ES DISPATCH TOOL

Production Development Plan

Node.js + Express + PostgreSQL + Clerk + Railway

Created	February 2026
Target Team Size	30–50 users (max 100 concurrent)
Deployment Target	Railway (backend + DB)

Project Overview

This document is your full development roadmap from current state to production deployment. Each phase is ordered by priority — do not skip ahead. Phases 1 and 2 are blockers for everything else.

What Is Already Built

Task	Detail / Why	Status
Core dispatch logic	Fair ES assignment with last_assigned_at ordering	✓ DONE
BUSY ES fallback	Falls back to busiest-available when no AVAILABLE ES exists	✓ DONE
Offer lifecycle	Accept and reject endpoints with re-dispatch on reject	✓ DONE
Timeout cron job	Expires pending offers every 15s, re-dispatches to next ES	✓ DONE
User management CRUD	Create, update, delete, status-change endpoints	✓ DONE
DB schema + indexes	6 migration files, constraints, performance indexes	✓ DONE
Transaction safety	FOR UPDATE locking prevents duplicate assignments	✓ DONE
Health endpoint	GET /health (minor mount bug — see Phase 1)	✓ DONE

What Is Missing (Gaps Before Production)

Task	Detail / Why	Status
Authentication	All endpoints are fully open right now — anyone can call them	⚠ CRITICAL
Role-based access control	No distinction between dispatcher and ES permissions	⚠ CRITICAL
Request validation	No input sanitisation — malformed bodies hit the database directly	⚠ CRITICAL
Accept offer side-effects	Accept handler may not update enrollment status + assigned_es_id	⚠ CRITICAL
Migration tooling	Manual SQL file execution is fragile and un-auditable	TODO
Frontend	React + Vite app does not appear in folder structure yet	TODO
Pagination	List endpoints return all rows — will degrade at volume	TODO
Audit log	No record of who did what or when	TODO
Tests	No automated tests of any kind	TODO
Build + deploy pipeline	No production build script or Railway config	TODO

Phase Summary

#	Phase	Goal	Blocker For
1	**Backend Fixes & Auth**	Lock down the API, fix the accept-offer bug, add Clerk	**Everything**
2	**Validation & Error Contracts**	All inputs validated, consistent error responses	Frontend, Tests
3	**Database Migration Tooling**	Replace manual SQL with node-pg-migrate	Deployment
4	**Missing Endpoints & Pagination**	Fill gaps, add page/limit to list endpoints	Frontend
5	**Audit Log**	Record all mutations with user identity	Compliance
6	**Frontend**	React + Vite UI for dispatchers and ES users	User Acceptance
7	**Testing**	Verify dispatch fairness, race conditions, auth	Production Sign-off
8	**Production Deployment**	Railway deploy, env vars, health monitoring	Go-Live

PHASE 1: Backend Fixes & Authentication *Do this first. The app is fully open right now. Also fix the accept-offer bug before writing any frontend.*	**CRITICAL**

1.1 Fix the Health Route Mount (5 minutes)

⚠ **WARNING:** In index.ts, the health route is mounted as app.get("/health", healthRoutes) — this passes a Router as a callback, which doesn't work. Change it to app.use("/health", healthRoutes).

1.2 Verify the Accept Offer Handler (Critical Bug)

Open backend/src/routes/offers.ts and find the POST /offers/:id/accept handler. It must perform ALL of the following side-effects inside a single transaction. If any are missing, add them now.

Task	Detail / Why	Status
Mark offer ACCEPTED	UPDATE enrollment_offers SET status = 'ACCEPTED' WHERE id = $1	TODO
Expire all other pending offers	UPDATE enrollment_offers SET status = 'EXPIRED' WHERE enrollment_id = $1 AND id != $2 AND status = 'PENDING'	TODO
Set enrollment to ASSIGNED	UPDATE enrollments SET status = 'ASSIGNED', assigned_es_id = $1 WHERE id = $2	TODO
Set ES status to BUSY	UPDATE users SET status = 'BUSY' WHERE id = $1	TODO
Set responded_at timestamp	UPDATE enrollment_offers SET responded_at = NOW()	TODO

🛑 **CRITICAL:** Without step 3 and 4, the POST /enrollments/:id/complete endpoint will always fail because it checks enrollment.status === 'ASSIGNED'.

1.3 Install and Configure Clerk

Clerk handles all login, session management, and identity. You don't build any login screens — Clerk provides them.

