January 15, 2026

TermWise: OSU Course Planner

Abderrahmane Rhandouri **Eduardo Balzan** **Quinn Carey**

Abstract

TermWise is a web app that helps Oregon State University students build a term-by-term plan without repeatedly hunting through the catalog, scheduler, and prerequisite pages. Students select a major (starting with Computer Science), enter completed/planned courses, and drag classes into Fall/Winter/Spring/Summer terms. As the plan is built, TermWise checks prerequisites and term offerings, warns about invalid sequences, and summarizes credit load and progress. The goal is to reduce planning mistakes (e.g., missing prerequisites or waiting a full year for a course), reduce time spent cross-referencing sources, and make it easier to iterate on "what-if" plans before registration. Our MVP focuses on a reliable course database, prerequisite validation, and a simple planner UI, with optional grade/GPA calculation to support academic decision-making.

Contents

1 Team Info

1.1 Team members and roles

- Abderrahmane Rhandouri — Product + UX Lead (requirements, wireframes, usability, UI implementation support)
- Eduardo Balzan — Full-Stack Lead (core architecture, planner features, integration, data pipeline)
- Quinn Carey — Backend + QA/DevOps Lead (API + database, testing strategy, CI, release workflow)

1.2 Project artifacts

- Git repository: https://github.com/CareyQT/Software-Engr-II
- Issue tracker/backlog: GitHub Issues (same repo)
- Project board (sprint planning): GitHub Projects (linked from repo)
- UI mockups (if used): Figma link: https://www.figma.com/design/I1udQyiFAr4RtYFuvXuunC/CS-362-Figma?node-id=0-1&t=Pgan5NGFZADgzv7Q-1
- Living document: `/docs/typst/main.typ` (generated PDF committed for submissions)

1.3 Communication channels and rules

- Primary chat: Microsoft Teams (private team channel)
- Async decisions: GitHub Issues (each task has an issue; decisions documented in comments)
- Meetings: 1 weekly sync (30–45 minutes) + additional short check-ins as needed
- Response expectations:
  - Discord: respond within 24 hours on weekdays
  - GitHub review requests: respond within 48 hours on weekdays
- Rules:
  - No direct commits to `main` (PRs only)
  - Every PR must reference an issue and include a short test note ("how I verified")
  - At least 1 approval required before merge
  - If blocked >24 hours, post the blocker in Discord + the issue thread

2 Product Description

2.1 Goal

Help OSU students create a valid, term-by-term class plan for an academic year (and beyond) by automatically applying constraints like prerequisites, co-requisites (where supported), and which terms a course is offered. The system should reduce planning mistakes and time spent cross-checking multiple OSU sites.

2.2 Current practice

Today, students typically plan by manually consulting the OSU catalog for prerequisites, the Schedule of Classes for term offerings, and degree audit tools to estimate progress. This is slow, error-prone, and repetitive—especially when a student changes a plan and must re-check prerequisite chains and course availability across Fall/Winter/Spring/Summer. Existing tools often feel fragmented (catalog here, planner elsewhere, GPA tools elsewhere) and don't provide immediate "this plan is invalid because…" feedback while building the schedule.

2.3 Novelty

TermWise combines (1) a planner UI and (2) automated validation against prerequisite rules and term offerings in one place, with fast feedback as students build "what-if" schedules. Instead of searching course pages repeatedly, the app surfaces key constraints directly inside the planning workflow (warnings, unmet prereqs, next eligible term). The project is not trying to replace official registration; it targets the planning step before registration, where students need clarity and iteration speed.

2.4 Effects

If successful, TermWise will help students:

- Avoid delaying graduation due to missed prerequisites or missed once-per-year offerings
- Build more balanced term credit loads and spot overload early
- Reduce advisor meeting time spent on basic prerequisite/availability lookups
- Make faster, more confident schedule decisions before registration windows open

2.5 Technical approach

We will build a web application with:

- Frontend: React/Next.js planner UI (drag-and-drop term columns, course search, plan summary)
- Backend: REST API for courses, offerings, and plan validation
- Database: relational storage (e.g., PostgreSQL) for course metadata, offering terms, and prerequisite structures
- Data ingestion: a small ETL script that populates course data from public OSU sources (catalog + schedule listings) into our database
- Validation engine: given a student's completed courses + planned terms, compute unmet prerequisites and mark courses as eligible/ineligible per term

(Exact tech choices may be adjusted based on team strengths and course expectations, but the architecture remains: UI + API + DB + ingestion + validator.)

