ChatGPT

# School Performance Evaluation System (Next.js Implementation)

## Overview

This document outlines how to build a **School Performance Evaluation System** using **Next.js**. The system is based on a live demo from a school in the Sultanate of Oman and enables schools to upload and manage evidence of performance across several domains. The key objectives are:

- Provide a **user-friendly dashboard** that displays the number of uploaded evidence items, their status (approved, under review, rejected) and statistics by domain.
- Allow teachers and administrators to **upload evidence** (files or links) for specific standards and indicators. Uploaded evidence should support various file types (images, PDFs, videos) and self-evaluation notes.
- Enable **reviewers** to approve or reject evidence, add comments, and track the status of each submission.
- Offer **filtering** and **search** capabilities by domain, status, standard and indicator.
- Generate **reports** (with export to Excel) showing which domains have the highest or lowest evidence counts and provide a summary of recent submissions.
- Include **user management** features, such as adding/editing users, assigning roles (system manager, principal, supervisor, data entry), and maintaining an activity log.
- Integrate **AI-assisted tools** for writing evaluative phrases, calculating proficiency percentages, analyzing visit forms and drafting self-evaluation reports.
- Support **bilingual interfaces** (Arabic and English) and a **dark mode** option.

The demo emphasises a clean, modern interface where each domain is represented by a card with an icon, short description and a button to view details. Evidence statistics are shown in colourful yet restrained charts and counters. The overall look is approachable and professional, with emphasis on clarity rather than ornamentation.

## Color Design & UI Guidelines

### Choosing a Unified Color Palette

A common problem in internal tools is inconsistent colour usage. To avoid a “shitty” UI, choose a **single primary colour** and derive the rest of the palette from it. For example:

| Role | Colour | Hex code |
|---|---|---|
| Primary brand colour | Azure blue | `#3B82F6` |
| Secondary background | Soft light grey | `#F5F7FA` |
| Accent for success | Emerald green | `#10B981` |
| Accent for warning | Golden yellow | `#F59E0B` |

| Role | Colour | Hex code |
|---|---|---|
| Accent for error | Rose red | `#EF4444` |

These colours provide contrast without clashing. You can later swap the primary colour globally by editing a single value.

## Implementing the Palette

Use **Tailwind CSS** or plain CSS variables to centralize the palette. For instance, define colours in `tailwind.config.js`:

```
// tailwind.config.js
module.exports = {
  theme: {
    extend: {
      colors: {
        primary: {
          DEFAULT: '#3B82F6',   // azure blue
          light: '#60A5FA',    // hover state
          dark: '#2563EB',     // active state
        },
        success: '#10B981',    // green for approved evidence
        warning: '#F59E0B',    // yellow for items under review
        error: '#EF4444',      // red for rejected evidence
      },
    },
  },
};
```

By referencing these colour keys (`bg-primary`, `text-primary`, etc.) throughout components, the interface remains consistent. To change the colour scheme, modify the definitions above and rebuild.

## Dark Mode

Tailwind's `dark` variant or CSS classes can handle dark mode. For example:

```
<div className="bg-primary-50 dark:bg-gray-900 text-gray-700 dark:text-
gray-300">
  {/* content */}
</div>
```

Add a toggle button in the header to switch between modes. Use local storage or cookies to persist the choice.

### Icons and Emojis

Icons convey meaning and support navigation, but they must remain professional. Use **standard icon libraries** (e.g., Font Awesome or Heroicons) and avoid inserting silly or informal emojis. Playful emojis might appeal in casual apps, but they reduce the professional tone expected in an educational tool. Only include emojis when they serve a clear purpose and maintain a consistent, polished look.