Task	Detail / Why	Status
Install Clerk SDK	npm install @clerk/express (run inside /backend)	TODO
Add clerkMiddleware to index.ts	import { clerkMiddleware } from '@clerk/express'; app.use(clerkMiddleware()); — must be before all routes	TODO
Protect all routes with requireAuth()	import { requireAuth } from '@clerk/express'; app.use('/enrollments', requireAuth(), enrollmentRoutes); — repeat for /offers and /users	TODO
Add CLERK_PUBLISHABLE_KEY and CLERK_SECRET_KEY to .env	These are already in your .env — confirm they are correct	TODO
Test: unauthenticated request returns 401	Hit any protected endpoint in Postman without a Bearer token — should get 401, not data	TODO

Task	Detail / Why	Status
Test: authenticated request works	Get a session token from Clerk dashboard → add as Authorization: Bearer <token> in Postman → endpoint should respond normally	TODO

1.4 Link Clerk Identity to Your Users Table

Clerk manages login but your users table manages roles and dispatch state. You need to connect them.

Task	Detail / Why	Status
Create migration 007_add_clerk_id.sql	ALTER TABLE users ADD COLUMN IF NOT EXISTS clerk_id VARCHAR(255) UNIQUE; ALTER TABLE users ADD COLUMN IF NOT EXISTS email VARCHAR(255);	TODO
Create POST /users/sync endpoint	Called on first login to create or find a user record by clerk_id. Body: { clerk_id, email, name }. If user exists, return them. If not, create with role='ES' by default.	TODO
Create requireRole middleware	src/middleware/requireRole.ts — looks up user by clerk_id, checks role matches allowed list, returns 403 if not. See Phase 1 detail below.	TODO
Apply role guards to sensitive routes	Only DISPATCHER can POST /enrollments. Only ES can accept/reject their own offers. Only DISPATCHER can DELETE /users.	TODO

requireRole Middleware — what it needs to do

Create the file src/middleware/requireRole.ts with this logic:

- Get the Clerk user ID from req.auth.userId (available after clerkMiddleware runs)
- Query your users table: SELECT role FROM users WHERE clerk_id = $1
- If no row found → 403 with error: 'User not registered in system'
- If role is not in the allowed list → 403 with error: 'Forbidden'
- If role matches → call next()
- Export as: export const requireRole = (...roles: string[]) => async (req, res, next) => { ... }
- Usage in routes: router.post('/', requireRole('DISPATCHER'), async (req, res) => { ... })

PHASE 2: Validation & Error Contracts *Every write endpoint needs validated inputs. Unvalidated data reaching PostgreSQL causes unpredictable failures.*	**CRITICAL**

2.1 Install Zod

Zod is a validation library. It lets you declare the shape of expected input and will automatically reject anything that doesn't match.

- Run: npm install zod (inside /backend)
- Create src/middleware/validate.ts — a reusable middleware factory that takes a Zod schema and rejects bad requests before they hit your route handler
- Standard rejection response: { error: 'Validation failed', details: { fieldErrors: {...} } } with HTTP 400

2.2 Add Schemas for Every Write Endpoint

Task	Detail / Why	Status
POST /enrollments schema	premise_id: string min 1, requested_by: number integer positive, timeslot: string min 1	**TODO**
POST /users schema	name: string min 1, role: enum(['ES','DISPATCHER']), status: enum(['AVAILABLE','BUSY','UNAVAILABLE'])	**TODO**
PUT /users/:id schema	All fields optional (partial update) but same type rules apply	**TODO**
PATCH /users/:id/status schema	status: enum(['AVAILABLE','BUSY','UNAVAILABLE'])	**TODO**
POST /users/sync schema	clerk_id: string min 1, email: string email format, name: string min 1	**TODO**

2.3 Standardise All Error Responses

Right now some routes return { error: 'message' } and some return different shapes. Pick one format and stick to it everywhere.

Recommended standard:

- Success: standard HTTP 200/201 with data payload
- Client error: HTTP 400/403/404 with { error: 'Human readable message', code: 'MACHINE_CODE' }
- Server error: HTTP 500 with { error: 'Internal server error' } — never expose stack traces
- Create a small helper: src/utils/apiError.ts — function apiError(res, status, message, code?) that formats and sends the response

PHASE 3: Database Migration Tooling *Replaces the manual 'run these SQL files in order' process with a proper migration system.*	**TODO**

3.1 Install node-pg-migrate

- Run: npm install node-pg-migrate (inside /backend)
- Add to package.json scripts: "migrate:up": "node-pg-migrate up" and "migrate:down": "node-pg-migrate down"
- Create a database.json or pgm config pointing to your DB connection string

3.2 Convert Existing SQL Files

Create a migrations/ folder and convert your 6 existing SQL files into numbered migration files. node-pg-migrate uses a timestamp prefix format.

