Career Roadmap Tool - Project Responsibilities & Delivery Plan

Group 31 (TU Darmstadt) - Final version aligned with provided specifications

Document version: 1.0
Date: 2026-01-18

1. Scope and Non-Negotiables

This document defines what the team will build (and what it will not build) based strictly on the uploaded Spezifikationsdokument and the team responsibility outline, plus one agreed addition: local user registration and login.

- Do not expand scope beyond the specification without a logged change request (issue + approval).
- Data availability is uncertain (TUCaN, Moodle, etc.). The prototype must support structured mock or manually maintained data.
- No real student data during development; comply with GDPR/DSGVO constraints.
- Prototype must be deployable on TU infrastructure over HTTPS.
- Register/Login is required now (local accounts). University SSO may be integrated later, but is not required for v1.

2. Target Outcome (Definition of Done)

A running web prototype that demonstrates the minimal functional scope defined in the specification:

1. Retrieval and processing of examination regulation data (via TUCaN API if feasible; otherwise via mock/manual data).
2. Tracking completed study milestones and computing an “on-track” status.
3. Integration of career goals into a personalized roadmap.
4. Display of relevant TU support services and associated guidance.
5. Notification mechanisms for upcoming milestones, missed steps, and recommended support resources.
6. Local registration/login (added requirement).

3. In-App Student Milestones (Duolingo/Subway Model)

This section defines the milestone system that students see in the website/app. The milestone visualization is a “subway line” of stations. Completing modules and meeting requirements moves the student forward.

3.1 Principles

- Milestones are checkpoints derived from examination regulations and degree requirements (e.g., mandatory modules, CP thresholds, deadlines).

- Milestones must be configurable per regulation version; the system should not hard-code module names in logic.
- Every milestone has: a short label, description, requirements, status, and recommended next actions (with links to support services where relevant).
- The UI shows progression: Locked → Available → Completed (and optionally "Overdue").

3.2 Proposed Subway Line Stations (Generic, Regulation-Driven)

Because the specification does not enumerate exact milestone names or CP numbers, the app implements a generic engine. The following station types are used to build a line for a given regulation dataset:

- Onboarding stations
    - Account created
    - Examination regulation selected (or defaulted)
    - Roadmap generated (initial plan created)
- Progress stations
    - Mandatory module group completed (group defined by regulation data)
    - Credit point threshold reached (e.g., 30/60/90 CP - values configurable)
    - Key administrative deadline reached (date-based station: e.g., registration windows)
    - Career goal checkpoint (user selected a goal and the roadmap updates recommendations)
- Completion-prep stations
    - All mandatory requirements fulfilled
    - Thesis readiness (if the regulation data defines prerequisites)
    - Degree completion (final station)

Important: the exact ordering and thresholds are driven by the regulation dataset (TUCaN or maintained mock data). For v1, the system must support at least: (a) mandatory module groups and (b) CP threshold stations.

3.3 "On-Track" Status

The specification requires an "on-track" assessment. v1 should implement a simple, explainable rule set:

- Compute earned CP from completed modules.
- Compare earned CP and completed milestone stations against expected CP/milestone targets by time (e.g., by semester boundaries).
- Return a status with an explanation: On-track / At-risk / Off-track, plus the next 1-3 actions to recover.
- Keep targets configurable in data so it can match different regulations or program expectations.

3.4 Data Model (Minimum)

Entity	Purpose	Key fields (v1)
MilestoneDefinition	Defines a station on the subway line	id, regulation_id, order_index, type, label, description,

		rule_payload (JSON), expected_by (optional)
MilestoneProgress	Stores a user's status for a station	user_id, milestone_id, status, achieved_at (optional), computed_explanation
Module	Module catalog entry	module_code, name, CP, category/group, prerequisites (optional)
UserModuleCompletion	User completed modules	user_id, module_id, status, completed_at
CareerGoal	User interests/goals that influence roadmap	user_id, tags/goal_type, created_at
Notification	In-app alerts and reminders	user_id, type, message, due_at, read_at (optional)

4. Team Responsibilities (Detailed)

Responsibilities below are detailed to reduce overlap and prevent "everyone owns everything" confusion. Each area has an explicit owner, while others may contribute via PRs.

4.1 Amine & Emir - Backend Lead (Django + API Structure)

- Own the backend architecture and API contract stability (versioning, error format, pagination rules).
- Implement authentication for v1: register, login, logout, and protected endpoints (secure password hashing, token or session strategy).
- Deliver core REST endpoints required by the minimal functional scope:
- Endpoints (v1)
    - Regulation data ingestion (TUCaN integration if feasible; otherwise load mock/manual datasets)
    - Modules, CP totals, and completion tracking
    - Milestone definitions + user milestone progress + on-track computation
    - Career goals (store + retrieve to influence roadmap)
    - Support services listing + guidance content endpoints
    - Notifications: create, list, mark as read
- Configure dev environment (SQLite OK) and production-ready settings (ALLOWED_HOSTS, HTTPS awareness, secrets in env).
- CORS setup for local development; ensure prod routing works behind Traefik.
- Provide seed data tooling: fixtures or a Django management command so frontend has stable demo data.
- Add basic backend tests: auth flows + endpoint smoke tests to protect integration stability.