## Project Structure & Technology Stack

Use **Next.js (v13 or later)** with the **App Router** for building both the front-end and server routes. Configure TypeScript for type safety. The recommended directory structure is:

```
ses-app/
├── app/                    # Next.js pages and layouts (App Router)
│   ├── layout.tsx         # Root layout (header, footer, theme provider)
│   ├── page.tsx           # Landing page
│   ├── dashboard/         # Dashboard pages
│   ├── upload/            # Evidence upload form
│   ├── evidence/          # Evidence list and detail pages
│   ├── users/             # User management pages
│   ├── reports/           # Reporting pages
│   └── api/               # API routes for server actions (file upload,
auth)
├── components/            # Reusable UI components
│   ├── Header.tsx
│   ├── Sidebar.tsx
│   ├── EvidenceCard.tsx
│   ├── Charts.tsx
│   └── …
├── lib/                   # Utility functions (e.g., database, auth, AI
calls)
├── models/                # Database models (if using an ORM like Prisma)
├── middleware.ts          # Authentication/authorization middleware
├── public/                # Static assets (logos, fonts)
├── styles/                # Global CSS and Tailwind config
└── tailwind.config.js
```

### Authentication & Authorization

Implement authentication using **NextAuth.js** or a custom JWT solution. Users should log in with a username and password to access protected routes. Define user roles (system manager, school principal, supervisor, data entry) in the database. Use middleware to enforce role-based access control. For example, store accessible routes in enums for maintainability; an answer on Stack Overflow suggests defining enums such as `DashboardRoutes` and `AdminRoutes` for consistent routing [1].

### Internationalization (i18n)

Use a library like **next-i18next** to manage translations. Store translation strings in JSON files (e.g., `public/locales/en/common.json` and `public/locales/ar/common.json`). Wrap the application with the `appWithTranslation` provider and use the `t` function to retrieve labels. Provide a language switcher in the UI that sets a cookie or session value.

### AI Assistant Integration

To integrate AI features (for evaluative phrases or proficiency calculations), encapsulate API calls in the `lib/` directory. For example, create a module `lib/ai.ts` that calls the OpenAI API or another service. The UI can then invoke these helper functions when generating reports or feedback.

## Data Models

The system revolves around domains, standards, indicators, evidence and users. A relational database (PostgreSQL or MySQL) works well, but a NoSQL solution like MongoDB may also suffice. The following table outlines core entities and their relationships:

| Entity | Key fields | Relationship |
|---|---|---|
| **Domain** | `id`, `name`, `axis` | A domain belongs to an axis (e.g., “Quality of Learning Outcomes”). One domain has many standards. |
| **Standard** | `id`, `domainId`, `code`, `name` | Belongs to a domain; a standard has many indicators. Codes follow the numbering (e.g., 2.1). |
| **Indicator** | `id`, `standardId`, `code`, `description` | Belongs to a standard; defines what evidence should demonstrate. |
| **Evidence** | `id`, `title`, `description`, `domainId`, `standardId`, `indicatorId`, `type`, `filePath`, `url`, `status`, `submittedBy`, `submittedAt`, `reviewedBy`, `reviewedAt`, `notes` | Each evidence item references a domain/standard/indicator and is either a **file** or **link**. `status` can be “under_review”, “approved”, or “rejected”. `notes` store reviewer comments. |
| **User** | `id`, `name`, `email`, `role`, `passwordHash` | Role determines accessible domains. Use salted hashes for passwords. |
| **ActivityLog** | `id`, `userId`, `action`, `timestamp`, `ip`, `userAgent` | Stores every activity (upload, approve, reject, edit). |

Additional tables can store self-evaluation entries, AI suggestions and report snapshots. Use a migration tool (e.g., Prisma Migrate) to manage schema changes.

### Extensibility and Customization

Design the codebase so that new domains, standards and indicators can be added without modifying the front-end logic. Store axes, domains, standards and indicators in the database and retrieve them via API calls rather than hard-coding them. Use dynamic components (e.g., `<Select>` components populated from the database) to generate dropdowns for domains and standards. When adding a new domain or indicator to the database, it should automatically appear in the UI.