Task	Detail / Why	Status
001_create_users	Convert 001_create_users.sql → migrations/1_create_users.js	**TODO**
002–006 existing files	Convert each remaining SQL file in the same way	**TODO**
007_add_clerk_id	New migration from Phase 1.4	**TODO**
008_create_audit_log	New migration from Phase 5	**TODO**
Mark existing migrations as run	If DB already exists, insert migration records manually to prevent re-running: INSERT INTO pgmigrations (name, run_on) VALUES ('...')	**TODO**

i **NOTE:** For Railway deployment, your start script will run migrations automatically before starting the server. This means every deploy is self-migrating.

PHASE 4: Missing Endpoints & Pagination *Fill the remaining API gaps and add pagination to all list endpoints before building the frontend.*	**TODO**

4.1 Missing Endpoints to Add

Task	Detail / Why	Status
GET /enrollments/:id	Single enrollment with its full offer history joined in. Needed for detail views.	**TODO**
GET /offers/:id	Single offer detail. Used when ES views their current offer.	**TODO**
GET /users	List all users — currently exists? Confirm it's implemented and add pagination.	**TODO**
GET /users/:id	Single user profile. Needed for ES dashboard.	**TODO**
GET /enrollments?status=X	Filter enrollments by status. Dispatchers need to see WAITING vs ASSIGNED vs COMPLETED.	**TODO**
GET /offers?es_id=X	List offers for a specific ES. Needed for ES view of their pending/historical offers.	**TODO**

4.2 Add Pagination to All List Endpoints

Every GET endpoint that returns multiple rows needs pagination. Without it, a GET /enrollments call that returns 10,000 rows will freeze both the server and the browser.

- Accept query params: ?page=1&limit=20
- Validate: page >= 1, limit between 1 and 100
- Use SQL: LIMIT $1 OFFSET $2 where offset = (page - 1) * limit
- Include COUNT(*) OVER() in SELECT to get total rows without a second query
- Return: { data: [...rows], total: N, page: 1, limit: 20, totalPages: X }

PHASE 5: Audit Log *Records who did what and when. Required for operational accountability in a field service context.*	**TODO**

5.1 Create the Audit Log Table

This is migration 008. The table structure:

- id SERIAL PRIMARY KEY
- user_id INTEGER REFERENCES users(id) ON DELETE SET NULL — who performed the action
- action VARCHAR(100) NOT NULL — e.g. 'ENROLLMENT_CREATED', 'OFFER_ACCEPTED', 'USER_STATUS_CHANGED'
- entity_type VARCHAR(50) — e.g. 'enrollment', 'offer', 'user'
- entity_id INTEGER — the ID of the affected record
- metadata JSONB — any extra context (old value, new value, reason)
- created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

5.2 Add Audit Inserts to Existing Transactions

Inside each existing transaction, add an INSERT INTO audit_log(...) call before the COMMIT. The user_id comes from req.auth.userId resolved to your internal user ID (you'll have this after Phase 1).

Task	Detail / Why	Status
Enrollment created	Log action='ENROLLMENT_CREATED', entity_type='enrollment', entity_id=enrollment.id	**TODO**
Offer accepted	Log action='OFFER_ACCEPTED', entity_type='offer', entity_id=offer.id	**TODO**
Offer rejected	Log action='OFFER_REJECTED', include reason in metadata if available	**TODO**
Offer expired (cron)	Log action='OFFER_EXPIRED' — user_id will be null since cron has no user context	**TODO**
Enrollment completed	Log action='ENROLLMENT_COMPLETED'	**TODO**
User status changed	Log action='USER_STATUS_CHANGED', metadata: { from: oldStatus, to: newStatus }	**TODO**

PHASE 6: Frontend *React + Vite app. Build this after the backend API is stable and fully protected.*	**TODO**

⚠ **WARNING:** The README mentions a React + Vite frontend but the folder doesn't exist yet. Create it with: npm create vite@latest frontend -- --template react-ts

6.1 Setup & Dependencies

Task	Detail / Why	Status
Scaffold Vite app	npm create vite@latest frontend -- --template react-ts	**TODO**
Install Clerk React SDK	npm install @clerk/clerk-react — provides ClerkProvider, useAuth, useUser, SignIn component	**TODO**
Install TanStack Query	npm install @tanstack/react-query — handles all API calls, caching, loading/error states	**TODO**
Install a UI library	Recommended: shadcn/ui (Tailwind-based, copy-paste components). Alternative: Chakra UI or MUI.	**TODO**
Create API client	src/lib/api.ts — a wrapper around fetch that automatically attaches the Clerk session token as Bearer header	**TODO**

6.2 Views to Build

Two distinct user roles need different views. Build shared layout first, then role-specific pages.

