# Goal

Build a full‑stack, production‑ready **Alumni Network Portal** from scratch to deployment. Deliver a clean, secure, and fully functional implementation of all features listed below, with tests, CI, and deployment config.

## High‑Level Requirements

Implement these modules end‑to‑end (DB ➜ API ➜ UI), matching the behaviors described.

1. **Auth & Accounts**

* Email+password auth with password reset via magic link.
* Roles: alumni (default), admin.
* Profile creation on first login (graduation year, degree, company, skills, location).
* Admin-only verification toggle (is\_verified), with **upload of proof of graduation (PDF)**.
* Display a “verified” checkmark on profiles when is\_verified=true.

1. **Alumni Directory**

* Search by name, grad year, company, degree; filter by location and skills.
* Paginated results with server-side filtering and sorting.

1. **Private Messaging**

* Alumni↔Alumni direct messages with conversation view, inbox/sent folders.
* Email notifications for new messages (user can opt in/out).

1. **Job Board**

* Alumni can post jobs.
* Filters: Job type (Remote/On-site), Industry.
* Apply via external link (LinkedIn/email) stored for each posting.

1. **Events**

* University-hosted event calendar (list + detail + RSVP).

1. **Discussion Forums**

* Topic-based forums (e.g., Career Advice).
* Threads, posts, up/down votes, and mark best response (per thread owner or admin).

1. **Mentorship**

* Mentor/mentee matching (mentors expose skills & availability slots).
* Booking flow: request ➜ accept ➜ scheduled session ➜ feedback form.

1. **Newsletter**

* Admin UI for bulk email campaigns.
* Per-user subscription preferences.
* Open-rate tracking with a tracking pixel.

1. **Analytics Dashboard (Admin)**

* “Top Companies” among alumni, degree distribution charts.
* Export CSV and PDF for selected datasets.

1. **Donations**

* Donation progress bar (goal configurable).
* Support recurring donations; generate tax receipts (PDF) and email to donor.

## Tech Stack & Architectural Decisions

* **Framework**: Next.js 14 (App Router) + TypeScript.
* **UI**: Tailwind CSS + shadcn/ui components.
* **Forms & Validation**: React Hook Form + Zod.
* **State/Data**: Next.js Server Actions + React Query for client queries where useful.
* **Database**: Postgres (Neon). Access via **Prisma ORM**.
* **Auth**: Auth.js (NextAuth) credentials provider (email+password) + email sign-in for resets.
* **Storage**: Object storage (S3-compatible) for PDF uploads and generated receipts.
* **Email**: SMTP provider (e.g., Resend/SendGrid). Implement via nodemailer abstraction.
* **PDF**: pdfkit or puppeteer for PDF receipts/exports.
* **Charts**: recharts.
* **Payments (Donations)**: Stripe.
* **Background jobs**: Lightweight queue using **BullMQ** + Redis (for newsletters, email sends, tracking aggregation). If Redis not available, fall back to serverless cron endpoints (but prefer Redis in docker-compose for local dev).
* **Testing**: Jest + Testing Library (unit), Playwright (e2e), Prisma test DB.
* **CI**: GitHub Actions (lint, typecheck, test, prisma migrate deploy, build).
* **Deployment**: Vercel (web) + Neon (DB) + S3-compatible storage + Stripe + managed Redis.

Security & DX: - Use .env with strong typing via zod schema guard on process.env at boot. - CSRF/Session security via Auth.js defaults; rate-limit login/message send endpoints (e.g., @upstash/ratelimit). - Input validation on all server actions and API routes; strict TypeScript. - Audit trail: store admin verification actions and role changes.