2.6 Risks

The most serious risk is accurately interpreting and validating prerequisite rules at the project scale and within time. OSU prerequisites are sometimes written in complex natural language (OR/AND groups, grade requirements, concurrent enrollment, placement tests). Mitigation:

- Scope the first release to a well-defined subset (e.g., OSU Computer Science core + common electives) with test cases for each prereq pattern we support
- Store prerequisites in a structured internal format (not only raw text), and manually curate edge cases for the subset
- Add automated tests using real prerequisite strings from our supported course set to prevent regressions

2.7 Major features (MVP)

- **Feature 1**: Course explorer and search
  - Search/filter by subject/number, credits, and offered terms (Fall/Winter/Spring/Summer where known)
- **Feature 2**: Term-by-term planner
  - Drag/drop courses into term columns, see total credits per term, and reorder terms easily
- **Feature 3**: Prerequisite and eligibility validation
  - Real-time warnings for unmet prerequisites; highlight the earliest eligible term for a selected course
- **Feature 4**: Saved plans (persistence)
  - Store a user's plan and completed courses; reload and edit later (basic accounts or local persistence acceptable for MVP)
- **Feature 5**: Grade/GPA calculator (simple)
  - Let users enter expected grades and compute term GPA and cumulative GPA estimates

2.8 Stretch goals

- **Stretch 1**: Auto-plan suggestions
  - Given constraints (max credits/term, target graduation term), generate a recommended sequence
- **Stretch 2**: Degree progress tracking (limited)
  - For one major (starting with Computer Science), show requirement groups and completion status from the user's plan
- **Stretch 3**: Share/export
  - Export plan to PDF and/or share a read-only link for advisors/peers

2.9 Use Cases (Functional Requirements)

Use Case 1 (Abderrahmane Rhandouri): Search and add a course to a term plan.

- **Actors**
  - Primary: Student
- **Triggers**
  - Student opens TermWise and wants to add one or more courses for a specific term.
- **Preconditions**
  - Course dataset is available (at minimum for the supported subset).
  - Student is viewing a plan that contains at least one term column (e.g., Fall/Winter/Spring/Summer).
- **Postconditions (success scenario)**
  - Selected course appears in the chosen term column.
  - Term credit total and summary update to include the course's credits.
- **List of steps (success scenario)**
  1. Student types a subject/number or keyword into the course search box.
  2. System returns matching courses with key metadata (credits, typical offering terms when known).
  3. Student selects a course from results.
  4. Student chooses a target term (drag-and-drop or "Add to term" action).
  5. System places the course into that term and updates totals.
- **Extensions/variations of the success scenario**
  - Student filters by offered term (e.g., "Spring") before selecting a course.
  - Student adds multiple courses in a row using recent searches.
  - Student moves a course between terms by dragging it to another column.
- **Exceptions: failure conditions and scenarios**
  - Search query returns no results (system displays "no matches" and suggests a different query).
  - Student tries to add a duplicate course to the same term (system blocks or warns, depending on policy).
  - Course metadata is missing credits or offering info (system still allows adding, but marks metadata as unknown).

Use Case 2 (Eduardo Balzan): Validate plan prerequisites and term eligibility.

- **Actors**
  - Primary: Student
  - Supporting: Validation engine
- **Triggers**
  - Student adds/moves a course, edits completed courses, or explicitly clicks "Validate plan".
- **Preconditions**
  - Student has a plan with at least one planned course.
  - Prerequisite rules exist in structured form for the supported course subset.
- **Postconditions (success scenario)**
  - System marks each planned course as eligible/ineligible for its term.

  - System shows clear reasons for ineligibility (missing prerequisites, not offered that term).
  - System can indicate the earliest eligible term (within the plan horizon) when possible.
- **List of steps (success scenario)**
  1. Student changes the plan (add/move/remove a course) or updates completed courses.
  2. System recomputes eligibility for each term in order (completed + prior planned terms).
  3. For any ineligible course, system lists unmet prerequisite(s) and/or offering mismatch.
  4. Student adjusts the plan (move prerequisites earlier, swap term, etc.).
  5. System re-validates and clears warnings when constraints are satisfied.
- **Extensions/variations of the success scenario**
  - Student toggles "assume concurrent enrollment" for supported co-requisite patterns.
  - Student views a "Why invalid?" panel that expands prerequisite chains.
- **Exceptions: failure conditions and scenarios**
  - Prerequisite rule is not supported by the MVP parser/format (system labels it "manual check required" and does not claim validity).
  - Course offering data is unavailable or ambiguous (system warns that offering is unknown and avoids a hard failure).
  - Validation cannot complete due to internal error (system keeps last-known results and shows a recoverable error message).