Dispatcher Views

Task	Detail / Why	Status
Dashboard	Summary counts: WAITING enrollments, AVAILABLE ES, BUSY ES, pending offers. Auto-refreshes every 30s.	**TODO**
Enrollment List	Paginated table with status filter. Columns: ID, premise, timeslot, status, assigned ES, created time.	**TODO**
Create Enrollment form	Form fields: premise_id, timeslot. Submit → POST /enrollments → shows dispatch result.	**TODO**
Enrollment Detail	Full offer history for a single enrollment.	**TODO**
User Management	List all users, change status (AVAILABLE/BUSY/UNAVAILABLE), create/edit users.	**TODO**

ES (Energy Specialist) Views

Task	Detail / Why	Status
My Current Offer	Shows the ES their active PENDING offer (if any). Accept / Reject buttons. Shows countdown to expiry.	**TODO**

Task	Detail / Why	Status
My History	Paginated list of past accepted/rejected/expired offers for this ES.	**TODO**
My Status	Allows ES to toggle their own status (AVAILABLE / UNAVAILABLE). Cannot set themselves BUSY — that's system-managed.	**TODO**

6.3 Authentication Flow

- Wrap the entire app in <ClerkProvider publishableKey={...}>
- Use <SignedIn> and <SignedOut> components to conditionally render pages vs. login screen
- On first login, call POST /users/sync with the Clerk user details to create the internal user record
- Use useUser() hook to get role and drive conditional rendering (dispatcher dashboard vs ES dashboard)

PHASE 7: Testing	
Verify the most critical behaviours before deploying to production. Focus on correctness, not coverage.	TODO

7.1 Setup

- Install: npm install -D vitest supertest @types/supertest (inside /backend)
- Add to package.json: "test": "vitest run" and "test:watch": "vitest"
- Create backend/src/__tests__/ folder

7.2 Priority Test Cases

These are the tests that matter most for a dispatch system. Write them in this order.

Task	Detail / Why	Status
Accept offer — all side-effects	After accepting: enrollment.status='ASSIGNED', assigned_es_id set, ES.status='BUSY', other offers EXPIRED. This is the most critical test.	TODO
Dispatch fairness	Create 5 ES users with different last_assigned_at values. Submit 5 enrollments. Verify assignment order matches last_assigned_at ASC.	TODO
Concurrent dispatch — no duplicate assignment	Use Promise.all to fire 10 simultaneous POST /enrollments. Verify each gets a distinct ES assigned. This tests your FOR UPDATE lock.	TODO
Offer timeout — re-dispatch	Create an offer, manually set created_at to 10 minutes ago, trigger the cron job, verify: offer EXPIRED, ES UNAVAILABLE, new offer created.	TODO
Complete enrollment — ES freed	Accept an offer (ES goes BUSY), complete the enrollment, verify ES.status returns to AVAILABLE.	TODO
Auth — 401 without token	All protected endpoints return 401 when called without Authorization header.	TODO
Validation — 400 on bad input	POST /enrollments with missing fields returns 400 with helpful error message.	TODO

i **NOTE:** You can test these via Postman initially, but the concurrent dispatch test requires code (Promise.all) — you cannot do that in Postman manually.

PHASE 8: Production Deployment (Railway) *Deploy backend + database to Railway. Frontend to Vercel (recommended) or Railway.*	TODO

8.1 Prepare the Backend for Production Build

Task	Detail / Why	Status
Add build and start scripts	package.json: "build": "tsc", "start": "node-pg-migrate up && node dist/index.js"	TODO
Configure tsconfig for output	Ensure outDir is set to ./dist in tsconfig.json	TODO
Add .gitignore entries	Make sure dist/, node_modules/, and .env are all in .gitignore	TODO
Switch DB connection to DATABASE_URL	Railway provides a single DATABASE_URL connection string. Update src/config/db.ts to use it: const pool = new Pool({ connectionString: process.env.DATABASE_URL, ssl: { rejectUnauthorized: false } })	TODO
Set pool size	For 30–50 users: max: 10 is sufficient. Add idleTimeoutMillis: 30000, connectionTimeoutMillis: 3000	TODO