## Repository Layout

/ (root)  
├─ app/ # Next.js app router  
│ ├─ (marketing)/  
│ ├─ (dashboard)/  
│ ├─ api/ # Route handlers (REST-ish) where needed  
│ └─ auth/ # sign-in/up/reset flows  
├─ components/  
├─ lib/  
│ ├─ auth/  
│ ├─ db/  
│ ├─ email/  
│ ├─ storage/  
│ ├─ payments/  
│ └─ analytics/  
├─ prisma/  
│ ├─ schema.prisma  
│ └─ migrations/  
├─ public/  
├─ scripts/  
├─ tests/  
│ ├─ e2e/  
│ └─ unit/  
├─ .github/workflows/  
├─ docker-compose.yml (dev: web+db+redis+mailhog)  
└─ README.md

## Environment Variables

Create .env (never commit):

# Database  
DATABASE\_URL="postgresql://<user>:<pass>@<neon-host>/<db>?sslmode=require&channel\_binding=require"  
DIRECT\_URL="postgresql://<user>:<pass>@<neon-host>/<db>?sslmode=require&channel\_binding=require"  
  
# Auth  
NEXTAUTH\_URL="https://<your-domain>"  
NEXTAUTH\_SECRET="<generated-secret>"  
  
# Email  
SMTP\_HOST="<smtp-host>"  
SMTP\_PORT="587"  
SMTP\_USER="<user>"  
SMTP\_PASS="<pass>"  
  
# Storage (S3-compatible)  
S3\_ENDPOINT="https://s3.amazonaws.com"  
S3\_REGION="us-east-1"  
S3\_BUCKET="<bucket>"  
S3\_ACCESS\_KEY\_ID="<key>"  
S3\_SECRET\_ACCESS\_KEY="<secret>"  
  
# Stripe  
STRIPE\_SECRET\_KEY="sk\_live\_..."  
STRIPE\_WEBHOOK\_SECRET="whsec\_..."  
  
# Redis (BullMQ)  
REDIS\_URL="redis://<user>:<pass>@<host>:6379"

**Important**: Secrets must never be exposed to client-side code. Access only in server components, server actions, API routes, or backend libs.

## Database Design (Prisma Schema Outline)

*(Representative subset; implement with proper indices & FKs.)*

model User {  
 id String @id @default(cuid())  
 email String @unique  
 passwordHash String  
 role Role @default(ALUMNI)  
 profile Profile?  
 createdAt DateTime @default(now())  
 updatedAt DateTime @updatedAt  
}  
  
enum Role { ADMIN ALUMNI }  
  
model Profile {  
 id String @id @default(cuid())  
 userId String @unique  
 user User @relation(fields: [userId], references: [id])  
 name String  
 graduationYear Int  
 degree String  
 company String?  
 skills String[]  
 location String?  
 isVerified Boolean @default(false)  
 verification Verification?  
}  
  
model Verification {  
 id String @id @default(cuid())  
 profileId String @unique  
 profile Profile @relation(fields: [profileId], references: [id])  
 documentKey String // S3 object key for proof PDF  
 verifiedById String? // admin who verified  
 verifiedAt DateTime?  
}  
  
model MessageThread {  
 id String @id @default(cuid())  
 userAId String  
 userBId String  
 messages Message[]  
 createdAt DateTime @default(now())  
}  
  
model Message {  
 id String @id @default(cuid())  
 threadId String  
 thread MessageThread @relation(fields: [threadId], references: [id])  
 senderId String  
 body String  
 createdAt DateTime @default(now())  
 readAt DateTime?  
}  
  
model Job {  
 id String @id @default(cuid())  
 authorId String  
 title String  
 company String  
 industry String  
 jobType JobType  
 location String?  
 applyUrl String?  
 applyEmail String?  
 createdAt DateTime @default(now())  
}  
  
enum JobType { REMOTE ONSITE HYBRID }  
  
model Event {  
 id String @id @default(cuid())  
 title String  
 startsAt DateTime  
 endsAt DateTime  
 location String  
 details String?  
 rsvps RSVP[]  
}  
  
model RSVP {  
 id String @id @default(cuid())  
 eventId String  
 userId String  
}  
  
model ForumCategory { id String @id @default(cuid()) name String slug String @unique }  
model Thread { id String @id @default(cuid()) categoryId String title String authorId String createdAt DateTime @default(now()) bestPostId String? votes Vote[] posts Post[] }  
model Post { id String @id @default(cuid()) threadId String authorId String body String createdAt DateTime @default(now()) votes Vote[] }  
model Vote { id String @id @default(cuid()) userId String postId String value Int }  
  