Similarly, avoid scattering business logic across multiple components. Encapsulate data access in the `lib/` or `models/` layers and keep UI components generic. For example, use a generic `EvidenceForm` component that accepts a list of domains and standards as props. This makes it easy to extend the system with new features or custom fields without rewriting existing components.

## File Upload & Storage Best Practices

Handling file uploads securely is crucial. The process involves a client component for selecting files and a server component to handle the upload.

1. **Client-side selection:** Use an `<input type="file">` or drag-and-drop component to select files. Before uploading, validate size and type on the client to enhance user experience [2]. For example, restrict uploads to images/PDFs and limit size to 10 MB.

2. **Server-side handling:** Create an API route (`/app/api/upload/route.ts`) that accepts a `FormData` payload. Disable the default body parser and use a library like **formidable** or **multer** to parse the file stream. The Tekody guide demonstrates using `formidable` and saving files to an `uploads` directory [3]. Always generate unique filenames to avoid collisions and sanitize user input. After processing, respond with a JSON object containing the stored path.

3. **Validation:** Always validate the file type and size on the server, even if client validation exists [4]. Limit accepted MIME types and file sizes to prevent abuse [5].

4. **Storage location:** For small projects or prototypes, saving files in a `/public/uploads` directory via Next.js server actions may suffice [6]. However, the article cautions that storing sensitive data in `/public` exposes files to anyone with the URL [7]. For production, store files in a secure folder outside the public directory and serve them through authenticated endpoints. Alternatively, use cloud storage services like **Amazon S3**, **Google Cloud Storage** or **Azure Blob Storage** for scalability [8]. These services allow fine-grained access controls and support large files.

5. **Security:** Use HTTPS for all uploads [9]. Sanitize file names and remove special characters [10]. Implement authentication and authorization on upload endpoints so only permitted users can upload files [11]. Consider scanning uploaded files for malware and limiting upload rates to mitigate denial-of-service attacks [12]. When using cloud storage, restrict access tokens and rotate credentials regularly.

6. **Access control:** If storing files in cloud buckets, implement access control logic similar to the Edge Store example. Define contexts containing the user ID and role, then grant download permissions only to the file owner or administrators [13]. This prevents unauthorized users from viewing others' evidence.

7. **Link submissions:** When a user submits a URL instead of a file (e.g., YouTube or Google Drive), validate the URL format on the server. Store the link alongside the evidence record and treat it as a distinct type. Ensure that reviewers can open the link in a new tab and that malicious links are filtered through a safe-browsing API.

## Dashboard & Data Visualization

Implement a dashboard using charts and cards to summarize evidence status and domain statistics. A typical layout includes:

- **Counters** showing total evidence, items under review, approved items and rejected items.
- **Bar or column charts** representing the number of evidence items per domain and per status.
- **Latest submissions** displayed as a table or list with the submitter, status and upload date.

Libraries like **Chart.js**, **Recharts** or **ApexCharts** integrate well with React. The bar chart can group by domain, while a doughnut chart shows the proportion of approved, under review and rejected evidence. Data for charts should be fetched via API calls to ensure real-time updates.

## Evidence Upload & Review Workflow

1. **Upload Form:** Provide a form where users enter the title, description, select a domain, then choose a standard and indicator. If self-evaluation is required, include a rich text area for comments. Offer two inputs: one for file upload and another for direct URL submission. The user chooses either method, and the other field remains disabled.

2. **Saving Evidence:** On submission, send the data to an API route. The server creates an `Evidence` record and either saves the file (with validated name and type) or stores the URL. The evidence status defaults to **under review**.

3. **Review Interface:** Reviewers log in and see a list of evidence items. For each item, they can view the file or open the link, then decide to approve or reject. If rejected, they must provide a reason. The system updates the `status` and `notes` fields and records the reviewer's ID and timestamp.

4. **Notifications (optional):** Implement real-time notifications or email alerts to inform users when their evidence is approved or rejected.

