

DECIDE - Development Manual

Decision Centered Interactive Dynamic Environment



Ruan van Heerden - 41763882

Hendril Willem Mooiman - 41293584

Andre De Goede - 56155093

1. Introduction	4
1.1 Purpose of the manual	4
2. System overview	5
2.1 Key Objectives:	5
2.2 Core Functions:	5
2.3 System requirements	5
3. Architecture and Technology Stack	6
3.1 System Architecture	6
3.2 Technology Stack	6
4. Environment Setup and Installation	7
Prerequisites	7
5. Core Components and Features	8
7.1 Navigation System	8
7.1.1 Sidebar Component (src/Pages/Sidebar.jsx):	8
7.1.2 Layout Component (src/Pages/Layout.jsx):	9
7.1.3 .Application Routes (src/AppRoutes.jsx):	10
7.2 User and Module Management	11
7.2.1 User Management (UsersPage.jsx)	12
7.2.2 Pending User Management (PendingUsers.jsx)	13
7.2.3 Module Management (ModulesPage.jsx)	15
7.2.4 UI/UX Behavior:	16
7.3 Assessment Management	17
7.3.1 Assessments.jsx — Management Dashboard:	18
7.3.2 AssessmentTaking.jsx — Student Assessment Interface:	18
7.3.4 AssessmentViewing.jsx — Student Review Interface:	18
7.3.5 AssessmentReview.jsx — Lecturer Review and Grading Interface:	18
7.3.6 AssessmentGrades.jsx (CompletedAssessments.jsx) — Assessment Grades Overview	18
7.4 Modular Assessment	22
7.4.1 QuestionRenderer.jsx	22
7.4.2 QuestionTypes.jsx:	23
7.4.3 SectionRenderer.jsx:	23
7.3.4 AssessmentRenderer.jsx:	23
6. Firestore Structure	27
6.1 Firebase Configuration	27
6.2 Firebase CRUD Operations	30
7. Assessment Question Types	33
7.1 Forecasting Question	33
7.2 Simulation Question	40
7.3 Decision Theory Question	46
10.4 Inventory Control Question	53
8. Evaluation Logic	59
9. Security and Authentication	59
9.1 Authentication Flow Overview	59

9.2 AuthService Module	60
9.2.1 Firebase Integration	60
9.2.2 Methods and Their Purposes	60
9.3 Login Component	61
9.3.1 Core Functionality	61
9.3.2 Validation Logic	62
9.3.3 Firestore User Synchronisation	62
9.3.4 UI and State Management	62
9.4 Sign-Up Component	63
9.4.1 Validation and Pending User Handling	63
9.4.2 Account Creation with Firebase Auth	63
9.4.3 Firestore Integration and Auto-Login	64
9.4.4 UI and Feedback Handling	64
9.5 Data Synchronisation between Auth and Firestore	64
9.6 Error Handling and Common Issues	64
9.7 Security Considerations	65
9.8 Conclusion	65
10. Support and contact information	66

1. Introduction

DECIDE is an interactive decision support platform developed as part of the **Decision Support Systems (DSS) module**. The system is designed to assist users, particularly students and educators, in understanding, applying, and evaluating multiple decision-making models within a single, unified interface.

The application provides practical, hands-on engagement with several key DSS questions, including **Forecasting, Simulation, Inventory Control** and **Decision Theory** that integrates these approaches. Through this system, users can input data, perform analyses, and view computed results in real time, enabling both learning and applied decision-making within a controlled digital environment.

The DECIDE system was developed using **Electron, React, Tailwind CSS**, and **Firebase**, ensuring that it functions seamlessly as a cross-platform desktop application. Its intuitive design, modular architecture, and clean interface make it suitable for both instructional use in the DSS module and for users seeking to explore multi-criteria decision-making processes independently.

Overall, the DECIDE application transforms complex theoretical models into an accessible, practical tool, bridging the gap between academic understanding and real-world problem-solving.

1.1 Purpose of the manual

The purpose of this Development Manual is to provide clear, step-by-step guidance for users to effectively install, navigate, and operate the DECIDE application. It is intended for future developers continuing with the development of the application, as well as any individual seeking to understand the functionality and application of decision support tools in practice.

Specifically, this manual aims to:

1. Explain the application's core features— guiding users through each module (Forecasting, Simulation, Inventory Control, Decision Theory).
2. Provide setup and access instructions — detailing installation requirements and login procedures.
3. Demonstrate the workflow — from data input and analysis execution to viewing and interpreting results.
4. Outline user roles and permissions — clarifying the differences between Lecturer and Student access levels.
5. Assist with troubleshooting — offering common error resolutions and offline usage tips.

By following this manual, users will gain the necessary understanding to confidently interact with the DECIDE system, perform decision-making analyses, and interpret outcomes as part of their coursework or applied research within the DSS context.

2. System overview

DECIDE is a **desktop application** built using the **Electron framework** with a **React frontend** and **Firebase backend**. It combines the accessibility of a web application with the reliability of a native desktop environment.

2.1 Key Objectives:

- Provide an interactive environment for learning Decision Support Systems concepts.
- Enable real-time synchronization of assessments and results.
- Allow offline access once assessments are downloaded.
- Simplify management of model-based learning tasks for lecturers.

2.2 Core Functions:

- **Lecturer Mode:** Create, edit, and evaluate DSS assessments.
- **Student Mode:** Attempt, submit, and review assessments.
- **View Mode:** Display results, model solutions, and feedback.

2.3 System requirements

Minimum system requirements:

Component	Requirement
Operating System	Windows 10 / macOS 12 / Linux (Ubuntu 20.04+)
Processor	Intel i3 or equivalent
Memory	4 GB RAM minimum
Storage	500 MB free disk space
Internet	Required for initial sign-in and Firebase synchronization

Recommended:

- 8 GB RAM
- 1920x1080 display resolution
- Stable Wi-Fi connection

3. Architecture and Technology Stack

This section will discuss the architecture and technology stacked used for the development of the DECIDE application.

3.1 System Architecture

The application follows a **modular architecture** that separates the frontend interface from backend services and logic layers.

3.2 Technology Stack

Layer	Technology	Description
Frontend	React.js + Tailwind CSS	Interactive UI and state management
Backend	Firebase Firestore	Cloud NoSQL database for assessments and users
Authentication	Firebase Authentication	Role-based login (lecturer/student)
Desktop Framework	Electron	Cross-platform packaging of the web app
Logic	Math.js	Expression parsing and evaluation of formulas
Storage	Firebase Storage	Optional file upload (images, attachments)

4.Environment Setup and Installation

Prerequisites

- Node.js (v18 or newer)
- npm or yarn package manager
- Firebase project credentials
- Electron globally installed

Steps:

Step 1: Clone the repository

Clone the DECIDE project from your version control platform (e.g., GitHub or internal repo):

- Git clone <https://github.com/Henk-Mooiman/DSS-teaching-and-learning-sc.git>

Step 2: Install dependencies

```
npm install
```

This command will automatically install:

- Core dependencies (react, electron, firebase, mathjs, etc.)
- UI libraries (@tailwindcss/*, @headlessui/react, lucide-react, etc.)
- Build and packaging tools (@electron-forge/*, webpack, babel-loader, etc.)

Step 3: Run development environment

```
npm start
```

This command will:

1. Launch the React frontend using Webpack.
2. Open the Electron application window.
3. Connect the interface to Firebase.

If successful, you will see the DECIDE interface load in an Electron desktop window.

Step 4: Build production app

To package the app for Windows, macOS, or Linux, use:

```
npm run make
```

Electron Forge will automatically generate platform-specific installers (e.g., .exe, .deb, .rpm, .zip) in the out/ directory.

5. Core Components and Features

The following section will discuss the core components and features needed to understand how the application functions.

7.1 Navigation System

This section explains the **navigation system**, which includes the **Sidebar**, **Layout**, and **Routing**. Together, these components handle how users navigate between different pages and how access is restricted based on roles (e.g., student, lecturer, admin).

7.1.1 Sidebar Component (src/Pages/Sidebar.jsx):

```
teaching-and-learning-styling-configured > src > Pages > Sidebar.jsx > SidebarNavItem > toggleExpand
1  import React, { useEffect, useState } from "react";
2  import { useNavigate, useLocation } from "react-router-dom";
3  import { signOut } from "firebase/auth";
4  import { auth } from "../Services/firebase";
5  import { firestoreCrud } from "../Services/firebaseCrud";
6  import UserEditModal from "../Components/UserEditModal";
7  import { useUserInfo } from "../Components/useUserInfo";
8  import {
9    Wrench, HomeIcon, BookOpen, BookOpenCheck, User, Logout, ChevronRight, ChevronLeft
10 } from "lucide-react";
11 import logo from "../Assets/NWU-holding-shape-digital-white.png";
12
13 const SidebarNavItem = ({ item, collapsed, isExpanded, setIsExpanded }) => {
14   const navigate = useNavigate();
15   const location = useLocation();
16   const isActive = location.pathname === item.to;
17
18   const toggleExpand = () => {
19     setIsExpanded(prev => ({ ...prev, [item.label]: !prev[item.label] }));
20   };
21
22   const handleClick = () => {
23     if (item.hasSubitems) toggleExpand();
24     else navigate(item.to);
25   };
26 }
```

The Sidebar component provides role-based navigation and user profile access within the app. It dynamically renders menu items according to the user's role (student, lecturer, or admin) and supports collapsing, submenus, and profile editing.

Key Features

- **Role-based menu rendering:** Items are filtered according to the user's role from Firebase.
- **Collapsible navigation:** Uses localStorage to remember the collapsed state.
- **Expandable submenus:** Managed via the isExpanded state object.
- **User profile modal:** Allows viewing and limited editing of user info.
- **Sign-out functionality:** Calls Firebase signOut() and redirects to /login.

Important Files

- Sidebar.jsx
- UserEditModal.jsx (used for profile editing)
- useUserInfo.jsx (fetches user data and role)
- firebaseCrud.js (used to update user information)

Component Breakdown

Element	Function
SidebarNavItem	Handles rendering of individual nav items and submenus.
navItems	Array defining all possible navigation items and their role restrictions.
filteredNavItems	Dynamically generated list filtered by the current user's role.
handleLogout()	Signs the user out of Firebase and navigates to the login page.
User Profile Section	Displays logged-in user details and allows profile editing.

7.1.2 Layout Component (src/Pages/Layout.jsx):

```

teaching-and-learning-styling-configured > src > Pages > Layout.jsx > Layout
1 // src/Pages/Layout.jsx
2 import React, { useState, useEffect } from "react";
3 import { Outlet } from "react-router-dom";
4 import Sidebar from "../Sidebar";
5 import { useUserInfo } from "../Components/useUserInfo";
6
7 export default function Layout() {
8   const { userInfo } = useUserInfo();
9   const [refreshKey, setRefreshKey] = useState(0);
10
11   // Expose a function for Sidebar/UserEditModal to trigger a page reload
12   useEffect(() => {
13     window.forcePageReload = () => setRefreshKey(prev => prev + 1);
14   }, []);
15
16   if (!userInfo) return <div className="p-6">Loading...</div>;
17
18   return (
19     <div className="flex h-screen overflow-hidden">
20       <Sidebar />
21       <main className="flex-1 overflow-y-auto bg-gray-100">
22         { /* The key changes whenever a reload is requested */ }
23         <Outlet key={refreshKey} />
24       </main>
25     </div>
26   );
27 }

```

The Layout component defines the **page structure** shared across all protected routes. It ensures the sidebar stays visible and that the active route's content is displayed within the main view.

