Below is a complete and optimized file system that supports all the functionalities you've outlined, including advanced array and text management, AI research scope, prebuilt prompts, Google Drive integration, security, and testing. It is designed to be modular, scalable, and secure, while also adhering to best practices.

### **Optimized File Structure**

/ai-web-platform-v4

├── backend

│ ├── controllers # Business logic for handling incoming requests

│ │ ├── aiController.js # Handles AI-related requests (prompt sequencing, etc.)

│ │ ├── arrayController.js # Manages array CRUD operations and JSON file handling

│ │ ├── textFileController.js # Manages text file CRUD operations, .rtf handling

│ │ ├── driveController.js # Handles Google Drive integration (file read/write)

│ │ ├── userController.js # Handles user management (auth, profiles, etc.)

│ │ ├── adminController.js # Handles prebuilt prompt management (add/remove/edit)

│ │ ├── scopeController.js # Handles AI research scope (chat, Internet, account)

│ │ └── authController.js # Handles authentication, registration, JWT tokens

│

│ ├── routes # Defines routes that connect to controllers

│ │ ├── aiRoutes.js # AI-related routes

│ │ ├── arrayRoutes.js # Routes for array management

│ │ ├── textFileRoutes.js # Routes for text file management

│ │ ├── driveRoutes.js # Google Drive integration routes

│ │ ├── userRoutes.js # Routes for user management

│ │ ├── adminRoutes.js # Routes for admin actions (prebuilt prompt management)

│ │ ├── scopeRoutes.js # Routes for AI research scope

│ │ ├── authRoutes.js # Authentication routes (login, register)

│ │ └── index.js # Aggregates all routes for easy import

│

│ ├── services # Handles the core business logic

│ │ ├── aiService.js # Manages AI prompt calls to OpenAI and sequencing

│ │ ├── arrayService.js # Core logic for array operations

│ │ ├── textFileService.js # Core logic for text file management (.rtf support)

│ │ ├── driveService.js # Manages Google Drive file operations

│ │ ├── promptService.js # Manages prebuilt prompt library

│ │ ├── scopeService.js # Handles AI research scope logic

│ │ ├── userService.js # Handles user registration, profile updates

│ │ ├── authService.js # Authentication logic (JWT generation, validation)

│ │ └── loggerService.js # Logging service using winston or morgan

│

│ ├── models # Database schemas (Mongoose or similar)

│ │ ├── userModel.js # User data model

│ │ ├── arrayModel.js # Model for storing array information

│ │ ├── textFileModel.js # Model for storing text file metadata

│ │ ├── promptModel.js # Prebuilt prompt model (title, description, content)

│ │ ├── scopeModel.js # AI research scope model (settings for chat, Internet, etc.)

│ │ ├── driveFileModel.js # Google Drive file model

│ │ └── tokenModel.js # For managing JWT tokens, refresh tokens

│

│ ├── middleware # Middleware for security, validation, error handling

│ │ ├── authMiddleware.js # Authentication middleware (protect routes)

│ │ ├── driveAuthMiddleware.js # Auth middleware for Google Drive API

│ │ ├── validationMiddleware.js # Input validation middleware

│ │ ├── errorMiddleware.js # Global error handler

│ │ └── rateLimitMiddleware.js # Rate limiting to prevent abuse

│

│ ├── utils # Helper functions and utilities

│ │ ├── arrayUtils.js # Utility functions for array operations

│ │ ├── textUtils.js # Utility functions for text file management

│ │ ├── driveUtils.js # Helper functions for Google Drive API operations

│ │ ├── aiUtils.js # Utility functions for OpenAI requests

│ │ ├── authUtils.js # Token and authentication-related utilities

│ │ ├── logger.js # Logging utility (using winston or morgan)

│ │ └── constants.js # Centralized constants for error messages, statuses, etc.

