simultaneously).

11.7 Admin Inbox & Human-in-the-loop

- **Unassigned / Suggested** view lists incoming mails where `company_id` is null or confidence < `verified`.
- UI shows system-suggested company (from domain lookup), thread history, and a one-click confirm/deny action.
- Confirming a suggestion upgrades `company_domains.confidence` to `manual` and backfills previous messages classified as `learned` for that domain.

11.8 Bounce & Suppression handling

- Persist bounce events from provider webhooks and add bounced addresses to a suppression list.
- Workers check suppression lists before sending.
- Provide UI for the TPO to review and reinstate addresses after manual verification.

11.9 Webhook handler (pseudo-code)

- Normalize payload
- Deduplicate using `message_id`
- Insert normalized row into `emails` with `direction='inbound'`
- Enqueue `classification_job` with email id
- Respond 200 OK

11.10 Security & Deliverability

- Ensure SPF, DKIM, DMARC records are configured for sending domain(s).
- Use provider-managed domains or verified sending domains.
- Monitor bounce and complaint rates; alert TPO when thresholds hit.

12. Security & Compliance

- **Authentication:** Auth is implemented by **Auth.js (NextAuth)** running inside the Next.js backend. For web, NextAuth uses cookie-based sessions; for mobile, NextAuth is configured in JWT mode and the backend exposes mobile-friendly token endpoints. User identity and roles are persisted in Supabase Postgres. MFA, invitation flows, and admin-only actions are enforced by Next.js API middleware and server-side checks.
- **Authorization:** RBAC enforced at the API layer; critical operations also checked at DB level with row-level security policies that reference user IDs and roles stored in Postgres.
- **Transport & Storage:** TLS everywhere; database encryption at rest (managed by Supabase).
- **Secrets:** Store in Vercel / platform environment variables; rotate periodically.
- **Input sanitization:** Parameterized queries (Drizzle) to prevent SQL injection; XSS protection in UI via sanitization libraries for blog content.
- **Audit trail:** All admin actions logged in `audit_logs`.
- **Privacy:** PII minimised; alumni contact details require consent; retention policy for personal data.

---

13. Observability & Monitoring

- **Application logs:** structured logs (JSON) collected via a provider (Logflare, Datadog).
- **Error tracking:** Sentry for exceptions & performance traces.
- **DB metrics:** enable `pg_stat_statements`, monitor slow queries, connection counts.
- **Redis metrics:** memory usage & eviction rates.
- **Alerts:** set thresholds for error rate, DB connection saturation, job queue backlog.

13A. AI Assist Layer (Phase 1: Email & Blog Generation)

Scope (Phase 1). In the initial phase, AI usage in Nexus is deliberately limited to **assistive text generation** for: - Email drafting (single & bulk, human-approved) - Blog generation and structuring (student interview experiences)

AI is **not** used for authoritative decisions, automated actions, or source-of-truth data updates.

13A.1 Design principles

- **Assistive-only:** AI outputs require human review before use.
- **Deterministic boundaries:** Strict prompts, limited context, capped tokens.
- **Self-hosted & cost-aware:** Small text models hosted on an Oracle Free Tier VM.
- **Auditable:** Prompts, outputs, and approvals are stored.
- **Fail-safe:** If AI service is unavailable, core workflows continue without degradation.

13A.2 Architecture

```
Next.js (Admin / Student UI)
        |
        | Internal API call
        v
AI Assist Service (Oracle Free Tier VM)
        |
        | llama.cpp / lightweight inference server
        v
Small Text Model (quantized)
```

- AI service runs as a **separate internal microservice**.
- Never exposed directly to public clients.
- Access restricted by IP allowlists and internal API keys.

13A.3 Supported AI features (Phase 1)

A. Email Draft Generation

Use cases: - First outreach to company HR - Polite follow-up after no response - Reminder emails - Thank-you emails post visit

Inputs: - Template type (invitation / follow-up / reminder) - Company name - Optional context (previous email summary) - Tone (formal / concise / polite)

Output: - Draft email body (subject + content)

Constraints: - No auto-send - Draft saved with status = `pending_approval` - TPO must explicitly approve or edit before enqueueing

---

B. Blog Generation & Structuring

Use cases: - Convert raw student-submitted notes into structured interview experiences - Improve clarity, grammar, and sectioning

AI responsibilities: - Structure content into sections (OA, Technical, HR, Tips) - Fix grammar and flow - Remove redundancy

Explicitly disallowed: - Adding new factual content - Inventing questions or answers

Generated content is tagged as **AI-assisted** and goes through admin approval.