model MentorProfile { id String @id @default(cuid()) userId String @unique skills String[] bio String? }  
model AvailabilitySlot { id String @id @default(cuid()) mentorId String start DateTime end DateTime }  
model MentorshipPair { id String @id @default(cuid()) mentorId String menteeId String createdAt DateTime @default(now()) }  
model Session { id String @id @default(cuid()) pairId String start DateTime end DateTime feedback Feedback? }  
model Feedback { id String @id @default(cuid()) sessionId String rating Int comments String? }  
  
model NewsletterPref { id String @id @default(cuid()) userId String @unique subscribed Boolean @default(true) }  
model Campaign { id String @id @default(cuid()) subject String bodyHtml String createdAt DateTime @default(now()) sends Send[] }  
model Send { id String @id @default(cuid()) campaignId String userId String sentAt DateTime @default(now()) openEvents OpenEvent[] }  
model OpenEvent { id String @id @default(cuid()) sendId String openedAt DateTime @default(now()) ip String? ua String? }  
  
model Donation { id String @id @default(cuid()) userId String? amountCents Int currency String @default("USD") recurring Boolean @default(false) stripePaymentIntentId String? createdAt DateTime @default(now()) }  
model Receipt { id String @id @default(cuid()) donationId String pdfKey String issuedAt DateTime @default(now()) }

Indexing & constraints: - Add composite indexes for directory search (e.g., name, graduationYear, company, degree, skills via GIN text[]). - Unique constraints where appropriate (e.g., single bestPostId per thread validated in logic).

## API & Server Actions (Representative)

* **Auth**: POST /api/auth/register, POST /api/auth/login, POST /api/auth/reset, POST /api/auth/reset/confirm.
* **Profiles**: GET /api/profiles (filters), GET /api/profiles/:id, PATCH /api/profiles/:id (owner), POST /api/profiles/:id/verify (admin; sets isVerified).
* **Upload**: POST /api/upload/proof → returns S3 key; only owner & admin read.
* **Messaging**: GET /api/threads, POST /api/threads/:userId, GET /api/threads/:id, POST /api/threads/:id/messages.
* **Jobs**: GET /api/jobs, POST /api/jobs, GET /api/jobs/:id.
* **Events**: GET /api/events, POST /api/events (admin), POST /api/events/:id/rsvp.
* **Forum**: GET /api/categories, GET /api/threads?category=, POST /api/threads, POST /api/posts, POST /api/posts/:id/vote, POST /api/threads/:id/best.
* **Mentorship**: GET /api/mentors, POST /api/mentors/availability, POST /api/mentorship/request, POST /api/sessions/:id/feedback.
* **Newsletter**: GET /api/newsletter/prefs, POST /api/newsletter/prefs, POST /api/campaigns (admin), GET /api/open.gif?sendId=... (tracking pixel).
* **Donations**: POST /api/donations/checkout, POST /api/donations/webhook (Stripe), GET /api/donations/progress, POST /api/donations/:id/receipt.
* **Analytics**: GET /api/admin/analytics/top-companies, GET /api/admin/analytics/degree-distribution, exports.

For most operations prefer **Server Actions** from components; keep route handlers for webhooks/third-party callbacks or when REST is clearer.

## UI Pages (App Router)

* / marketing/landing.
* /auth/signin, /auth/signup, /auth/reset.
* /dashboard (alumni home): profile card, quick links.
* /directory with filters & search.
* /messages (threads list + chat view).
* /jobs (list, filters, create form).
* /events (calendar/list, details, RSVP).
* /forums (categories, thread list, thread detail with voting & best answer).
* /mentorship (find mentors, manage slots, requests, sessions, feedback).
* /newsletter (prefs; admin: campaigns & sends).
* /donate (checkout + progress bar).
* /admin (verification queue, users, analytics dashboard with charts & exports).

## Email Flows

* **Verify email** (on signup if desired) & **reset password** magic link.
* **New message** notification (debounced; digest option).
* **Campaign sends** with per-user unsubscribe link (/newsletter?unsubscribeToken=...).
* **Donation receipt** with attached PDF.