Functionality

- Integrates Sidebar for navigation.
- Uses React Router's <Outlet /> to load nested routes dynamically.
- Exposes a global window.forcePageReload function for components (like the sidebar modal) to trigger a refresh.

7.1.3 .Application Routes (src/AppRoutes.jsx):

```
teaching-and-learning-styling-configured > src > Routes.jsx > ...
1  import React, { useEffect, useState } from "react";
2  import { Routes, Route, Navigate } from "react-router-dom";
3  import Login from "../Pages/Login.jsx";
4  import SignUp from "../Pages/SignUp.jsx";
5  import Users from "../Pages/Users.jsx";
6  import PendingUsers from "../Pages/PendingUsers.jsx";
7  import Modules from "../Pages/Modules.jsx";
8  import Home from "../Pages/Home.jsx";
9  import Assessments from "../Pages/Assessments.jsx";
10 import AssessmentTaking from "../Pages/AssessmentTaking.jsx";
11 import Layout from "../Pages/Layout.jsx";
12 import AssessmentEditor from "../Pages/AssessmentEditor.jsx";
13 import PublishedAssessments from "../Pages/PublishedAssessments.jsx";
14 import AssessmentViewing from "../Pages/AssessmentViewing.jsx";
15 import AssessmentGrades from "../Pages/AssessmentGrades.jsx";
16 import AssessmentReview from "../Pages/AssessmentReview.jsx";
17
18 export default function AppRoutes({ isLoggedIn, onLogout }) {
19   return (
20     <Routes>
21       { /* Public Routes */ }
22       <Route path="/login" element={<Login />} />
23       <Route path="/signup" element={<SignUp />} />
24
25       { /* Protected Routes with Sidebar */ }
26       {isLoggedIn ? (
27         <Route element={<Layout />}>
28           <Route path="/" element={<Navigate to="/home" replace />} />
29           <Route path="/home" element={<Home onLogout={onLogout} />} />
30           <Route path="/users" element={<Users />} />
31           <Route path="/pending-users" element={<PendingUsers />} />
32           <Route path="/modules" element={<Modules />} />
33           <Route path="/assessments" element={<Assessments />} />
34           <Route path="/assessment-grades" element={<AssessmentGrades />} />
35           <Route path="/assessment-editor/:id" element={<AssessmentEditor />} />
36           <Route path="/published-assessments" element={<PublishedAssessments />} />
37           <Route path="/assessment-viewing/:id" element={<AssessmentViewing />} />
38           <Route path="/assessment-review/:id" element={<AssessmentReview />} />
39           <Route path="/assessment-taking/:id" element={<AssessmentTaking />} />
40           <Route path="*" element={<Navigate to="/home" replace />} />
41         </Route>
42       ) : (
43         <Route path="*" element={<Navigate to="/login" replace />} />
44       )}
45     </Routes>
46   );
47 }
```

Defines the app's **route structure** and **access control** logic.

Routes are divided into:

- **Public routes:** /login, /signup
- **Protected routes:** Only accessible when isLoggedIn is true

Implementation Notes

- Uses React Router v6.
- The Layout component wraps all protected routes.
- Default redirects:
 - If logged in → /home
 - If not logged in → /login

Developer Notes:

- **Adding a new page:**
 1. Create the page component in src/Pages/.
 2. Add a route in AppRoutes.jsx.
 3. (Optional) Add a new navigation item in Sidebar.jsx under navItems.

- **Changing role access:**
Edit the roles array for the relevant navigation item or subitem.
- **Refreshing sidebar data:**
When a user's details are updated via the modal, the sidebar is refreshed using:

7.2 User and Module Management

```
teaching-and-learning-styling-configured > src > Pages > Users.jsx > UsersPage
1  import React, { useState, useEffect } from "react";
2  import { firestoreCrud } from "../Services/firebaseCrud";
3  import SearchInput from "../Components/SearchInput";
4  import FilterDropdown from "../Components/FilterDropdown";
5  import EntityTable from "../Components/EntityTable";
6  import ConfirmDeleteModal from "../Components/ConfirmDeleteModal";
7  import UserEditModal from "../Components/UserEditModal";
8  import { PencilLine, Trash2 } from "lucide-react";
9  import { useUserInfo } from "../Components/useUserInfo";
10
11 export default function UsersPage() {
12   const [users, setUsers] = useState([]);
13   const [confirmDeleteModal, setConfirmDeleteModal] = useState(null);
14   const [editUser, setEditUser] = useState(null);
15   const [searchQuery, setSearchQuery] = useState("");
16   const [roleFilter, setRoleFilter] = useState("all");
17
18   const { userInfo: currentUser, role: currentUserRole, refetchUserInfo } = useUserInfo();
19
20   useEffect(() => {
21     fetchUsers();
22   }, []);
23
24   const fetchUsers = async () => {
25     const data = await firestoreCrud("User", "read");
26     setUsers(data || []);
27   };
28
29   const canEditUser = (u) => {
30     if (!currentUser) return false;
31     if (currentUserRole === "admin") return true;
32     if (currentUserRole === "lecturer" && u.role === "student") return true;
33     return false;
34   };
35
36   const canDeleteUser = (u) => canEditUser(u);
37
38   const handleDelete = async (id) => {
39     if (!id) return;
40     await firestoreCrud("User", "delete", null, id);
41     setConfirmDeleteModal(null);
42     fetchUsers();
43   };
44 }
```

This section describes the implementation and interaction logic of the User, Pending User, and Module Management systems. These modules enable administrators and lecturers to manage users, pending registrations, and academic modules dynamically through Firestore integration.

The management system provides three main administrative pages:

- **UsersPage:** Manage active users (Admin and Lecturer roles).
- **PendingUsers:** Handle unverified or pre-registered users before activation.
- **ModulesPage:** Manage academic modules, link users, and control role-based access.

All components interact with Firebase via the `firestoreCrud` abstraction layer, ensuring consistent database operations.

Dependencies:

All pages rely on:

- Firebase Firestore for persistence.
- React Hooks (useState, useEffect) for local state and lifecycle handling.
- Custom Components:
 - SearchInput – reusable search field.
 - FilterDropdown – dynamic filtering component.
 - EntityTable – generic tabular renderer with customizable columns.
 - ConfirmDeleteModal, PopupModal – modal components for user confirmation.
 - UserEditModal – inline editing dialog for user updates.
 - useUserInfo – hook returning authenticated user info and role context.

7.2.1 User Management (UsersPage.jsx)

```
const canEditUser = (u) => {
  if (!currentUser) return false;
  if (currentUserRole === "admin") return true;
  if (currentUserRole === "lecturer" && u.role === "student") return true;
  return false;
};

const canDeleteUser = (u) => canEditUser(u);

const handleDelete = async (id) => {
  if (!id) return;
  await firestoreCrud("User", "delete", null, id);
  setConfirmDeleteModal(null);
  fetchUsers();
};

const handleUpdate = async (updatedUser) => {
  if (!updatedUser) return;

  try {
    await firestoreCrud(
      "User",
      "update",
      {
        name: updatedUser.name,
        surname: updatedUser.surname,
        role: updatedUser.role
      },
      updatedUser.id
    );

    fetchUsers();

    if (updatedUser.id === currentUser.id) {
      await refetchUserInfo();
    }

    setEditUser(null);
  } catch (error) {
    console.error("Error updating user:", error);
  }
};
```

Allows administrators and lecturers to view, search, edit, and delete users based on their role privileges.

Key Features

- Role-based access control:
 - Admin: Full CRUD on all users.
 - Lecturer: Can manage only student users.
- Search and Filtering by name, surname, email, or role.
- Real-time Firestore Integration for fetching and updating users.
- Modal Editing for inline user updates.
- Self-Update Detection – updates current user info when self-edited.

Core Functions:

Function	Description
fetchUsers()	Retrieves all users from Firestore.
canEditUser(u)	Determines if the current user has permission to edit user.
handleUpdate(updatedUser)	Updates a user document in Firestore and refreshes data.
handleDelete(id)	Deletes a user document by ID.

7.2.2 Pending User Management (PendingUsers.jsx)

```
teaching-and-learning-styling-configured > src > Pages > PendingUsers.jsx > PendingUsers
1  import React, { useState, useEffect } from "react";
2  import { firestoreCrud } from "../Services/firebaseCrud";
3  import SearchInput from "../Components/SearchInput";
4  import FilterDropdown from "../Components/FilterDropdown";
5  import EntityTable from "../Components/EntityTable";
6  import ConfirmDeleteModal from "../Components/ConfirmDeleteModal";
7  import { useUserInfo } from "../Components/useUserInfo";
8  import { Trash2, Plus } from "lucide-react";
9
10 export default function PendingUsers() {
11   const [pendingUsers, setPendingUsers] = useState([]);
12   const [confirmDeleteModal, setConfirmDeleteModal] = useState(null);
13   const [showAddModal, setShowAddModal] = useState(false);
14   const [bulkStudentNumbers, setBulkStudentNumbers] = useState("");
15   const [selectedRole, setSelectedRole] = useState("student");
16   const [searchQuery, setSearchQuery] = useState("");
17   const [roleFilter, setRoleFilter] = useState("all");
18   const { role: currentUserRole } = useUserInfo();
19   const [popupMessage, setPopupMessage] = useState(null);
20
21   // Fetch pending users from Firestore
22   const fetchPendingUsers = async () => {
23     const data = await firestoreCrud("PendingUser", "read");
24     setPendingUsers(data || []);
25   };
26
27   useEffect(() => {
28     fetchPendingUsers();
29   }, []);
30
31   // Delete a pending user by id
32   const handleDelete = async (id) => {
33     if (!id) return;
34     await firestoreCrud("PendingUser", "delete", null, id);
```

Enables bulk registration and management of users awaiting activation (e.g., students not yet verified).

Features:

- Bulk Upload via text area (one student number per line).
- Duplicate Detection against existing users and pending records.
- Automatic Role Assignment based on current user privileges:
 - Lecturers can only add students.
 - Admins can add any role.
- Popup Feedback System to confirm successful additions or warn of duplicates.
- Filtering and Search by role or student number.

Core Functions:

Function	Description
fetchPendingUsers()	Loads all pending users from Firestore.
handleAddPendingUsers()	Parses bulk input, validates duplicates, creates new pending entries.
handleDelete(id)	Removes a pending user record.
openAddModal() / closeAddModal()	Controls the Add User modal.