Use Case 3 (Quinn Carey): Save and reload a plan (persistence).

- **Actors**
  - Primary: Student
  - Supporting: Backend API + database (or local storage for MVP mode)
- **Triggers**
  - Student clicks "Save plan" or returns later and wants to continue planning.
- **Preconditions**
  - Student has a plan with at least one term and optional completed courses.
  - System has a configured persistence mechanism (account-based save, or local/device save for MVP).
- **Postconditions (success scenario)**
  - Plan data is persisted and can be retrieved later without losing term order or course placements.
  - Student can resume editing from the restored state.
- **List of steps (success scenario)**
  1. Student clicks "Save plan".
  2. System validates the plan data schema (required fields present, no malformed entries).
  3. System stores the plan and returns a confirmation (and optionally a plan name or timestamp).
  4. Student returns later and opens "My plans" (or auto-load occurs for local save).
  5. System loads the plan and displays the same term structure and courses.
- **Extensions/variations of the success scenario**
  - Student maintains multiple plans (e.g., "Plan A", "Plan B") and switches between them.
  - Student exports a read-only snapshot (PDF/export) for sharing (may be stretch depending on implementation).
- **Exceptions: failure conditions and scenarios**
  - Network error during save (system shows "save failed", keeps local state, allows retry).
  - Stored plan is from an older schema version (system runs a migration or prompts to re-save).
  - Student exceeds storage limits (system blocks save and explains what to remove/simplify).

2.10 Non-functional Requirements

- **Usability and accessibility**

    - The planner must be usable with keyboard-only navigation for core actions (search, add course, move course, view validation messages), and provide clear, readable error/warning states.
- **Performance**
    - For typical plans (e.g., 3–8 terms and up to 60 planned courses in the supported subset), validation feedback should feel immediate; the UI should not freeze during recomputation.
- **Security and privacy**
    - If accounts are used, authentication must use secure practices (hashed passwords or external auth provider), and the system should store only the minimum necessary user data (plan + completed courses) and avoid collecting sensitive personal data.
- **Reliability**
    - The app must fail gracefully when course data is missing or inconsistent, clearly marking "unknown" rather than silently producing incorrect validations.

2.11 External Requirements (Specialized to TermWise)

- **Robust against expected errors**
    - TermWise must handle invalid inputs (unknown course codes, malformed plan data, impossible term orderings) with actionable messages, and must not crash the UI or API on bad requests.
- **Accessible deployment for others**
    - TermWise will be deployed as a public web app with a stable URL that course staff and peers can access for evaluation; deployments should include a basic status/health page and a sample dataset for demonstration.
- **Buildable from source + documented for new developers**
    - The repository will include clear setup instructions (prerequisites, environment variables, database setup), and a one-command local dev workflow (e.g., package scripts and/or containerized services). CI will build and run tests on pull requests so others can verify the build.
- **Scope matches team resources**
    - For the quarter project, TermWise will target a limited, well-tested subset of OSU courses (e.g., CS core + common prereq chains) and a limited prereq rule grammar; unsupported prerequisite formats will be explicitly labeled instead of guessed.

2.12 Team process description

Development process (quarter-long). We will use iterative, 1-week mini-sprints with a prioritized backlog in GitHub Issues. Each sprint ends with a measurable demo (running on main) and a short retrospective (what worked, what blocked us, what to change). Work is merged via pull requests only, with at least one reviewer approval and CI passing.

Software toolset (and why).

- **Version control and collaboration**: GitHub (Issues + PR reviews + Projects board) to keep scope visible and decisions documented.
- **Frontend**: Next.js + React for a fast, component-driven planner UI; a component library (e.g., shadcn/Radix) to build consistent accessible UI quickly.
- **Backend/API**: REST API (Node/Next API routes or a small server) to separate validation/data from UI and support future clients.
- **Database**: PostgreSQL for relational course/prereq/offering data and straightforward querying.
- **Testing**: unit tests for validation logic, integration tests for API endpoints, and end-to-end tests for core planner flows.
- **CI/CD**: GitHub Actions to build/test on PRs and produce reliable deploys.

Team roles (and justification).