4.2 Stas - Data Modeling & Database (PostgreSQL later)

- Own the domain model: regulations, modules, milestones, user progress, career goals, support services, notifications.
- Create and maintain an ER diagram and short data dictionary (what each field means).
- Implement Django models and manage migrations carefully (coordinate migration merges).

- Define how regulation datasets are represented: tables/JSON fields, and how to support multiple regulation versions.
- Prepare Postgres readiness for deployment: separate DB/user/schema for your team; document required environment variables.
- Define indexes and constraints to prevent duplicate data (e.g., unique module code per regulation version).

4.3 Alp - Frontend Lead (React + UI Foundation)

- Own frontend architecture: routing, page structure, component conventions, and UI consistency.
- Implement the main pages listed in the team responsibility outline and align them with the specification:
- Pages (v1)
    - Start/landing
    - Dashboard/overview (progress, on-track summary, next actions)
    - Study overview (modules grouped per regulation)
    - Module overview/detail
    - Roadmap view (subway milestone line + career goal integration)
    - Support services and events view (even if populated by mock/manual data)
    - Notifications center
- Build reusable UI components: loading, error states, list/cards, filters, milestone station components.
- Ensure accessibility basics: readable contrast, keyboard navigation for primary flows, clear empty states.
- Maintain a mock-data mode that mirrors API payloads so UI can progress independently when integration is blocked.

4.4 Yigit - Frontend Integration + API Consumption

- Own API integration and data flow in React (services + hooks).
- Implement auth UI flows: registration, login, logout, protected routes, token/session persistence.
- Own the API client layer (base URL via env, standardized headers, consistent error handling).
- Wire all screens to real backend endpoints; remove mock reliance once endpoints are stable.
- Implement state handling for milestones: fetch definitions + progress, render station statuses, show on-track explanation.
- Integrate notifications: fetch, display, mark as read; show reminders contextually on roadmap/milestone stations.
- Integration testing: validate payload shapes and edge cases; open issues for contract mismatches quickly.

5. Backend - Frontend Connection Plan (Now and Later)

5.1 Local Development

- Run Django API on localhost (e.g., :8000) and React on localhost (e.g., :3000).
- Enable CORS for the React origin in Django during development.
- Use environment variables in React for API base URL (dev vs production).

5.2 Production (Shared Server with Traefik)

- Deploy frontend as built static assets served by a web server container (or a lightweight static server).
- Deploy backend via Gunicorn/Uvicorn behind Traefik.
- Prefer subdomain routing: frontend at app.<domain>, backend at api.<domain> (clean separation).
- If path-based routing is required (e.g., /api), ensure the frontend never intercepts /api routes.
- Store secrets only in environment variables (never commit).

6. Sharing the Server with Another Team (Traefik exists)

Because another team already manages Traefik, your responsibility is to integrate cleanly without breaking their routes. This requires strict isolation and naming discipline.

- Use Docker Compose for your stack (frontend, backend, db).
- Use a dedicated Docker network and unique container names to avoid collisions.
- Coordinate unique Traefik router/service names and unique hostnames (subdomains) with the other team.
- Expose only what Traefik needs; do not bind extra ports publicly.
- Use a separate Postgres database and user for your app (no shared credentials).
- Set container resource limits (CPU/memory) so one team cannot starve the other.
- Logging: keep your logs separate and easy to inspect.

7. GitHub and Coding Practices to Avoid Conflicts

7.1 Branching and PR Rules

- Never commit directly to main. Work on feature branches (feature/<name>-<topic>).
- Open a PR for every change; at least one reviewer required.
- Keep PRs small (focused). Large PRs create merge conflict risk.
- Update your branch from main frequently (rebase or merge main into branch).
- PR checklist: tests pass, no debug prints, no secrets, migrations reviewed, frontend builds.

7.2 Django Migration Rules (Conflict Hotspot)

- Only merge model changes with migrations via PR (never force-push to main).
- When two branches touch models, coordinate: merge one PR first, then rebase the other and regenerate migrations if needed.
- Prefer one migration per logical change; avoid editing old migrations once merged.
- If migration conflicts happen: rebase onto main, run makemigrations again, test migrate on a clean DB.

7.3 Frontend Lockfile and Dependency Discipline

- Do not edit package-lock.json manually. Use npm install and commit both package.json + lockfile together.
- Avoid unnecessary dependency additions; agree before adding major libraries.
- Keep a single API base module (api.js) - no scattered fetch/axios setups.

7.4 API Contract and Integration Discipline

- Maintain an API contract document (endpoints + example payloads).
- Backend changes to response shape require a coordinated PR or a version bump.
- Frontend should validate assumptions and fail gracefully (loading/error states).
- Use consistent naming across backend serializers and frontend models (decide on snake_case vs camelCase once).

7.5 Communication and Ownership

- Single owner per area: backend API stability (Amine/Emir), data model + migrations (Stas), UI structure (Alp), integration layer (Yigit).
- Use issues for scope changes and decisions (especially on milestone rules and regulation data format).
- Write short decision records in docs/ (what was decided, why, and when).

8. References

1) Spezifikationsdokument: Career Roadmap Tool - Group 31 & FEC (November 2025).

2) TPSE Verantwortungen (team responsibility outline).