7.2.3 Module Management (ModulesPage.jsx)

```
teaching-and-learning-styling-configured > src > Pages > Modules.jsx > ModulesPage
1 import React, { useState, useEffect } from "react";
2 import { firestoreCrud } from "../Services/firebaseCrud";
3 import { db } from "../Services/firebase";
4 import { collection, query, where, getDocs, doc, getDoc } from "firebase/firestore";
5 import { SquarePen, Trash2, Search, ChevronRight, Plus } from "lucide-react";
6 import ConfirmDeleteModal from "../Components/ConfirmDeleteModal";
7 import EntityTable from "../Components/EntityTable";
8 import PopupModal from "../Components/PopupModal";
9 import FilterDropdown from "../Components/FilterDropdown";
10 import { useUserInfo } from "../Components/useUserInfo";
11
12 export default function ModulesPage() {
13   const { user, role, loading } = useUserInfo();
14   const [modules, setModules] = useState([]);
15   const [selectedModule, setSelectedModule] = useState(null);
16   const [showModal, setShowModal] = useState(false);
17   const [confirmDeleteModal, setConfirmDeleteModal] = useState(null);
18   const [expandedModules, setExpandedModules] = useState({});
19   const [searchQuery, setSearchQuery] = useState("");
20   const [popup, setPopup] = useState({ message: "", title: "" });
21   const [roleFilter, setRoleFilter] = useState("all");
22
23   useEffect(() => {
24     if (!loading) fetchModules();
25   }, [loading, role]);
26
27   const fetchModules = async () => {
28     const data = await firestoreCrud("Module", "read");
29
30     const resolved = await Promise.all(
31       data.map(async (mod) => {
32         const resolveUsers = async (refs = []) =>
33           Promise.all(
34             refs.map(async (r) => {
35               const docSnap = await getDoc(r);
36               if (docSnap.exists()) {
37                 const u = docSnap.data();
38                 return {
39                   id: r.id,
40                   studentNumber: u.email.split("@")[0],
41                   name: u.name || "",
42                   surname: u.surname || "",
43                 };
44               }
45             })
46           );
47         return { id: mod.id, studentNumber: mod.id, name: "" };
48       })
49     );
50   }
```

Manages modules, including assigned lecturers and students. Supports role-based visibility and module-level editing.

Role-Based Access:

Role	Permission
Admin	Full access: create, edit, delete modules and manage users.
Lecturer	Can edit assigned modules and add students.
Student	Read-only access to enrolled modules.

Features:

- Dynamic Module Expansion to view linked users.
- Inline User Deletion within modules.
- Bulk User Addition (students/lecturers by ID or email prefix).
- Validation for non-existent users.
- Nested Filtering across module and user levels.
- Popup Modals for invalid entries and confirmation.

Core Functions:

Function	Description
fetchModules()	Loads all modules and resolves linked user references.
getUserRefByNumber(number)	Finds Firestore user reference from ID/email prefix.
saveModule()	Handles creation and updating of module entries.
handleDeleteModule(module)	Deletes an entire module.
handleDeleteUser(module, user)	Removes a specific user from a module.

7.2.4 UI/UX Behavior:**Modals:**

All CRUD actions use confirmation or popup modals to prevent accidental deletions or inform users of success/failure.

Filtering:

Filtering options (role, search input) dynamically update rendered tables without a page reload.

Responsiveness:

All pages are responsive, using Tailwind utility classes to adjust layouts for mobile and desktop.

Error Handling:

- All Firestore operations are wrapped in try/catch blocks with console logging.
- Failed CRUD actions trigger popups when applicable.
- Invalid or missing user data is gracefully skipped with notifications.

Firestore Schema Summary:**Collections:**

Collection	Description
User	Stores all registered users with roles and basic info.
PendingUser	Temporary store for unapproved or pre-registered users.
Module	Contains module metadata and references

	to users.
--	-----------

Reference Fields:

- **Module.studentID** → Array of Firestore User references.
- **Module.lecturerID** → Array of Firestore User references.

Access Control Summary:

Operation	Admin	Lecturer	Student
View users	Yes	Yes	No
Edit user	Yes	Students only	No
Delete user	Yes	Students only	No
Manage modules	Yes	Assigned only	View only
Add pending users	Yes0	Students only	No

Maintenance Notes:

- Ensure Firestore security rules enforce identical restrictions at the backend.
- Keep UI components (EntityTable, FilterDropdown, ConfirmDeleteModal) modular for reuse in future pages.
- For bulk imports, validate data formatting before submission to avoid Firestore write quota overuse.

7.3 Assessment Management

The Assessment Management System manages the complete lifecycle of assessments, from creation and delivery to submission, grading, and review. It supports online and offline modes, ensuring students can complete assessments even with unstable connectivity.

The module now includes **five primary components**:

1. **Assessments.jsx** – Management dashboard for assessments and submissions.
2. **AssessmentTaking.jsx** – Interface for students to take assessments.
3. **AssessmentViewing.jsx** – Read-only interface for viewing submitted assessments.
4. **AssessmentReview.jsx** – Lecturer grading and review interface.
5. **AssessmentGrades.jsx (CompletedAssessments.jsx)** – Unified grade overview for students, lecturers, and admins.

7.3.1 Assessments.jsx — Management Dashboard:

Key functions:

- Recursive deletion of assessments and related submissions.
- Real-time Firestore synchronization.
- Publish/unpublish functionality.
- Nested display of submissions with live updates.
- Access control via role (lecturer/admin).

7.3.2 AssessmentTaking.jsx — Student Assessment Interface:

Enhancements:

- Offline-first approach: Saves progress in both Firestore and AES-encrypted local storage.
- Auto-resume: Restores unfinished work after disconnection.
- Auto-submit on timeout.

7.3.4 AssessmentViewing.jsx — Student Review Interface:

Displays a previously submitted assessment in read-only mode. It fetches the submission and its corresponding assessment metadata, then renders all sections and questions using the shared AssessmentRenderer.

7.3.5 AssessmentReview.jsx — Lecturer Review and Grading Interface:

Allows lecturers to review, grade, and update student submissions.

Includes:

- Dynamic score computation per section/question.
- Editable marks and comments (depending on renderer support).
- Direct Firestore updates to Submissions subcollection.

7.3.6 AssessmentGrades.jsx (CompletedAssessments.jsx) — Assessment Grades Overview

Provides an aggregated view of completed assessments and grades for all users.

- Students see their own submitted assessments with scores.
- Lecturers and admins can see all submissions, filter by module, and review individual performances.

Key Features

- Role-based data filtering:
 - *Students* see only their submissions.
 - *Lecturers* see submissions from their modules.

- *Admins* see all submissions.
- Dynamic score computation: Automatically calculates total and max scores if missing.
- Search and filtering:
 - Search by assessment title or student name.
 - Filter by module using dropdown.
- Data mapping:
 - usersMap: Resolves student IDs to names.
 - modulesMap: Resolves module IDs to names.
- Navigational integration: Clicking “View” routes to the detailed AssessmentViewing.jsx page.

Core Functions

- User & Module Fetching: Retrieves users and modules to determine access scope and populate mapping tables.
- Submissions Aggregation: Iterates through all assessments, fetching each subcollection of Submissions. Enriches them with derived fields (moduleName, totalScore, maxScore, etc.).
- Score Calculation:
- Filtering Logic: Matches searchQuery and selectedModule against titles, student names, and module IDs.

UI Components

- EntityTable – Displays tabular results for all submissions.
- SearchInput – Provides live search for filtering by name or title.
- FilterDropdown – Filters results by module.
- View button – Opens the selected submission in AssessmentViewing.jsx.

Data Display:

Column	Description
Assessment Title	Assessment name
Module	Module name
Student	(Only for lecturer/admin) Student name
Score	Displays totalScore / maxScore
Submitted At	Local timestamp of submission
Actions	“View” button linking to submission details

Example Firestore Read Structure

Assessments/{assessmentId}/Submissions/{submissionId}

- └─ studentId: string
- └─ totalScore: number
- └─ maxScore: number
- └─ sections: [...]
- └─ submittedAt: Timestamp
- └─ moduleId: string
- └─ title: string

Permissions:

Role	Access Scope
Admin	All modules and submissions
Lecturer	Module assigned to them
Student	Only their submissions

Firestore Data Model *(Extended with grading attributes)*

Assessments (Collection)

- └─ assessmentId (Document)
 - └─ title: string
 - └─ moduleId: string
 - └─ durationMinutes: number
 - └─ status: "draft" | "published"
 - └─ createdAt / updatedAt: Timestamp
 - └─ sections: [{ id, title, questions: [...] }]
 - └─ Submissions (Subcollection)
 - └─ submissionId (Document)
 - └─ studentId: string
 - └─ totalScore: number

```
| | └─ maxScore: number
```

```
| | └─ sections: [ ... ]
```

```
| | └─ status: "submitted" | "graded"
```

```
| | └─ submittedAt: Timestamp
```

```
| | └─ feedback?: string
```

```
| | └─ gradePercentage?: number
```

Component Interactions:

Component	Purpose	Connected Components
Assessments.jsx	Management dashboard	ConfirmDeleteModal, EntityTable
AssessmentTaking.jsx	Student interface for taking tests	AssessmentRenderer, ConfirmModal
AssessmentViewing.jsx	View-only mode for submissions	AssessmentRenderer
AssessmentReview.jsx	Lecturer review and grading	AssessmentRenderer, firestoreCrud
AssessmentGrades.jsx	Completed/graded results overview	SearchInput, FilterDropdown, EntityTable

The Assessment Management and Grading System provides an end-to-end assessment solution for students, lecturers, and administrators. The inclusion of `AssessmentGrades.jsx` enables clear visibility into performance metrics, supporting data-driven academic evaluation and transparent grade reporting.

Key strengths:

- Real-time Firestore integration.
- Offline and encrypted draft handling.
- Modular and role-based access.
- Unified view of assessment creation, completion, and grading.

Would you like me to extend this with a Firestore query reference section (showing typical CRUD and filtering queries used by each component)? That can make it easier for future developers to modify or integrate analytics later.

7.4 Modular Assessment

The Modular Assessment Backend provides a flexible framework for managing assessments consisting of multiple sections and questions.

It supports:

- Dynamic question types via QuestionTypes registry.
- Role- and mode-based rendering: edit, take, view, review.
- Real-time scoring and updates for assessments, sections, and questions.
- Add/Delete/Update operations at the question and section levels.

This system is highly reusable for students, lecturers, and administrators.

7.4.1 QuestionRenderer.jsx

Renders a single question dynamically based on its type. Handles score display, point editing, and user responses depending on the mode.

Props:

Prop	Type	Description
question	object	Question object containing id, type, and data.
mode	string	Rendering mode: "edit"
onUpdate	function	Callback invoked when the question is updated.
onDelete	function	Callback invoked when the question is deleted. Only relevant in edit mode.
showPoints	boolean	Whether to display points/score. Default: false.

Features

- Dynamic Component Resolution: Uses questionTypes to find the correct component for the question type.
- Score Display:
 - edit: points editable.
 - review: score editable for grading.
 - take/view: display user score vs max points.
- Delete Functionality: Only shown in edit mode.

7.4.2 QuestionTypes.jsx:

Registry of all available question types for dynamic rendering.

Implementation Notes

- Automatically imports all JSX files in ./Questions.
- Enables plug-and-play addition of new question types.