- **Abderrahmane Rhandouri — Product + UX Lead**

    - Needed to keep requirements, use cases, and UI workflows consistent and to drive usability testing.
    - Suited due to focus on requirements, wireframes, and user-facing polish.
- **Eduardo Balzan — Full-Stack Lead**
    - Needed to define the overall architecture and ensure end-to-end integration (UI ↔ API ↔ DB ↔ ETL).
    - Suited due to ownership of core features and cross-cutting implementation.
- **Quinn Carey — Backend + QA/DevOps Lead**
    - Needed to ensure the API/data layer is reliable and that testing/CI/deployments are not an afterthought.
    - Suited due to focus on backend implementation, test strategy, and release workflow.

Weekly schedule (measurable milestones).

- **Week 1**
    - Abderrahmane: wireframes for planner/search/validation states completed and reviewed.
    - Eduardo: repo skeleton runs locally (frontend + backend stub) with a placeholder dataset.
    - Quinn: CI pipeline runs build + basic test job on PRs; deploy target chosen and documented.
- **Week 2**
    - Abderrahmane: course search UX works end-to-end with mock data (search ⟶ select ⟶ add).
    - Eduardo: database schema draft for courses/offerings/prereqs committed.
    - Quinn: API endpoint `GET /courses` returns paginated/filterable results from seed data.
- **Week 3**
    - Abderrahmane: drag-and-drop (or equivalent move) between term columns works in UI.
    - Eduardo: ETL script imports a small curated subset of courses with offerings/prereqs.
    - Quinn: API endpoint `POST /validate-plan` accepts plan payload and returns validation results.
- **Week 4**
    - Abderrahmane: validation messages displayed in-context (per course card + summary panel).
    - Eduardo: validation engine supports at least 2 prerequisite patterns (simple prereq and AND-group).
    - Quinn: automated tests added for those prerequisite patterns using real course examples.
- **Week 5**
    - Abderrahmane: “completed courses” input UI works and updates validation output.
    - Eduardo: offering-term checks integrated (flag course as not offered in chosen term when known).
    - Quinn: error handling and API input validation implemented (bad payloads return clear errors).
- **Week 6**
    - Abderrahmane: usability test session (at least 2 participants) completed; findings documented.
    - Eduardo: plan persistence MVP works (local save or account-backed; load restores exact layout).
    - Quinn: end-to-end test covers “search ⟶ add ⟶ validate ⟶ save ⟶ reload”.
- **Week 7**
    - Abderrahmane: UI polish pass on warnings/errors; accessibility checks on core flows.
    - Eduardo: prerequisite grammar expanded (e.g., OR-group support for supported subset).
    - Quinn: staging deployment updated; smoke test checklist documented.
- **Week 8**
    - Abderrahmane: export/share UX drafted (even if implemented as minimal export for MVP).
    - Eduardo: performance pass on validation for typical plan sizes; reduce noticeable delays.
    - Quinn: monitoring/logging basics added (request logging + error reporting plan).
- **Week 9**
    - Abderrahmane: demo script prepared and validated against fresh data.
    - Eduardo: integration bugs closed; “happy path” works without manual fixes.
    - Quinn: documentation complete (setup, run, test, deploy) and reproducible by a new machine.

- **Week 10**
  - Abderrahmane: final UI/UX fixes based on feedback; presentation-ready.
  - Eduardo: final feature freeze + cleanup; stretch features only if low risk.
  - Quinn: final release build; test report and deployment URL verified for course staff access.

Major risks (and mitigations).

- **Prerequisite complexity**
  - Risk: natural-language prereqs exceed our supported rule grammar.
  - Mitigation: limit course subset and explicitly label unsupported prereqs as "manual check required"; add regression tests for supported patterns.
- **Data accuracy and drift**
  - Risk: offerings/prereqs scraped/imported incorrectly or change between terms.
  - Mitigation: curated subset for MVP, checksums/manual spot checks, and visible "data last updated" metadata.
- **Integration and deployment delays**
  - Risk: UI/API/DB/ETL integration takes longer than expected; deploy issues block demos.
  - Mitigation: early vertical slice (Week 2–3), CI from Week 1, and staging deployment well before final weeks.

External feedback (when and how). External feedback is most useful after the first working vertical slice (planner + basic validation) and again once persistence and validation messages are stable. We will solicit feedback from OSU students (and/or an advisor if available) via short usability tests and will incorporate course staff feedback after milestone demos by translating it into GitHub issues for the next sprint.