13A.4 Prompt & rendering strategy

- Prompts are stored as **versioned prompt templates** in code or DB.
- Inputs are sanitized and length-limited before being sent to the model.
- Use logic-less generation (no tool calls, no browsing).

Example (email prompt skeleton):

```
You are an assistant helping draft a professional placement email.
Do not invent facts.
Tone: {{tone}}
Company: {{company_name}}
Context: {{context}}
Generate a concise email draft.
```

13A.5 Data storage & audit

Add a table to persist AI interactions:

```
ai_generations (
  id uuid primary key,
  type text check (type in ('email','blog')),
  input jsonb,
  output text,
  model text,
  approved boolean default false,
  approved_by uuid,
  created_at timestamptz default now()
)
```

Each generated artifact links back to: - Email template or blog draft - User who requested generation - Approval action

---

13A.6 Performance & limits

- Max tokens per request (configurable, e.g., 512–1024)
- Timeout: 2–5 seconds per request
- Concurrency limits enforced at AI service layer

13A.7 Model choices (Oracle Free Tier)

Recommended quantized models: - Phi-2 / Phi-3-mini — fast, summarization & drafting - Mistral 7B (Q4/Q5) — higher-quality drafting if memory allows

Inference stack: - llama.cpp (HTTP server mode) - No GPU required

---

13A.8 Failure handling

- If AI service fails or times out:
- UI falls back to manual editor
- Clear messaging: "AI assist unavailable"
- No critical workflow depends on AI availability

13A.9 Compliance & transparency

- All AI-generated content is visibly marked as **AI-assisted**
- Users retain full control to edit or discard suggestions
- Logs retained for auditing and debugging

14A. Mobile App (Flutter) & Responsive Web Portal

Goal. Deliver a high-quality mobile experience using **Flutter** while ensuring the web portal is fully **responsive** and mobile-first. Provide a carefully chosen feature subset on mobile to maximize value and performance for students, while keeping admin-heavy operations on the web.

14A.1 Overall approach

- **Mobile-first design:** All UX decisions start with the smallest common screen and scale up. Use consistent design tokens across web and mobile (colors, spacing, typography).
- **Platform roles:**
- **Web (Next.js)** — Full feature set (TPO admin consoles, heavy reporting, templates, bulk email management, inbox moderation, content governance, audit logs).
- **Mobile (Flutter)** — Student-centric features + light coordinator workflows (quick updates, call logging).
- **Responsive web:** Next.js UI must be responsive and provide parity for essential student flows (calendar, company intelligence, blogs, wishlist) and admin quick actions (approve blog, view notifications) with clear progressive enhancement.
- **Shared design system:** Maintain a shared component library/spec (Figma + tokens) so mobile and web feel consistent.

14A.2 Mobile app — feature subset (recommended)

Primary audience: Students (core) and Student Coordinators (limited).

Must-have features (Phase 1 Mobile): 1. **Authentication** — Supabase Auth (magic link / email) or OAuth; tokens stored securely in platform storage (secure storage / keychain). 2. **Personalized Dashboard** — Top notifications, upcoming drives countdown, status summary. 3. **Placement Calendar** — Interactive read-only calendar and timeline; ability to add to device calendar (optional). 4. **Company Intelligence & Search** — View company profiles, historic stats, and alumni connect lists. 5. **Experience Blogs** — Read approved blogs, search, filter, bookmark for offline reading. 6. **My Wishlist / Follow** — Follow companies and receive push notifications for updates. 7. **Notifications** — Push (FCM) + in-app; urgent alerts for schedule changes. 8. **Inbox (Student)** — View communications relevant to the student (optional: read-only copies of TPO emails sent to all students). 9. **Profile** — Basic profile editing (branch, year, contact preferences) and privacy toggles for alumni contact sharing.

Coordinator-specific lightweight features (mobile-friendly): - Quick call logging (mark call outcome) - Mark drive attendance / confirmations (quick taps)

Not-on-mobile (keep web-only initially): - Bulk email composer & scheduling - Template manager (full editing & approval flows) - Full audit log and complex reporting - Inbox moderation & advanced classification tools (admin inbox)

Rationale: Mobile should focus on consumption, notifications, and light tasking — actions that are quick, low-risk, and frequently used by students.