8.2 Railway Setup

Task	Detail / Why	Status
Create Railway project	railway.app → New Project → Deploy from GitHub repo	TODO
Add PostgreSQL service	In Railway project → Add Plugin → PostgreSQL. Copy the DATABASE_URL it generates.	TODO
Set environment variables	In Railway service settings → Variables. Add: DATABASE_URL, PORT=4000, CLERK_SECRET_KEY, CLERK_PUBLISHABLE_KEY	TODO
Set start command	Railway Settings → Deploy → Start Command: npm start	TODO
Run migrations on first deploy	The start script (npm start) already runs node-pg-migrate up before starting the server. First deploy will create all tables.	TODO
Verify health endpoint	After deploy, hit https://your-app.railway.app/health — should return { status: 'ok' }	TODO

8.3 Frontend Deployment

Vercel is the easiest option for a Vite + React app. It's free for small teams and deploys automatically from GitHub.

Task	Detail / Why	Status
Set VITE_API_URL env var	In Vercel project settings → Environment Variables. Set VITE_API_URL to your Railway backend URL.	TODO

Task	Detail / Why	Status
Set VITE_CLERK_PUBLISHABLE_KEY	In Vercel, also add your Clerk publishable key as VITE_CLERK_PUBLISHABLE_KEY	**TODO**
Update CORS on backend	In index.ts, change app.use(cors()) to app.use(cors({ origin: 'https://your-vercel-app.vercel.app' })) to restrict to your frontend domain only	**TODO**
Add Vercel redirect rule	Create vercel.json with rewrites rule to redirect all routes to index.html for SPA routing	**TODO**

Pre-Launch Checklist

Use this as your final gate before sharing the app with users.

Security

☐ **All routes protected by requireAuth() — unauthenticated requests return 401**
☐ **Role guards in place — ES cannot call dispatcher endpoints and vice versa**
☐ **Request validation on all write endpoints — bad input returns 400 not 500**
☐ **CORS restricted to frontend domain only**
☐ **No stack traces exposed in 500 error responses**
☐ **.env file not in git — check .gitignore**

Data Integrity

☐ **Accept offer handler performs all 5 side-effects in a single transaction**
☐ **Concurrent dispatch test passes — no duplicate ES assignments**
☐ **Offer timeout job re-dispatches correctly**
☐ **Complete enrollment correctly frees the ES back to AVAILABLE**

Deployment

☐ **DATABASE_URL used instead of individual DB_* vars**
☐ **Migrations run automatically on start**
☐ **Health endpoint returns 200 on production URL**
☐ **CLERK_SECRET_KEY set in Railway environment — NOT in code**
☐ **Frontend VITE_API_URL points to Railway URL, not localhost**

Functional

☐ **Dispatcher can create an enrollment and see it dispatched to an ES**
☐ **ES can see their pending offer and accept or reject it**
☐ **Dispatcher can see enrollment move from WAITING → ASSIGNED → COMPLETED**
☐ **Offer timeout fires and re-dispatches if ES doesn't respond**

Known Issues & Decisions

Architecture Decisions Made

These are intentional design decisions given your team size and constraints. No action needed unless requirements change.

Decision	Rationale	Trade-off Accepted
No WebSocket / real-time push	Small team, polling every 30s is sufficient, much simpler to build and host	Dispatchers see a ~30s delay on status changes
Cron job for offer timeout (not event-driven)	Simple, reliable, sufficient for this load	Up to 15s delay before expired offers are detected
No Redis / caching layer	Pool of max 10 PG connections handles 30–50 concurrent users comfortably	Not applicable at this scale
Single Railway instance	30–50 users doesn't warrant horizontal scaling. Railway restarts crashed instances automatically.	If the process crashes, ~10s downtime before restart
node-cron runs in same process	Simple and sufficient. Alternative (separate worker) is over-engineering for this scale.	Timeout job pauses if server is under heavy load

Out of Scope (Post-Launch)

- Real-time notifications (WebSocket or SSE) — polling is fine for v1
- Mobile app — browser-responsive web UI covers this
- Reporting / analytics dashboard — add after stable usage data exists
- Multi-region deployment — not needed at this scale
- Automated CI/CD pipeline (GitHub Actions) — manual Railway deploys are fine to start

— End of Document —