## Filtering, Search & Reporting

### Filtering & Search

Provide UI controls to filter evidence by domain, status or submitter. Use controlled components (e.g., `<select>` and `<input>`) and update query parameters accordingly. For text search, allow queries like "2.1.1" to return all evidence related to indicator **2.1.1**. Maintain indexes on `code` fields in the database for efficient queries.

### Reporting

Create a **Reports** page that aggregates statistics across domains and time periods. Allow exporting reports to Excel using a library such as `xlsx` on the server. The reports can show:

- Which domains have the highest/lowest number of uploaded evidence.
- Recent evidence submissions with timestamps, domains and standards.
- Summary of approved vs. rejected items.

Provide date range pickers to generate time-bound reports. Ensure that exported files include column headers and are formatted for readability.

## User Management & Activity Log

### User Management

Create a user management interface where administrators can add, edit or delete users. Fields include name, email, role and assigned domains. Support batch user uploads by allowing administrators to import an Excel file; parse the file server-side and create user records accordingly. Provide buttons to reset passwords and assign roles.

### Activity Log

Log every significant action (login, upload, approval, rejection, deletion) with the user ID, timestamp, IP address and browser info. Display logs in a searchable table. This not only improves accountability but also simplifies audits.

## AI-Assisted Features

The system can integrate AI to assist users:

- **Evaluative phrase generation:** When a reviewer is writing feedback, call an AI model to suggest concise evaluative phrases based on the evidence description and indicator.
- **Proficiency calculations:** Use AI to compute proficiency percentages from self-evaluation forms or questionnaire responses.
- **Visit form analysis:** Analyze textual data from supervisor visit forms to identify areas for improvement.
- **Self-evaluation reports:** Generate draft reports summarizing strengths, weaknesses and development priorities.

Encapsulate each AI call in a function within `lib/ai.ts`, allowing the front-end to request suggestions asynchronously. Ensure that API keys are stored in environment variables and not exposed to the client.

## Deployment Considerations

### Environment & Infrastructure

- **Hosting:** Deploy the Next.js app to **Vercel**, **Netlify** or a containerized environment on **AWS**, **Azure** or **DigitalOcean**. For file storage, either mount a persistent volume or integrate with S3/ Blob storage. When using Vercel, be aware that the file system is immutable during runtime; thus, external storage is recommended for uploads.
- **Database:** Use a managed database service (e.g., PostgreSQL on RDS or Supabase) to handle persistence. Configure database connection strings via environment variables.
- **CI/CD:** Set up continuous integration to lint, test and build the application. Define environment variables (database URLs, file storage credentials, AI API keys) in the deployment platform.
- **HTTPS & Security:** Use HTTPS by default. Configure CORS policies on API routes. Keep dependencies updated and monitor for vulnerabilities.

### Iterative Development Strategy

Building this system should be **iterative**, not a monolithic effort. Work with your agent (e.g., Claude or Cursor) through cycles of planning, coding and review. An example roadmap:

1. **Initial setup:** Create the Next.js project, install Tailwind CSS, set up authentication and build the landing page with a unified colour scheme.
2. **Domain & standard management:** Define data models and seed the database with axes, domains, standards and indicators. Build pages to list and view these entities.
3. **Evidence upload & list:** Implement the upload form (file/link) and evidence list. Add client-side and server-side validation and store submissions in the database.
4. **Dashboard:** Build the dashboard with counters and charts. Fetch data from API routes and display statistics.
5. **Review workflow:** Add pages for reviewers to approve/reject evidence, with comment fields and status updates. Implement email/notification if desired.
6. **Filtering & search:** Add filtering by domain and status and search by code. Enhance queries with indexes.
7. **Reporting:** Implement report generation and export to Excel.
8. **User management:** Create admin pages for managing users and roles and for importing users from Excel. Add an activity log.
9. **AI integration:** Develop AI helpers for evaluative phrases and report generation.
10. **Polishing & deployment:** Refine the UI, optimize performance, implement dark mode and deploy to production.