7.4.3 SectionRenderer.jsx:

Manages a section of an assessment, containing multiple questions.

Props:

Prop	Type	Description
Section	object	Section object with id, title, and questions.
mode	string	""edit"
onUpdateSection	function	Callback when the section is updated (questions added/edited).
onDeleteSection	function	Callback to delete the entire section.

Features

- Add Question: Users can select a question type from a dropdown.
- Edit Section Title: Inline editable title in edit mode.
- Update Question: Propagates updates from QuestionRenderer.
- Delete Question / Section: Controlled via confirmation modal

7.3.4 AssessmentRenderer.jsx:

Renders the entire assessment, composed of multiple sections and questions.

Props:

Props	Type	Description
Assessment	object	Assessment object with sections and metadata.
mode	string	""edit"

onUpdateSection	Function	Callback when a section is updated.
onUpdateQuestion	Function	Callback when a question is updated.
onAddSection	Function	Callback to add a new section.
onDeleteSection	Function	Callback to delete a section.

Features:

- Score Calculation: Computes totalMaxScore and totalUserScore dynamically.
- Section Mapping: Iterates through sections and renders using SectionRenderer or direct question display depending on mode.
- Add Section Button: Only shown in edit mode.
- Flexible Modes:
 - edit: Full editing, adding/deleting questions and sections.
 - take: Student interface for taking assessments.
 - view: Read-only display for students or lecturers.
 - review: Lecturer grading view.

Score Computation:

Question Level: `pointsDisplay = `${question.data.userScore ?? 0} / ${question.data.points ?? 1}``

Section Level: `sectionScore = section.questions.reduce((sum, q) => sum + (q.data?.userScore ?? 0), 0)`

Assessment Level:

```
totalMaxScore = assessment.sections.reduce(
  (acc, section) => acc + section.questions.reduce(
    (sum, q) => sum + (q.data?.points ?? 1), 0
  ),
  0
)
```

Firestore / Backend Model

Assessments (Collection)

```
└── assessmentId (Document)
    ├── title: string
    ├── sections: [
    |   | {
    |   |   id: string,
    |   |   title: string,
    |   |   questions: [
    |   |   | {
    |   |   |   id: string,
    |   |   |   type: string,
    |   |   |   data: {
    |   |   |   |   prompt: string,
    |   |   |   |   correctAnswer?: string,
    |   |   |   |   userAnswer?: string,
    |   |   |   |   points: number,
    |   |   |   |   userScore?: number
    |   |   |   }
    |   |   | }
    |   |   ]
    |   | }
    |   ]
    ├── totalMaxScore?: number
    └── totalUserScore?: number
```

Notes:

- id fields are always unique using `crypto.randomUUID()`.
- Questions may store `userAnswer` for student submissions and `userScore` for grading.
- The modular design allows adding/removing sections and questions dynamically without schema changes.

Component Interactions:

Component	Child Components	Mode
AssessmentRenderer	SectionRenderer	edit, take, view, review
SectionRenderer	QuestionRenderer	edit, take, view, review
QuestionRenderer	Specific QuestionType component	all modes
ConfirmDeleteModal	-	edit mode

The Modular Assessment Backend provides:

- Dynamic question rendering with modular question types.
- Nested section-question architecture supporting CRUD operations.
- Role- and mode-based rendering for students, lecturers, and admins.
- Automatic score computation across questions, sections, and entire assessments.
- Firestore-compatible structure for persistence, enabling scalable assessment management.

This backend forms the foundation for the frontend assessment components (AssessmentTaking, AssessmentReview, AssessmentGrades) and ensures consistent grading and reporting across the system.

6. Firestore Structure

The following section will discuss the Firebase firestore setup as well as the Firebase CRUD operations.

6.1 Firebase Configuration

The Firebase configuration file initializes the connection between the DECIDE application and its Firebase backend services, including Authentication and Cloud Firestore.

This setup enables secure user management, real-time data synchronization, and seamless storage of all assessment-related data (e.g., users, roles, questions, and results).

Source Code:

```
import { initializeApp } from 'firebase/app';
import { getAuth } from 'firebase/auth';
import { getFirestore } from 'firebase/firestore';

const firebaseConfig = {
  apiKey: 'AIzaSyAjmiMglC5GGQ7T4rbPJVGpQWg-Kzh6rNA',
  authDomain: 'dss-teaching-and-learning.firebaseio.com',
  projectId: 'dss-teaching-and-learning',
  storageBucket: 'dss-teaching-and-learning.firebaseio.com',
  messagingSenderId: '880289327130',
  appId: '1:880289327130:web:d40bb4f4e32d91ea1f89a3',
  measurementId: 'G-6XQF9Y3B6H',
};

const app = initializeApp(firebaseConfig);
const auth = getAuth(app);
const db = getFirestore(app);

export { auth, db };
```

Importing Firebase Modules:

```
import { initializeApp } from 'firebase/app';
import { getAuth } from 'firebase/auth';
import { getFirestore } from 'firebase/firestore';
```

These imports bring in Firebase's modular SDK functions:

- initializeApp – Starts the Firebase app using your project's credentials.
- getAuth – Enables Firebase Authentication for user login and registration.
- getFirestore – Provides access to Cloud Firestore, Firebase's NoSQL database.

This modular import approach (v9+) ensures better performance and smaller bundle sizes.

Firebase Configuration Object:

```
const firebaseConfig = {
  apiKey: 'AIzaSyAjmiMglC5GGQ7T4rbPJVGpQWg-Kzh6rNA',
  authDomain: 'dss-teaching-and-learning.firebaseio.com',
  projectId: 'dss-teaching-and-learning',
  storageBucket: 'dss-teaching-and-learning.firebaseio.com',
  messagingSenderId: '880289327130',
  appId: '1:880289327130:web:d40bb4f4e32d91ea1f89a3',
  measurementId: 'G-6XQF9Y3B6H',
};
```

This object contains unique credentials generated by Firebase for your specific project:

Key	Description
apiKey	Identifies your Firebase project for authentication and API access.
authDomain	URL used for authentication redirects (email, Google sign-in, etc.).
projectId	Unique identifier for the Firebase project.
storageBucket	Cloud Storage location for uploaded files (if used).
messagingSenderId	Used for push notifications and Firebase Cloud Messaging (if enabled).
appId	Unique app identifier linking Firebase services to your app.
measurementId	Used for Google Analytics integration (optional).

Each key is automatically generated when you create a Firebase project, never hardcode these for production in public repositories; use environment variables instead (e.g., .env.local).

Initialising the Firebase App:

```
const app = initializeApp(firebaseConfig);
```

This command boots up Firebase using the configuration object. It creates a singleton instance (app) that is shared across the entire application. All Firebase services (e.g., Firestore, Authentication) rely on this initialized app to function.

Setting up authentication:

```
const auth = getAuth(app);
```

- Initializes the Firebase Authentication service for the app.
- Manages login, registration, and user session states.
- Used throughout the application to:
 - Create accounts (createUserWithEmailAndPassword)
 - Sign in (signInWithEmailAndPassword)
 - Track user roles and permissions (stored in Firestore's User collection).

Setting up firestore:

```
const db = getFirestore(app);
```

- Initializes a connection to Cloud Firestore.
- Provides CRUD access to all collections such as:
 - Users → user data and roles
 - Questions → assessment questions and configurations
 - Results → student performance data
 - Simulations / Forecasting → stored learning module data

Exporting the instance:

```
export { auth, db };
```

This line exports the **Authentication** and **Firestore** instances so they can be imported and reused across your entire React application. This ensures consistent access to the same Firebase app configuration throughout all components (e.g., Login, Question Management, Results Dashboard).

6.2 Firebase CRUD Operations

The `firebaseCrud.js` service provides a centralized, reusable abstraction layer for all database operations in the DSS Dynamos Teaching and Learning Application. It simplifies interaction with Firebase's Cloud Firestore by wrapping common CRUD (Create, Read, Update, Delete) operations into a single, consistent function, `firebaseCrud()`.

This design ensures:

- Clean and maintainable code.
- Reduced duplication across components.
- Automatic timestamp management.
- Basic offline detection and error handling.

Imports and initialisation:

```
import {  
  collection, addDoc, getDocs, doc,  
  updateDoc, deleteDoc, getDoc, setDoc, Timestamp  
} from "firebase/firestore";  
import { db } from "../firebase";
```

These imports bring in Firestore's modular functions for each database operation, and the shared `db` instance from your Firebase configuration file ([firebase.js](#)).

Universal function:

```
export const firebaseCrud = async (collectionName, operation, data = null,  
  const colRef = collection(db, collectionName);
```

Parameter	Description
collectionName	The Firestore collection to operate on (e.g., "User", "Assessments").
operation	The CRUD operation to perform ("create", "read", "update", "delete").
data	The object containing data to be written or updated. Optional for read/delete.
docId	The ID of the specific document (if known). Optional for creation or full reads.

All Firestore operations are wrapped in a single asynchronous function, reducing redundancy throughout the app.

Create operation:

```
case "create": {
  const dataWithTimestamp = {
    ...data,
    createdAt: Timestamp.now(),
    updatedAt: Timestamp.now(),
  };
  if (docId) return await setDoc(doc(db, collectionName, docId), dataWithTimestamp);
  return await addDoc(colRef, dataWithTimestamp);
}
```

Creates a new document inside the specified collection.

- addDoc() → Adds a document with an auto-generated ID.
- setDoc() → Writes to a specific docId if provided.
- Automatically includes timestamps (createdAt and updatedAt).

Read operation:

```
case "read": {
  if (docId) {
    const docSnap = await getDoc(doc(db, collectionName, docId));
    return docSnap.exists() ? { id: docSnap.id, ...docSnap.data() } : null;
  } else {
    const querySnapshot = await getDocs(colRef);
    return querySnapshot.docs.map((d) => ({ id: d.id, ...d.data() }));
  }
}
```

Retrieves data from Firestore.

- Single Document: When docId is provided, returns that document.
- All Documents: Without docId, it fetches the entire collection.
- Each returned object includes its unique document ID.

Update operation:

```
case "update": {
  if (!docId) throw new Error("docId required for update");
  return await updateDoc(doc(db, collectionName, docId), {
    ...data,
    updatedAt: Timestamp.now(),
  });
}
```

Updates an existing document identified by docId.

- updateDoc() only modifies provided fields.
- Automatically refreshes the updatedAt timestamp.

Delete operation:

```
case "delete": {
  if (!docId) throw new Error("docId required for delete");
  return await deleteDoc(doc(db, collectionName, docId));
}
```

Deletes a document by its docId.

Error handling and offline detection:

```
catch (err) {
  if (err.message?.includes("net::ERR_NAME_NOT_RESOLVED")) {
    console.warn("Offline detected via network error:", err.message);
    window.dispatchEvent(new CustomEvent("appOffline", { detail: err }));
  }
  throw err;
}
```

This block:

- Detects when the application is offline (ERR_NAME_NOT_RESOLVED).
- Dispatches a custom event (appOffline) to notify the UI.
- Rethrows other unexpected errors for debugging or logging.