14A.3 Mobile architecture & sync strategy

Architecture:

```
Flutter App (iOS/Android)
  ├─ Local cache (SQLite / Hive)
  ├─ Sync manager (background & foreground sync)
  └─ APIs (Next.js / RESTful endpoints)
```

Local storage choices: - **SQLite (sqflite)** — Relational local cache for calendar, wishlist, and saved blogs. - **Hive / sembast** — Fast key-value for small preferences and token storage.

Sync patterns: - **Delta sync endpoints**: provide `GET /api/v1/mobile/sync?since=<timestamp>` to fetch changes since last sync. - **Pull-to-refresh + background fetch:** keep mobile data fresh; background sync frequency configurable. - **Optimistic UI updates** for quick coordinator actions (local change → enqueue to send → reconcile with server). - **Conflict resolution:** Last-write-wins for low-risk fields, and server wins for authoritative records (e.g., drive schedule).

Connectivity & offline: - Offline read for cached calendar & saved blogs. - Queue local actions and retry when online (ex: call logs, wishlist toggles). - Graceful messaging when server calls fail.

Security on mobile: - Store tokens in secure storage (Keychain / Android Keystore). - Use short-lived JWTs with refresh tokens (server-side refresh endpoint). - Enforce TLS pinning optionally for extra security.

14A.4 Push notifications & background actions

- **FCM (Firebase Cloud Messaging)** for push notifications (already part of project context). Use a server-side component to send topic-based or user-targeted notifications.
- **Notification types:** urgent schedule changes, company confirms, blog approvals (if student authored), and admin announcements.
- **Deep links:** Notifications should deep-link into relevant screens (company, drive, blog).
- **Actionable notifications:** e.g., accept calendar add, view details.

14A.5 Responsive Web Requirements

Web portal must be responsive and mobile-friendly: - Use a mobile-first CSS approach and Tailwind tokens (or chosen design system). - Breakpoints: mobile (≤640px), tablet (641–1024px), desktop (>1024px). - Ensure touch-friendly UI on small screens: 44–48px tappable targets, avoid hover-only interactions. - Server-side rendering (Next.js) for SEO pages (company archives, blogs) and fast first paint. - Use RSC/ISR judiciously: cache student-facing read-only pages with stale-while-revalidate patterns. - Implement progressive enhancement: features that require JS should degrade gracefully.

Performance goals for responsive web: - First Contentful Paint (FCP) < 1.5s on 3G Lighthouse simulated mobile - Time to interactive (TTI) < 3s on typical mobile devices - Keep bundle sizes small: split vendor code and lazy-load admin-heavy modules

---

14A.6 Design & UX considerations

- **Shared design tokens**: colors, spacing, fonts, iconography in a central design system
- **Readable typography** for small screens
- **Skeleton loaders** and placeholders while fetching content
- **Accessible components** (aria labels, contrast, keyboard navigation on web)
- **Consistent navigation**: bottom navigation on mobile for main sections (Dashboard, Calendar, Companies, Blogs, Profile)

14A.7 Testing & Release

- **Flutter:** Use unit tests, widget tests, and integration tests (Flutter Driver / integration_test). Setup CI to build & run tests for both iOS and Android emulators.
- **Responsiveness tests:** Use Storybook / Chromatic or Percy snapshots for key pages and device sizes.
- **Beta distribution:** Use TestFlight (iOS) and Google Play Internal testing for early feedback.

14A.8 App store & privacy considerations

- Prepare privacy policy mentioning data sync, notifications, and PII handling.
- Ask for explicit consent for push notifications and optionally for sharing alumni contact details.
- Prepare app store metadata and screenshots that emphasize the student value props.

14A.9 API surface for mobile (subset)

Endpoints optimized for mobile usage and sync: - `GET /api/v1/mobile/sync?since=` — delta sync for calendar, drives, wishlist, notifications - `POST /api/v1/mobile/call-log` — quick coordinator call logs - `GET /api/v1/companies/:id` — company detail (cached) - `GET /api/v1/drives?from=&to=` — calendar feed - `POST /api/v1/wishlist/toggle` — follow/unfollow company - `POST /api/v1/ai/generate` — request AI-assisted email/blog draft (internal auth) - `GET /api/v1/blogs?company=&page=` — paginated blog archive

---

14. CI / CD and Deployment

- **Next.js:** Deploy to Vercel (preview branches auto-deploy); environment variables set per environment.
- **Migrations:** Use `drizzle-kit` migrations run in CI during deploy; apply migrations in a migration job.
- **Workers:** Deploy worker processes on a small VM or container (Fly / Render / Railway) with autoscaling for queue backlog.
- **Redis:** Use Upstash for serverless or Redis Cloud for managed low-latency Redis.
- **Secrets management:** Vercel secrets or an external vault for email provider keys, DB URLs.