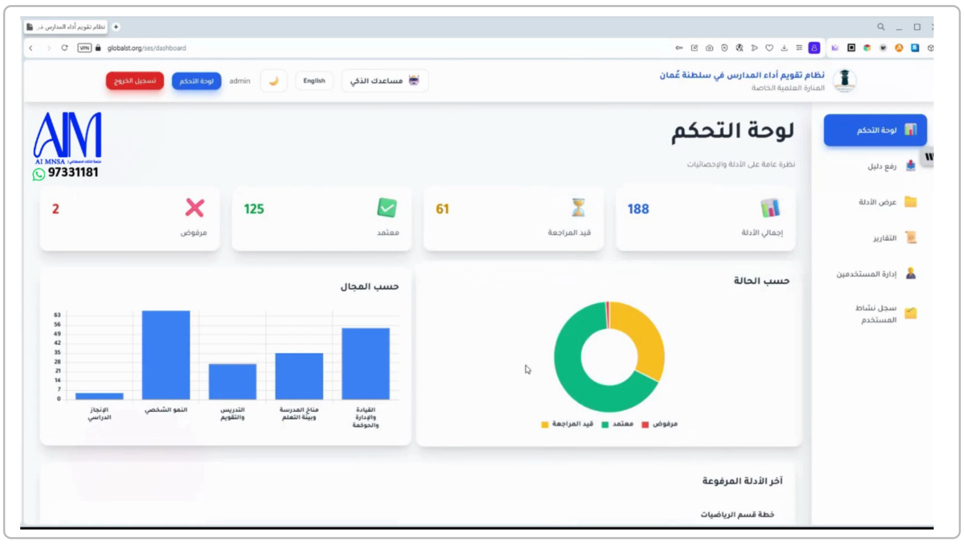
Each step should be tested thoroughly before proceeding. Prioritize the core functionality (upload, review, dashboard) to deliver value early. Keep the colour configuration modular so you can adjust the palette as you iterate.

### Version Control and Branching Strategy

Use **Git** as the version control system and organise work through dedicated **feature branches**. For each feature listed in the roadmap—such as “domain management”, “evidence upload”, “dashboard charts”, etc.—create a separate branch off the main branch (`main` or `develop`). This approach isolates changes, facilitates code reviews and makes it easy to revert or cherry-pick specific features. After a feature is complete and reviewed, merge it back into the main branch via a pull request. Avoid committing unrelated changes to the same branch; clear branch naming (`feature/upload-evidence`, `feature/reporting`) helps maintain clarity.

## Sample Interface

The following screenshot shows a prototype of the dashboard from the live demo. It features counters for evidence status, a bar chart by domain and a doughnut chart for status distribution. Note the restrained colour palette, clear typography and intuitive layout.



By following this guide and collaborating iteratively with your agent, you can build a robust, user-friendly school performance evaluation system that meets the requirements of the Ministry of Education in Oman and provides a scalable foundation for future enhancements.

1 javascript - What's the best way to store page links in next.js? - Stack Overflow
https://stackoverflow.com/questions/77773244/whats-the-best-way-to-store-page-links-in-next-js

2 3 8 Next.js File Uploads: Build A Seamless End-to-End System - Tekody Web Solution
https://tekody.com/next-js-file-uploads-build-a-seamless-end-to-end-system/

4 9 10 11 12 Handling File Uploads in Nextjs Best Practices and Security Considerations | MoldStud
https://moldstud.com/articles/p-handling-file-uploads-in-nextjs-best-practices-and-security-considerations

5 13 How to Optimize File Management in Next.js
https://www.telerik.com/blogs/how-optimize-file-management-next-js

6 7 Effortless File Uploads in Next.js Directly to Public Folder (No Cloud Required) | by Atul Anand | Medium
https://medium.com/@atulj10unofficial/effortless-file-uploads-in-next-js-directly-to-public-folder-no-cloud-required-98dc2d59151d