Best Practices:

1. Centralized Access: All Firestore operations across the app should use this single function for consistency.
2. Error Resilience: Wrap all calls in try...catch to handle offline scenarios or permission issues.
3. Timestamping: Automatic createdAt and updatedAt fields simplify sorting and change tracking.
4. Role-Based Access: Ensure Firestore Security Rules enforce permissions — e.g., students can only read their results, lecturers can write assessments.

7. Assessment Question Types

The section will discuss the existing questions within the application which include forecasting, simulation, inventory control, and decision theory.

7.1 Forecasting Question

The Forecasting Question component manages an interactive question type where users (students) complete forecasting tables based on randomly generated variables. Lecturers define formulas and parameters, while students enter calculated results. The system then evaluates their responses against correct answers derived dynamically from mathematical expressions.

Core Structure and Data Model:

Each forecasting question is defined by a structured data object containing:

- matrix: The main forecasting table (array of rows, each containing multiple cells).
- variables: A list of random variables (name, type, range, decimals).
- randomVars: A generated set of variable values per student attempt.
- alpha and beta: Optional special variables.
- userAnswers: Student-provided responses.
- subQuestions: Optional smaller questions linked to the main table.
- headers: Column labels (e.g., "Text", "Variable", "Equation").
- tolerance: Acceptable error margin for numeric comparisons.

```
const defaultData = {
  headers: ["Description", "Variable", "Equation", "Result"],
  matrix: [],
  variables: [],
  randomVars: {},
  alpha: { enabled: false },
  beta: { enabled: false },
  userAnswers: [],
  subQuestions: [],
  tolerance: 0.05,
};
```

This structure allows both lecturers and students to interact with the same data model but in different modes (edit, take, view).

Helper functions:

These small utilities keep data consistent and prevent invalid numerical input or unsafe substitutions.

```
// Normalize numbers with "." or "," and fallback
function normalizeNumber(input, fallback = NaN) {
  if (input == null) return fallback;
  const normalized = input.toString().trim().replace(",", ".");
  return isNaN(normalized) ? fallback : parseFloat(normalized);
}

// Escape curly braces and special characters for safe RegExp use
function escapeForRegex(s) {
  return s.replace(/[\.*+?^$()\[\]\]/g, "\\$&");
}

// Replace placeholders like {x1} with actual random variable values
function substituteText(text, randoms) {
  if (!text) return text;
  Object.entries(randoms).forEach(([key, val]) => {
    text = text.replace(new RegExp(`\\{${escapeForRegex(key)}\\}`, "gi"), val);
  });
  return text;
}
```

normalizeNumber(input, fallback)

Ensures numerical consistency by:

- Converting commas to dots in numbers (1,5 → 1.5).
- Returning a valid float or a fallback value if invalid. Used extensively to compare numerical answers with precision.

escapeForRegex(s)

Escapes special characters to safely construct regular expressions used in text replacement operations.

substituteText(text, randoms)

Replaces placeholders (e.g., {x1}, {alpha}) with their actual randomized numeric values during evaluation and question rendering.

Data management and state updates:

React hooks manage the question's current state (matrix, headers, variables, etc.). When the lecturer or student edits data, it updates through `onUpdate()`, syncing with the parent component.

```
const [headers, setHeaders] = useState(data.headers || []);
const [takeMatrix, setTakeMatrix] = useState(data.matrix || []);
const [errors, setErrors] = useState([]);

function handleMatrixChange(newMatrix) {
  setTakeMatrix(newMatrix);
  onUpdate({ ...data, matrix: newMatrix });
}
```

The component maintains internal state for:

- headers: Column names of the forecasting table.
- takeMatrix: The student's working version of the table.
- errors: Tracks invalid formulas in lecturer edit mode.

All state changes are propagated through the `onUpdate()` callback, which updates the parent's `question.data` object — ensuring reactivity and persistence in Firebase or local state.

Variable detection and management:

Lecturers define variables within curly braces (e.g., `{x1}`), which the system auto-detects and stores in variables.

```
function extractVariablesFromMatrix(matrix) {
  const vars = new Set();
  matrix.forEach(row => {
    row.cells.forEach(cell => {
      const matches = cell.value.match(/\{(.*)\}/g);
      if (matches) {
        matches.forEach(m => vars.add(m.replace(/[{}]/g, "")));
      }
    });
  });
  return Array.from(vars);
}
```

When lecturers edit the question (`mode === "edit"`), the component automatically:

- Scans the second column for text patterns like `{x1}`, `{demand}`, etc.
- Extracts these as variable names.

- Adds or removes them dynamically from the variables array. Each variable stores metadata:

```
{
  name: "x1",
  type: "int" | "float",
  min: 0,
  max: 10,
  decimals: 2
}
```

This ensures that when randomization occurs, variable values are generated within the defined range.

Random variable generation:

For each student attempt, the app generates random variable values according to type and range.

```
function generateRandomVars(vars) {
  const randoms = {};
  vars.forEach(v => {
    if (v.type === "int") {
      randoms[v.name] = Math.floor(Math.random() * (v.max - v.min + 1)) + v.min;
    } else {
      const val = Math.random() * (v.max - v.min) + v.min;
      randoms[v.name] = parseFloat(val.toFixed(v.decimals));
    }
  });
  return randoms;
}
```

When the question is in take or view mode:

- If randomVars is not already generated, it creates a set of random values for each defined variable.
- Special variables (alpha, beta) are generated separately if enabled.
- The randomization ensures each student receives a unique variation of the question.

Matrix handling logic:

The forecasting matrix is an array of rows, each containing multiple cell objects.

```
{
  row: 0,
  cells: [
    { col: 0, value: "Forecasted demand" },
    { col: 1, value: "{x1}" },
    { col: 2, value: "{x1} * 1.5 + 3" }
  ]
}
```

The forecasting table is a flexible matrix that supports:

- Adding/removing rows and columns.
- Storing each cell's text or formula value.
- Evaluating formulas in real time for correctness (in lecturer mode).

In lecturer view, the formula in column 3 ($\{x1\} * 2 + 5$) is evaluated using math.js, substituting $\{x1\}$ with its random value.

In student mode, this column is left blank for the student to fill with their computed result.

Sub question management:

Sub-questions allow additional scoring items tied to the same variable set.

```
{
  prompt: "Calculate total cost using {alpha} and {x1}",
  formula: "{x1} * {alpha}",
  correctAnswer: "",
  userAnswer: "7.5",
  points: 2
}
```

Each sub-question can:

- Use text with placeholders ($\{alpha\}$, $\{x2\}$, etc.).
- Include an optional formula for automatic evaluation.
- Have a manual fixed correct answer if no formula is given.
- Carry custom point values.

Evaluation Logic:

The main evaluation system computes the student's score based on formulas and tolerance.

```
function getMaxScore(q) {
  const tableScore = q.data.matrix.reduce(
    (acc, row) =>
      acc + row.cells.filter(c => c.value && c.value.includes("{")).length,
    0
  );
  const subScore = q.data.subQuestions.reduce(
    (acc, sq) => acc + (sq.points || 1),
    0
  );
  return tableScore + subScore;
}
```

getMaxScore(q)

Calculates total possible marks by:

- Counting all valid equations in the lecturer's matrix.
- Adding all points values from sub-questions.

evaluate(q)

Computes the student's total score

```
import { evaluate as mathEval } from "mathjs";

function evaluate(q) {
  let userScore = 0;
  const tolerance = q.data.tolerance || 0.05;
  const randVars = q.data.randomVars || {};
  const varMap = { ...randVars };

  // Case-insensitive Alpha/Beta variables
  if (q.data.alpha?.enabled) {
    varMap.alpha = varMap.Alpha = varMap.ALPHA = q.data.randomVars.alpha;
  }
  if (q.data.beta?.enabled) {
    varMap.beta = varMap.Beta = varMap.BETA = q.data.randomVars.beta;
  }
}
```

1. Matrix Evaluation:

```
// Evaluate forecasting matrix
q.data.matrix.forEach((row, i) => {
  const equationCell = row.cells[2];
  const answer = q.data.userAnswers?.[i];
  if (equationCell?.value && answer !== null) {
    try {
      const correct = mathEval(substituteText(equationCell.value, varMap));
      const userVal = normalizeNumber(answer);
      if (Math.abs(userVal - correct) <= tolerance) userScore++;
    } catch (err) {
      console.error("Eval error:", err);
    }
  }
});
```

- For each cell with a valid formula, evaluate the expression using math.js and the randomized variables.
- Compare with the corresponding student answer.
- If the difference $\leq$ tolerance, award 1 point.

2. Sub-Question Evaluation:

```
// Evaluate sub-questions
q.data.subQuestions.forEach(sq => {
  let correctVal = sq.correctAnswer;
  if (sq.formula) {
    try {
      correctVal = mathEval(substituteText(sq.formula, varMap));
    } catch (err) {
      console.error("SubQ eval error:", err);
    }
  }
  const userVal = normalizeNumber(sq.userAnswer);
  if (Math.abs(userVal - correctVal) <= tolerance) userScore += (sq.points || 1);
});

return { userScore, maxScore: getMaxScore(q) };
```

- Evaluate each sub-question's formula (if present) using the same variable set.
- Compare the computed value or fixed correct answer with the student's input.
- Award full points if within tolerance or exact string match.

Integration and extensibility:

The forecasting question module exports an object:

```
export default {  
  type: "forecasting",  
  label: "Forecasting Question",  
  defaultData: { ... },  
  Component: ForecastingQuestion,  
  getMaxScore,  
  evaluate  
};
```

This design allows the DSS Dynamos application to:

- Register it as one of several question types.
- Dynamically render it in different modes (edit, take, view).
- Compute scores in a consistent format across all question types.

Key External Dependencies:

- mathjs: Safely evaluates mathematical expressions and formulas.
- React State Hooks (useState, useEffect): Handle dynamic table updates and mode-specific behavior.
- InfoTooltip Component: Provides contextual hints (not core to logic but improves user understanding).

7.2 Simulation Question

The SimulationQuestion component is responsible for generating, displaying, and evaluating simulation-based problems dynamically.

It allows lecturers to define simulation tables (with probability or calculation data), and students to interact with them by filling in answers or viewing randomised simulations.

Variable handling and randomization:

This section ensures that simulation variables are dynamically generated:

- generateRandomValue creates random numbers (integer or decimal) based on type, range, and precision.

```
// Generate random values for variables
const generateRandomValue = (config) => {
  if (!config) return "";
  const { type, min = 0, max = 10, decimals = 2 } = config;
  if (type === "int" || type === "integer") {
    return Math.floor(Math.random() * (max - min + 1)) + min;
  }
  if (type === "float" || type === "decimal") {
    const factor = Math.pow(10, decimals || 2);
    return Math.trunc((Math.random() * (max - min) + min) * factor) / factor;
  }
  return "";
};
```

- generateProbabilities creates a set of random probabilities that always sum to 1, used in probability-based simulation tables.

```
// Generate probabilities that sum to 1
const generateProbabilities = (count, decimals = 3) => {
  let values = Array.from({ length: count }, () => Math.random());
  const sum = values.reduce((a, b) => a + b, 0);
  values = values.map(v => v / sum);
  const factor = Math.pow(10, decimals);
  values = values.map(v => Math.round(v * factor) / factor);
  // Fix rounding error to ensure total = 1
  const diff = 1 - values.reduce((a, b) => a + b, 0);
  values[values.length - 1] += diff;
  return values;
};
```

These functions form the foundation of stochastic simulation, giving each student a unique dataset for their exercise.