## Tracking Pixel

* GET /api/open.gif?send=<sendId> returns a 1×1 GIF. Log OpenEvent with IP/UA and timestamp.

## Donations

* Stripe Checkout for one-time & recurring.
* On successful payment webhook: create Donation, issue Receipt, generate PDF, store in S3, email receipt.

## Admin Verification Workflow

1. Alumni uploads proof (PDF).
2. Admin reviews in /admin with preview from S3.
3. Admin toggles isVerified, optionally sets verifiedById/verifiedAt.
4. User gets confirmation email; profile now shows blue checkmark.

## Data Exports

* CSV export endpoints for directory & donations.
* PDF exports (analytics snapshot & receipts) using headless Chromium.

## Implementation Plan (Step‑by‑Step for Codex)

1. **Scaffold**: npx create-next-app@latest (TypeScript, App Router). Add Tailwind, shadcn/ui.
2. **Env Guard**: Add a small lib/env.ts using Zod to validate required env vars on boot.
3. **Prisma**: Add schema from outline, generate client, run prisma migrate dev against Neon (use service connection string locally via proxy or sslmode=require). Seed admin user.
4. **Auth**: Configure Auth.js credentials provider with bcrypt hashing; add sessions; server-protect routes.
5. **Storage**: Add S3 client; implement signed upload URL endpoint; server-side PDF render/storage util.
6. **Email**: Add nodemailer transport + templating; implement email sender service.
7. **UI Shell**: Layouts, nav, toasts, auth pages.
8. **Profiles/Directory**: Forms with RHF+Zod, server actions, search with composite indices.
9. **Verification**: Admin queue + toggle + email.
10. **Messaging**: Thread model, optimistic UI, notifications.
11. **Jobs**: CRUD + filters.
12. **Events**: CRUD + RSVP.
13. **Forums**: Categories, threads, posts, votes, best answer.
14. **Mentorship**: Mentor profiles, availability slots CRUD, match/request flow, session & feedback.
15. **Newsletter**: Prefs + campaigns + tracking pixel + job queue for batching.
16. **Analytics**: Aggregations + charts + CSV/PDF export.
17. **Donations**: Stripe checkout + webhook + receipts.
18. **Admin Dashboard**: Users, verification, analytics.
19. **Testing**: Write unit tests for server libs; e2e for critical flows.
20. **CI/CD**: GitHub Actions: install, lint, typecheck, test, build, prisma migrate deploy.
21. **Deploy**: Vercel app linked to repo; Neon DB; storage bucket; Redis; Stripe keys; SMTP.

## Quality Bar

* Lighthouse perf/accessibility ≥ 90 on key pages.
* 100% typed public API (request/response DTOs) guarded by Zod.
* At least 80% unit test coverage on server libs; green e2e core paths: auth, profile edit, message send, job post, RSVP, forum post+vote, mentorship request, donation checkout, newsletter send.

## Security & Compliance

* Hash passwords with bcrypt (cost 12+). Never log secrets.
* Role-based authorization middleware + per-record access checks.
* Rate limit auth & messaging endpoints.
* Validate and virus-scan uploaded PDFs (clamav service in dev docker-compose is acceptable; stub in prod if needed).
* Content moderation hooks for posts/messages (simple bad-word filter + admin review flag).
* GDPR-friendly: delete account flow that removes PII or anonymizes posts.

## Developer Scripts

* pnpm db:push, pnpm db:migrate, pnpm db:seed
* pnpm dev, pnpm test, pnpm test:e2e, pnpm lint, pnpm typecheck

## README Must Include

* Setup instructions (env, Neon connection, migrations).
* How to run dev with docker-compose (web, redis, mailhog, clamav optional).
* Seed users: admin + sample alumni.
* Deployment checklist.

## Acceptance Criteria

* All modules implemented and navigable from the UI.
* Admin can verify users and see analytics; users can search directory, message, post jobs, RSVP to events, discuss in forums, use mentorship, manage newsletter prefs, donate, and receive receipts.
* Tests & CI passing; deployed app reachable at a public URL.