---

15. Testing Strategy

- **Unit tests:** for business logic, Drizzle query builders, and utility functions.
- **Integration tests:** API endpoints integrated with a test Postgres instance (use ephemeral databases or DB branching in Supabase/Neon).
- **E2E tests:** Playwright or Cypress for key user flows (login, schedule drive, submit blog).
- **Load testing:** k6 or Artillery for simulating peak loads (bulk email sends, concurrent students viewing dashboards).

---

16. Backup, Recovery & Migration Plan

- **Backups:** Use Supabase-managed automated backups + periodic dumps to object storage.
- **PITR:** Enable point-in-time recovery for critical data if offered in plan.
- **Migration off Supabase (if needed):**
- Export logical dumps (pg_dump) + storage assets.
- Provision new Postgres (Aiven / RDS / Neon) and import.
- Update connection strings and run smoke tests.
- For analytics, consider moving historical event logs to a columnar store (ClickHouse) before migrating.

---

17. API Reference (key endpoints)

All endpoints are under `/api/v1` and authenticated using Supabase session tokens.

Auth

- `POST /api/v1/auth/signin` — Supabase sign-in (delegated)
- `POST /api/v1/auth/signup` — Supabase sign-up

Companies

- `GET /api/v1/companies` — paginated list (supports `?search=&page=&limit=&status=`)
- `GET /api/v1/companies/:id` — details (cacheable)
- `POST /api/v1/companies` — create (TPO only)
- `PUT /api/v1/companies/:id` — update

Drives

- `GET /api/v1/drives?from=&to=&status=` — calendar feed
- `POST /api/v1/drives` — create drive (conflict detection + lock)
- `PUT /api/v1/drives/:id` — update (invalidate calendar caches)

Blogs

- `GET /api/v1/blogs?company=&tag=&page=` — archive
- `POST /api/v1/blogs` — submit blog (moderation queue)
- `POST /api/v1/blogs/:id/approve` — approve (moderator)

Notifications

- `GET /api/v1/notifications` — user notifications
- `POST /api/v1/notifications/mark-read` — mark read

---

18. Appendix: Examples

18.1 Drizzle model — companies (TypeScript)

```
import { pgTable, uuid, text, timestamptz } from 'drizzle-orm/pg-core'

export const companies = pgTable('companies', {
  id: uuid('id').primaryKey().defaultRandom(),
  name: text('name').notNull(),
  domain: text('domain'),
  industry: text('industry'),
  notes: text('notes'),
  created_at: timestamptz('created_at').defaultNow(),
})
```

18.2 Example Drizzle query (server-side)

```
const company = await db.select().from(companies).where(eq(companies.id,
someId)).limit(1)
```

18.3 Next.js server action example (invalidate cache)

```
'use server'
import { redisClient } from '@/lib/redis'
import { db } from '@/lib/db'

export async function createDrive(data) {
  const res = await db.insert(drives).values(data)
  // invalidate cache
  await redisClient.del(`calendar:${data.semester}`)
  return res
}
```

18.4 Redis key examples

- `company:{company_id}` — cached company JSON
- `calendar:{year}:{semester}` — cached calendar payload
- `email:domain:{domain}:quota` — integer counter for rate-limits

Final notes & next steps

- This doc is a working blueprint. Next practical steps:
- Convert FRs into Jira/Trello tickets with acceptance criteria.
- Create initial DB migration & seed sample data.
- Build skeleton Next.js app with Supabase auth and a companies CRUD.
- Add Redis worker for a simple email send flow and test rate-limiting.

If you'd like, I can now: - Produce ER diagrams from the data model, or - Generate ready-to-run `drizzle-kit` migration files for the schema above, or - Scaffold a minimal Next.js repo skeleton with Supabase & Redis config.

Tell me which next step you want and I'll generate it directly in this doc.