Table normalization and structure setup:

```
const normalizeTables = (tables) => {
  if (!Array.isArray(tables)) return [];
  return tables.map((t) => {
    if (t.headers && Array.isArray(t.matrix)) return t;
    const cols = t.columns || t.headers || ["Column 1"];
    const rows = t.rows || [];
    const matrix = rows.map((r, ridx) => ({
      row: ridx,
      cells: cols.map((_, cidx) => ({ col: cidx, value: r[cidx] ?? "" })),
    }));
    return { headers: cols, matrix, type: t.type || "calculation" };
  });
};
```

The normalization function guarantees that all simulation tables follow a consistent structure:

- Each table has a headers array (column names) and a matrix (rows and cells).
- This ensures compatibility with rendering and updating logic later in the component.

Essentially, this acts as a data sanitizer, ensuring that old or inconsistent table formats don't break the app.

Probability range and calculation:

```
const calculateProbabilityRanges = (probabilities, rangeMax = 100) => {
  let cumulative = 0;
  const ranges = [];
  probabilities.forEach((prob, index) => {
    const start = Math.round(cumulative * rangeMax) + 1;
    cumulative += prob;
    const end = index === probabilities.length - 1 ? rangeMax : Math.round(cumulative * rangeMax);
    ranges.push(`${start}-${end}`);
  });
  return ranges;
};
```

This function converts a set of probabilities (like [0.3, 0.5, 0.2]) into cumulative range values (e.g. "1–30", "31–80", "81–100").

These ranges represent random number intervals often used in simulation models (e.g., Monte Carlo simulation).

Variable detection and configuration:

```
useEffect(() => {
  if (mode !== "edit") return;
  const allCellsText = getTables().flatMap((t) => t.matrix.flatMap((r) => r.cells.map((c) => c.value)));
  const combined = allCellsText.join(" ");
  const detectedVars = Array.from(combined.matchAll(/\{([xp]\d+)\}/g)).map((m) => "{ " + m[1] + " }");

  const newVars = detectedVars.map((v) => ({
    name: v,
    type: v.includes("p") ? "probability" : "int",
    min: 0,
    max: v.includes("p") ? 1 : 10,
    decimals: v.includes("p") ? 3 : 2
  }));
  handleChange("variables", newVars);
}, [data.tables, mode]);
```

This section automatically detects variables written in the {x1} or {p1} format in any table or formula.

When a lecturer adds such placeholders, the system:

- Scans the tables and text.
- Registers the new variable.
- Assigns a default type and range based on its name (e.g. p = probability).

This makes the editor intelligent, lecturers don't need to manually define variables.

Random variable substitution:

```
useEffect(() => {
  if (!(mode === "take" || mode === "view")) return;
  let randVars = data.randomVars || null;
  if (!randVars) {
    randVars = {};
    const vars = data.variables || [];
    // Generate random and probability variables
    vars.forEach(v => {
      if (v.type === "probability") return;
      randVars[v.name] = generateRandomValue(v);
    });
    const probVars = vars.filter(v => v.type === "probability");
    if (probVars.length) {
      const probValues = generateProbabilities(probVars.length);
      probVars.forEach((v, i) => randVars[v.name] = probValues[i]);
    }
    handleChange("randomVars", randVars);
  }
});
```

When a student takes or views a question:

- The system generates a unique set of random variable values.
- These values are stored in randomVars and substituted wherever {x1}, {p1}, etc., appear.

This gives each student a unique version of the simulation, preventing answer sharing and encouraging independent calculation.

Table rendering and formula evaluation:

```
const renderMatrix = (matrixData, tableIdx) => {
  const randVars = data.randomVars || {};
  const varMap = {};
  Object.keys(randVars).forEach((k) => {
    varMap[k.replace(/[{}]/g, "")] = randVars[k];
  });

  return matrixData.map((r) => ({
    ...r,
    cells: r.cells.map((c) => {
      let value = c.value;
      Object.keys(randVars).forEach((k) => {
        value = value.replace(new RegExp(escapeForRegex(k), "g"), randVars[k]);
      });
      try {
        const result = evaluate(value.replace(/[{}]/g, ""), varMap);
        return { ...c, value: result };
      } catch {
        return { ...c, value };
      }
    })
  }));
}
```

This function dynamically renders and evaluates all simulation tables:

- Substitutes random variable placeholders with actual values.
- Uses the mathjs library to evaluate mathematical expressions within cells.
- Displays the final numeric results for both lecturer (view mode) and student tables.

This is where the simulation computation actually happens.

Probability range display (in lecturer table):

```
if (tableType === "probability") {
  const tableProbabilities = matrixData.map(
    (row) => randVars[row.cells[1]?.value] || 0
  );
  const ranges = calculateProbabilityRanges(tableProbabilities);
  display = ranges[r.row];
}
```

For probability tables, each row shows a range of outcomes based on probability values. This makes it easy for students to understand how random numbers map to specific simulation outcomes (e.g., demand levels or system states).

Key External Dependencies:

- **mathjs**: Safely evaluates mathematical expressions and formulas.
- **React State Hooks** (`useState`, `useEffect`): Handle dynamic table updates and mode-specific behavior.
- **InfoTooltip Component**: Provides contextual hints (not core to logic but improves user understanding).

7.3 Decision Theory Question

The `DecisionTheoryQuestion` component forms a core part of a larger React-based educational application.

It allows instructors to design, deploy, and evaluate questions grounded in Decision Theory, supporting models such as **MaxiMax**, **MaxiMin**, **Equally Likely**, and **Minimax Regret**.

The component supports three modes of interaction:

- **Edit Mode** – for instructors to configure and define decision problems.
- **Take Mode** – for students to input and calculate answers.
- **View Mode** – for displaying automatically evaluated results.

This modular design enables flexible use in online tests, simulations, or learning management systems.

Component Purpose and Architecture

The component blends mathematical evaluation, dynamic rendering, and data synchronization into one self-contained structure.

It uses React hooks for state management and `mathjs` for safe arithmetic formula evaluation.

Architectural Layers

1. **UI Layer**: Conditional JSX rendering depending on mode.
2. **Logic Layer**: Implements decision theory computations and marking logic.
3. **Data Layer**: Synchronizes question state with parent components via callbacks.

```
function DecisionTheoryQuestion({ question, mode, onUpdate, onDelete }) {  
  // internal state and helper logic  
  return (  
    <div className="p-4 border rounded mb-4 bg-white">  
      /* Renders dynamic question interface */  
    </div>  
  );  
}
```

Helper Functions:

Helper functions perform fundamental tasks such as formula evaluation, variable handling, and question marking.

Formula Normalisation and Evaluation

Before any math operation is evaluated, curly-brace placeholders (like {x1}) are converted to raw variable names.

```
// Replace {x1} → x1 for safe evaluation
const normalizeFormula = (formula) =>
  (formula || "").replace(/\{(\w+)\}/g, "$1");

// Evaluate formulas securely with mathjs
const safeEvaluate = (formula, vars) => {
  const normalized = normalizeFormula(formula);
  return evaluate(normalized, vars);
};
```

Purpose:

- Prevents code injection.
- Enables flexible variable references for dynamic test creation.

Truncating Decimals

Controls precision during random variable generation and output.

```
function truncateDecimals(num, decimals = 2) {
  if (isNaN(num)) return num;
  const factor = Math.pow(10, decimals);
  return Math.trunc(num * factor) / factor;
}
```

Purpose: Keeps numeric consistency when rounding isn't desirable (e.g., floating-point randomization).

Marking and Evaluation Helpers

evaluateQuestion(q, alts, vars)

Evaluates one sub-question, determines the correct answer, and assigns marks.

```

const evaluateQuestion = (q, alts = [], vars = {}) => {
  const method = (q.method || "").trim();
  let correctAnswer = "";
  let isCorrect = false;

  switch (method) {
    case "calculateMaxiMaxText":
      correctAnswer = calculateMaxiMaxText(alts, vars);
      isCorrect = q.userAnswer === correctAnswer;
      break;
    // ...other methods
  }

  const points = Number(q.points) || 4;
  return { correctAnswer, userScore: isCorrect ? points : 0, maxScore: points };
};

```

markAllQuestions(questionsList, userVariables, userAlternatives)

Processes a batch of sub-questions using the above function.

```

const markAllQuestions = (questionsList, userVariables, userAlternatives) => {
  return questionsList.map((q) => {
    const { correctAnswer, userScore, maxScore } = evaluateQuestion(q, userAlternatives, userVariables);
    return { ...q, correctAnswer, userScore, maxScore };
  });
};

```

Purpose:

- Allows batch scoring.
- Handles multiple decision method types dynamically.

Decision Theory Algorithms

Each algorithm implements a specific decision-making model. Every one has:

- A text-based version returning the best alternative name.
- A numeric version returning the computed value.

MaxiMax (Optemist's Criterion)

Selects the alternative with the highest possible payoff.

```
const calculateMaxiMaxText = (alts, vars = {}) => {
  const results = alts.map(alt => ({
    name: alt.name,
    favorable: safeEvaluate(alt.favorable, vars),
    unfavorable: safeEvaluate(alt.unfavorable, vars)
  }));

  const maxPerAlt = results.map(r => Math.max(r.favorable, r.unfavorable));
  const overallMax = Math.max(...maxPerAlt);
  const maxiMaxAlt = results.find((_, i) => maxPerAlt[i] === overallMax);

  return maxiMaxAlt?.name || "";
};
```

Numeric Version

```
const calculateMaxiMax = (alts, vars = {}) => {
  const maxPerAlt = alts.map(a =>
    Math.max(safeEvaluate(a.favorable, vars), safeEvaluate(a.unfavorable, vars))
  );
  return Math.max(...maxPerAlt);
};
```

MaxiMin (Pessimist's Criterion)

Chooses the alternative with the best worst-case payoff.

```
const calculateMaxiMinText = (alts, vars = {}) => {
  const results = alts.map(alt => ({
    name: alt.name,
    favorable: safeEvaluate(alt.favorable, vars),
    unfavorable: safeEvaluate(alt.unfavorable, vars)
  }));

  const minPerAlt = results.map(r => Math.min(r.favorable, r.unfavorable));
  const overallMin = Math.max(...minPerAlt);
  const maxiMinAlt = results.find((_, i) => minPerAlt[i] === overallMin);

  return maxiMinAlt?.name || "";
};
```

Equally Likely (Laplace Criterion)

Assumes each state has equal probability, selecting the highest average payoff.

```
const calculateEquallyLikelyText = (alts, vars = {}) => {
  const results = alts.map(alt => ({
    name: alt.name,
    favorable: safeEvaluate(alt.favorable, vars),
    unfavorable: safeEvaluate(alt.unfavorable, vars)
  }));

  const avgPerAlt = results.map(r => (r.favorable + r.unfavorable) / 2);
  const overallMax = Math.max(...avgPerAlt);
  return results.find((_, i) => avgPerAlt[i] === overallMax)?.name || "";
};
```

Minimax Regret

Minimizes potential regret between alternatives.

```
const calculateMinimaxRegretText = (alts, vars = {}) => {
  const results = alts.map(a => ({
    name: a.name,
    favorable: safeEvaluate(a.favorable, vars),
    unfavorable: safeEvaluate(a.unfavorable, vars)
  }));

  const opportunityLossTable = results.map(r => ({
    name: r.name,
    opportunityFavorable: Math.max(...results.map(a => a.favorable)) - r.favorable,
    opportunityUnfavorable: Math.max(...results.map(a => a.unfavorable)) - r.unfavorable,
  }));

  const maxPerAlt = opportunityLossTable.map(r =>
    Math.max(r.opportunityFavorable, r.opportunityUnfavorable)
  );
  const overallMin = Math.min(...maxPerAlt);

  return opportunityLossTable.find((_, i) => maxPerAlt[i] === overallMin)?.name || "";
};
```

Purpose:

Evaluates choices based on minimizing regret rather than maximizing reward, simulating risk-averse decision-making.

Data Flow and State Synchronization

The component uses multiple React `useState` and `useEffect` hooks to synchronize local and parent state.

It maintains real-time responsiveness while ensuring external updates persist.

Example — syncing field changes:

```
const handleChange = (field, value) => {
  onUpdate({
    ...question,
    data: { ...question.data, [field]: value },
  });
};
```

Key Synchronization Behaviors

- When `numAlternatives` or `includeDoNothing` changes → the alternatives table regenerates.
- When entering “take” mode → variables are randomized and persisted to parent.
- When “edit” mode updates formulas → they are validated instantly.

Matrix Rendering Logic

Students can perform calculations or build decision tables in an interactive matrix (“Rough Work Area”).

```
const renderMatrix = (matrixKey, editable) => {
  const targetMatrix = data[matrixKey];
  return (
    <table className="table-auto border w-full">
      <tbody>
        {targetMatrix.map((row) => (
          <tr key={row.row}>
            {row.cells.map((cell) => (
              <td key={cell.col} className="border px-2 py-1">
                {editable ? (
                  <input
                    value={cell.value}
                    onChange={(e) => updateCell(matrixKey, row.row, cell.col, e.target.value)}
                  />
                ) : (
                  <span>{cell.value}</span>
                )}
              </td>
            )}
          </tr>
        )}
      </tbody>
    </table>
  );
};
```

Purpose:

- Allows editable numeric input for problem-solving.
- Supports “Add Row” and “Add Column” actions dynamically.

- Used both for student input and for displaying computed answers.

Evaluation Integration

At export, the component provides evaluation utilities to integrate seamlessly into an assessment system.

```
export default {
  type: "decisionTheory",
  label: "Decision Theory Question",
  defaultData: { ... },
  Component: DecisionTheoryQuestion,
  getMaxScore: (q) =>
    q.data.questionsList.reduce((sum, i) => sum + (i.points || 0), 0),
  evaluate: (q) => {
    const updatedList = markAllQuestions(q.data.questionsList, q.data.userVariables, q.data.alternatives);
    return {
      userScore: updatedList.reduce((sum, i) => sum + (i.userScore || 0), 0),
      maxScore: updatedList.reduce((sum, i) => sum + (i.maxScore || 0), 0),
    };
  },
};
```

Purpose:

- This makes the component **plug-and-play** within a larger testing engine, automatically computing scores and tracking performance metrics.

Logical Flow Summary

Mode	Purpose	Key Actions
Edit Mode	Instructor Setup	Define prompt, variables, formulas, and scoring logic.
Take Mode	Student Interaction	Auto-generates variables, enables answer input and rough work.
View Mode	Evaluation display	Computes and displays scores, correct answers, and decision matrices.

Logical flow:

1. Instructor configures question → formulas validated.
2. Student takes test → randomized variables inserted.
3. On submission → evaluation logic runs via markAllQuestions.
4. Results displayed with automatic scoring in view mode.

10.4 Inventory Control Question

The InventoryControlQuestion component is an advanced React-based educational tool that allows instructors and students to interact with Inventory Control and Operations Management problems such as EOQ (Economic Order Quantity), Total Cost, and Reorder Point calculations.

It supports three operation modes:

- **Edit Mode:** Instructors design questions and define random variable ranges.
- **Take Mode:** Students receive unique randomized data and submit numeric answers.
- **View Mode:** System automatically evaluates and displays results, showing formulas and scores.

Component Purpose and Architecture

The component blends React state management, formula computation, and automated scoring to create a full question lifecycle.

Architecture Layers:

1. **Logic Layer** – Handles variable randomization, formula evaluation, and marking.
2. **UI Layer** – Dynamically renders question content based on mode.
3. **Integration Layer** – Communicates updates via onUpdate for parent data sync.

```
function InventoryControlQuestion({ question, mode, onUpdate }) {  
  const [variableValues, setVariableValues] = useState({});  
  const [studentAnswer, setStudentAnswer] = useState(question.data.userAnswer || "");  
  const [formulaError, setFormulaError] = useState("");  
  const [localTolerance, setLocalTolerance] = useState(question.data.tolerance ?? 0);  
}
```

Helper Functions

The component defines several utility functions to process data safely and accurately.

Number Normalisation

Converts user input (including commas or text) into valid numeric values.

```
function normalizeNumber(input, fallback = 0) {  
  if (input == null) return fallback;  
  const normalized = input.toString().trim().replace(",", ".");  
  const num = parseFloat(normalized);  
  return isNaN(num) ? fallback : num;  
}
```

Purpose: Ensures consistent handling of numeric data across locales (e.g., 2,5 → 2.5).

Decimal Truncation

Prevents floating-point rounding inconsistencies by truncating instead of rounding.

```
function truncateDecimals(num, decimals = 2) {  
  if (isNaN(num)) return num;  
  const factor = Math.pow(10, decimals);  
  return Math.trunc(num * factor) / factor;  
}
```

Purpose: Maintains deterministic values in random number generation and calculations.

Formula Computation

Implements various inventory management formulas (EOQ, AHC, TC, etc.) for automated calculation.

```
function computeFormula(method, vars) {  
  try {  
    switch (method) {  
      case "EOQ":  
        return Math.sqrt((vars.D * vars.Co) / vars.Ch);  
      case "AOC":  
        return (vars.D / vars.Q) * vars.Co;  
      case "AHC":  
        return (vars.Q / 2) * vars.Ch;  
      case "TC":  
        return (vars.D / vars.Q) * vars.Co + (vars.Q / 2) * vars.Ch;  
      case "ROP":  
        return vars.d * vars.L;  
      default:  
        return NaN;  
    }  
  } catch {  
    return NaN;  
  }  
}
```

Purpose: Standardizes computational logic for formulas within sub-questions.

Formula System and Definitions

A global FORMULAS array defines each method's label, required variables, and internal code.

```
const FORMULAS = [
  { code: "EOQ", label: "Economic Order Quantity", requires: ["D", "Co", "Ch"] },
  { code: "AOC", label: "Annual Ordering Cost", requires: ["D", "Q", "Co"] },
  { code: "TC", label: "Total Cost", requires: ["D", "Q", "Co", "Ch"] },
  { code: "ROP", label: "Reorder Point", requires: ["d", "L"] },
];
```

Purpose:

- Defines metadata for formula validation and selection.
- Ensures only formulas with available variables appear as selectable in edit mode.

Dynamic Variable Management

When instructors define {x1}, {x2}, etc., in the question prompt, the component dynamically detects them and updates the variable table.

```
useEffect(() => {
  if (mode !== "edit") return;
  const promptVars = Array.from(question.data.prompt.matchAll(/\{(\w+)\}/g)).map(m => m[1]);
  const existingVars = question.data.variables?.map(v => v.name) || [];
  const newVars = promptVars.filter(v => !existingVars.includes(v))
    .map(v => ({ name: v, type: "int", min: 0, max: 10, category: "" }));
  handleChange("variables", [...question.data.variables, ...newVars]);
}, [question.data.prompt, mode]);
```

Purpose: Keeps variable definitions synchronized with prompt content automatically.

Formula Validation Logic

Ensures that instructor-entered formulas are mathematically valid.

```
useEffect(() => {
  if (mode !== "edit" || !question.data.formula) return;
  try {
    const dummyVars = {};
    (question.data.variables || []).forEach(v => (dummyVars[v.name] = 1));
    evaluate(question.data.formula, dummyVars);
    setFormulaError("");
  } catch {
    setFormulaError("Invalid formula");
  }
}, [question.data.formula, question.data.variables]);
```

Purpose: Prevents invalid mathematical input from breaking student evaluations.

Randomisation in Take Mode

Generates unique, student-specific variable values within instructor-defined ranges.

```
useEffect(() => {
  if (mode !== "take") return;
  const vals = {};
  variableDefs.forEach(v => {
    const min = v.min ?? 0, max = v.max ?? 10;
    if (v.type === "int") vals[v.name] = Math.floor(random(min, max + 1));
    else vals[v.name] = truncateDecimals(random(min, max), v.decimals ?? 2);
  });
  handleChange("userVariables", vals);
  setVariableValues(vals);
}, [mode, variableDefs]);
```

Purpose: Provides controlled randomization for fairness and variety.

Formula Explanation Builders

These functions display formulas in both generic (symbolic) and numeric (contextual) forms for clarity in view mode.

```
function buildFormulaExplanation(method, vars) {
  switch (method) {
    case "AOC": return `=(${vars.D} ÷ ${vars.Q}) × ${vars.Co}`;
    case "EOQ": return `=√((${vars.D} × ${vars.Co}) ÷ ${vars.Ch})`;
    default: return "(formula not available)";
  }
}

function buildFormulaExplanationGeneric(method) {
  switch (method) {
    case "AOC": return "= (D ÷ Q) × Ordering Cost (Co)";
    case "EOQ": return "=√((D × Co) ÷ Ch)";
    default: return "(formula not available)";
  }
}
```

Purpose: Improves transparency in feedback by showing both theoretical and numeric representations.

Sub-Question Management

Each inventory control question can include multiple sub-questions, each with its own method, text, and points.

```
const addSubQuestion = () => {
  const newQuestion = { id: Date.now(), text: "", method: "", points: 0 };
  handleChange("subQuestions", [...(question.data.subQuestions || []), newQuestion]);
};

const updateSubQuestion = (index, field, value) => {
  const updated = [...(question.data.subQuestions || [])];
  updated[index][field] = value;
  handleChange("subQuestions", updated);
};
```

Purpose: Enables modular question design with individual scoring sections.

Evaluation and Scoring Logic

The scoring algorithm compares student answers to computed formula results using a fixed tolerance (± 0.2).

```
evaluate: (q) => {
  const subs = q.data.subQuestions || [];
  const vars = q.data.userVariables || {};
  const updatedSubs = subs.map((sq) => {
    const correct = computeFormula(sq.method, vars);
    const user = normalizeNumber(sq.userAnswer, NaN);
    const tol = 0.2;
    const isCorrect = Math.abs(user - correct) <= tol;
    const points = parseFloat(sq.points) || 0;
    return { ...sq, correctAnswer: correct, userScore: isCorrect ? points : 0, maxScore: points };
  });

  return {
    ...q,
    data: { ...q.data, subQuestions: updatedSubs },
    userScore: updatedSubs.reduce((s, x) => s + x.userScore, 0),
    maxScore: updatedSubs.reduce((s, x) => s + x.maxScore, 0),
  };
}
```

Purpose: Performs numeric evaluation and totals marks automatically.

User Interface Logic by Mode

Mode	Functionality	Features
Edit Mode	Instructor setup	Add prompt, define variables, assign formulas, and configure scoring.
Take Mode	Student answer input	Randomized variables injected into question text; answers typed directly.
View Mode	Evaluation and feedback	Displays computed results, formula explanations, and total scores.

View Mode Rendering Example:

```
<p><strong>Formula:</strong> {getFormulaLabel(sq.method)}<br/>
<em>{buildFormulaExplanationGeneric(sq.method)}</em><br/>
<em>{buildFormulaExplanation(sq.method, catVars)}</em></p>
```

Export and Integration Object

The component exports a structured object for easy integration into a larger question bank or LMS system.

```
export default {
  type: "inventoryControl",
  label: "Inventory Control Question",
  Component: InventoryControlQuestion,
  getMaxScore: (q) => (q.data.subQuestions || []).
    .reduce((sum, sq) => sum + (parseFloat(sq.points) || 0), 0),
  evaluate: (q) => { ... } // scoring logic as above
};
```

Purpose: Enables external systems to dynamically load, score, and display inventory questions.

Summary and Extensibility

The InventoryControlQuestion.jsx component encapsulates the logic for inventory-based quantitative questions, combining:

- Automatic variable parsing and range randomization.
- Built-in inventory formula evaluations.
- Clear numeric feedback and scoring automation.

Possible extensions:

- Add probabilistic inventory models (e.g., service-level EOQ).
- Integrate graphs or demand curves.
- Support for importing/exporting formula templates.

8. Evaluation Logic

The evaluation engine uses the **Math.js** library to interpret variable equations safely.

```
import { evaluate } from "mathjs";

function evaluateExpression(expression, variables) {
  try {
    return evaluate(expression, variables);
  } catch {
    return NaN;
  }
}
```

When a student submits an assessment:

1. Their input variables are collected into an object (e.g., {x1: 4, x2: 2}).
2. The lecturer's equations are re-evaluated using these inputs.
3. The output is compared to expected results within a tolerance threshold.

9. Security and Authentication

The authentication system enables users (students) to register, log in, and persist sessions using Firebase Authentication and Cloud Firestore.

It consists of three key parts:

- AuthService.jsx – Handles low-level Firebase authentication logic.
- Login.jsx – Authenticates existing users and regenerates Firestore profiles if missing.
- SignUpPage.jsx – Registers new users after validation against the PendingUser Firestore collection.

Both pages integrate seamlessly with the Firebase backend via the firestoreCrud() abstraction for reading and writing Firestore documents.

9.1 Authentication Flow Overview

The overall auth lifecycle:

Sign Up Page → Validate Student Number → Check PendingUser Exists → Firebase Auth (Create User) → Firestore(User) (Create Document) (UID) → Remove PendingUser Entry → Auto Login → Navigate to Home

Login Flow:

Login Page → Validate Student Number → Firebase Auth (Sign In) → Firestore(User) → Sync UID Record → Navigate to Home

9.2 AuthService Module

9.2.1 Firebase Integration

The AuthService.jsx module provides a secure interface to Firebase's authentication API. It encapsulates common operations: signing up, signing in, logging out, and tracking state changes.

```
import { auth } from "../firebase";
import {
  onAuthStateChanged as firebaseOnAuthStateChanged,
  signInWithEmailAndPassword,
  signOut,
  createUserWithEmailAndPassword,
} from "firebase/auth";

const AuthService = {
  onAuthStateChanged: (callback) => firebaseOnAuthStateChanged(auth, callback),
  signIn: async (email, password) => {
    try {
      return await signInWithEmailAndPassword(auth, email, password);
    } catch (err) {
      console.error(
        "Firebase auth error:",
        err.code,
        err.message,
        err.customData || err
      );
      throw err;
    }
  },
  signUp: (email, password) =>
    createUserWithEmailAndPassword(auth, email, password),
  signOut: () => signOut(auth),
};

export default AuthService;
```

9.2.2 Methods and Their Purposes

Method	Purpose	Return Value
signIn(email, password)	Authenticates a user with Firebase.	UserCredential object containing user data.
signUp(email, password)	Creates a new Firebase Auth user.	UserCredential with uid and email.
signOut()	Signs out the current Firebase session.	Promise<void>.

onAuthStateChanged(callback)	Listens for auth state changes (login/logout).	Unsubscribe function.
------------------------------	--	-----------------------

Purpose: Centralizes all authentication operations for easy reuse and better debugging.

9.3 Login Component

9.3.1 Core Functionality

The Login.jsx component authenticates users using student numbers and passwords, then cross-verifies their records in Firestore.

```
const handleSubmit = async (e) => {
  e.preventDefault();
  setIsLoading(true);
  const email = `${studentNumber}@mynwu.ac.za`;
  try {
    const userCredential = await AuthService.signIn(email, password);
    const authUser = userCredential.user;
    const uid = authUser.uid;
    let users = await firestoreCrud("User", "read");
    const exists = users.find((u) => u.id === uid);
    if (!exists) {
      const newUser = {
        uid,
        name: authUser.displayName || "Unknown",
        surname: "Unknown",
        studentNumber,
        email,
        role: "student",
        createdAt: new Date().toISOString(),
      };
    }
  }
}
```

```

    };
    await firestoreCrud("User", "create", newUser, uid);
  }
  navigate("/home", { replace: true });
} catch (err) {
  setError(
    err?.code === "auth/user-not-found"
      ? "Student number not found. Please sign up first."
      : "Invalid student number or password."
  );
} finally {
  setIsLoading(false);
}
};

```

9.3.2 Validation Logic

Before attempting authentication, the login form enforces:

```

const studentNumberPattern = /^\\d{8}$/;
if (!studentNumberPattern.test(studentNumber)) {
  setError("Student number must be exactly 8 digits and contain no letters.");
  return;
}

```

Purpose: Prevents invalid identifiers and ensures standardized student number formatting.

9.3.3 Firestore User Synchronisation

After signing in, the app verifies if the user exists in the Firestore User collection. If not, it automatically reconstructs the missing record.

```

if (!exists) {
  await firestoreCrud("User", "create", newUser, uid);
  console.log("Regenerated missing Firestore user for:", email);
}

```

Purpose: Keeps Firebase Authentication and Firestore in sync — essential for maintaining user role and profile data.

9.3.4 UI and State Management

The Login form uses React's `useState` hooks for UI interactivity:

- `showPassword`: Toggles visibility of password input.
- `isLoading`: Displays a loading spinner during sign-in.
- `error`: Displays validation or authentication errors.

Password toggle logic:

```
<button type="button" onClick={() => setShowPassword(!showPassword)}>
  {showPassword ? <EyeOff /> : <Eye />}
</button>
```

9.4 Sign-Up Component

The SignUpPage.jsx enables new users to register, ensuring they exist in a pre-validated PendingUser list.

9.4.1 Validation and Pending User Handling

- Ensures the student number is valid and unique.
- Confirms the student exists in Firestore's PendingUser collection.

```
const existingUsers = await firestoreCrud("User", "read");
const userExists = existingUsers.some(
  (u) => String(u.studentNumber) === normalizedStudentNumber
);
if (userExists) {
  navigate("/", { state: { message: "Account already exists. Please log in." } });
  return;
}

const pendingUsers = await firestoreCrud("PendingUser", "read");
const pendingUser = pendingUsers.find(
  (u) => String(u.studentNumber) === normalizedStudentNumber
);
if (!pendingUser) {
  setError("Student number not found.");
  return;
}
```

Purpose: Ensures only authorized students (approved in PendingUser) can register.

9.4.2 Account Creation with Firebase Auth

After validation, the system builds a structured email and registers a new Firebase user.

```
const email = `${normalizedStudentNumber}@mynwu.ac.za`;
const userCredential = await AuthService.signUp(email, password);
const uid = userCredential.user.uid;
```

9.4.3 Firestore Integration and Auto-Login

Once the Firebase user is created:

- A corresponding Firestore record is generated with the same UID.
- The pending user entry is deleted.
- The user is automatically logged in and redirected to /home.

```
await firestoreCrud("User", "create", newUser, uid);
await firestoreCrud("PendingUser", "delete", null, pendingUser.id);
await AuthService.signIn(email, password);
navigate("/home", { replace: true });
```

Purpose: Provides seamless account activation without requiring a manual login.

9.4.4 UI and Feedback Handling

Interactive states:

- showPassword, showConfirm: Toggle password visibility.
- isLoading: Prevents duplicate submissions.
- error: Displays human-readable Firebase errors (auth/weak-password, auth/email-already-in-use).

9.5 Data Synchronisation between Auth and Firestore

Action	Firebase Auth	Firestore Collection
Sign-Up	Creates UID + Email	Creates User doc with UID, role, and metadata
Sign-In	Verifies credentials	Checks Firestore profile, regenerates if missing
Sign-Out	Terminates session	No Firestore impact
PendingUser Deletion	N/A	Removes pre-approval entry post-registration

Purpose: Ensures 1:1 correspondence between Auth and Firestore entities.

9.6 Error Handling and Common Issues

Scenario	Firebase Error Code	User Message
Incorrect password	auth/wrong-password	"Invalid student number or password."
User not found	auth/user-not-found	"Please sign up first."
Duplicate registration	auth/email-already-in-use	"Account already exists."
Weak password	auth/weak-password	"Password is too weak."

All handled within:

```
catch (err) {
  if (err?.code === "auth/email-already-in-use") {
    setError("An account with that student number already exists.");
  }
}
```

9.7 Security Considerations

- **No plaintext password storage:** passwords are handled only by Firebase Authentication.
- **UID-based user mapping:** prevents collisions or data hijacking.
- **Role assignment on registration:** student/instructor separation maintained securely.
- **Regex input validation:** blocks malformed student numbers.
- **Error abstraction:** no sensitive data exposed to the client.

9.8 Conclusion

The authentication subsystem is a secure, modular, and scalable implementation using Firebase services.

It ensures:

- Consistent data synchronization between Firebase Auth and Firestore.
- Controlled registration via PendingUser pre-approval.
- Error-tolerant login and sign-up handling.
- Clear, user-friendly feedback for every auth state.

This architecture lays a solid foundation for future features such as role-based routing, multi-factor authentication, or email verification.

10. Support and contact information

For any uncertainty or assistance in using the application please reach out to the following people.

Ruan van Heerden

Email: 41763882@mynwu.ac.za

Hendrik Willem Mooiman

Email: 12345678@mynwu.ac.za

Andre De Goede

Email: 12345678@mynwu.ac.za