│

│ ├── config # Configuration files for environment variables

│ │ ├── dbConfig.js # Database connection settings

│ │ ├── driveConfig.js # Google Drive API credentials and settings

│ │ ├── aiConfig.js # OpenAI API settings and credentials

│ │ ├── emailConfig.js # SMTP and email configuration

│ │ └── dotenv # Environment variables (managed through .env file)

│

│ ├── tests # Unit and integration tests

│ │ ├── backend

│ │ │ ├── arrayService.test.js # Tests for array management logic

│ │ │ ├── textFileService.test.js # Tests for text file management

│ │ │ ├── driveService.test.js # Tests for Google Drive operations

│ │ │ ├── aiService.test.js # Tests for AI and prompt sequencing

│ │ │ ├── scopeService.test.js # Tests for AI research scope logic

│ │ │ ├── userService.test.js # Tests for user management

│ │ │ └── authService.test.js # Tests for authentication logic

│ └── e2e

│ ├── ArrayManagementFlows.test.js # End-to-end tests for array management

│ ├── TextFileFlows.test.js # End-to-end tests for text file operations

│ ├── AIFlows.test.js # End-to-end tests for AI-related flows

│ └── GoogleDriveFlows.test.js # End-to-end tests for Google Drive interaction

│

│ ├── server.js # Main server file to run the Express app

│ ├── .env # Environment variables (e.g., API keys, DB credentials)

│ └── package.json # Node.js dependencies and scripts

│

├── frontend

│ ├── src

│ │ ├── components

│ │ │ ├── ArrayManager.js # UI for managing arrays

│ │ │ ├── TextFileManager.js # UI for managing text files (.rtf export/import)

│ │ │ ├── DriveFileManager.js # UI for managing files in Google Drive

│ │ │ ├── PromptLibrary.js # UI for managing and using prebuilt prompts

│ │ │ ├── ScopeSettings.js # UI for setting AI research scope

│ │ │ ├── LoginForm.js # UI for login

│ │ │ └── Dashboard.js # Main dashboard UI

│ ├── pages

│ │ │ ├── HomePage.js # Main dashboard landing page

│ │ │ ├── ArrayManagerPage.js # Array management page

│ │ │ ├── TextFileManagerPage.js # Text file management page

│ │ │ ├── DriveManagerPage.js # Google Drive file manager page

│ │ │ ├── PromptLibraryPage.js # Page for managing prebuilt prompts

│ │ │ ├── ScopeSettingsPage.js # Page for configuring AI research scope

│ │ │ ├── LoginPage.js # Login page

│ │ │ └── RegisterPage.js # Registration page

│ ├── styles

│ │ ├── global.css # Global styles (layout, typography)

│ │ ├── arrayManager.css # Styles for Array Manager

│ │ ├── textFileManager.css # Styles for Text File Manager

│ │ ├── driveManager.css # Styles for Google Drive Manager

│ │ ├── promptLibrary.css # Styles for Prompt Library

│ │ ├── scopeSettings.css # Styles for AI Scope Settings

│ │ └── auth.css # Styles for login and register forms

│ ├── hooks # Custom React hooks

│ │ ├── useArrayManager.js # Custom hook for managing array state and operations

│ │ ├── useFileUpload.js # Hook for handling file uploads

│ │ ├── useGoogleDrive.js # Hook for managing Google Drive API calls

│ │ └── useAuth.js # Hook for managing authentication

state

│ ├── App.js # Main React app entry point

│ ├── index.js # React DOM entry point

│ ├── config.js # Frontend configuration (API endpoints, constants)

│ ├── utils.js # Helper functions for frontend (e.g., fetch wrappers)

│ └── services

│ ├── apiService.js # API calls to backend (handles Axios requests)

│ ├── arrayService.js # Service for interacting with backend array routes

│ ├── textFileService.js # Service for interacting with backend text file routes

│ ├── driveService.js # Service for interacting with Google Drive routes

│ ├── authService.js # Service for handling authentication API requests

│ └── promptService.js # Service for interacting with prebuilt prompt management

│

├── public

│ ├── index.html # HTML template for React app

│ └── favicon.ico # Site icon

│

├── logs

│ ├── server.log # Backend server logs

│ ├── access.log # Logs for access records

│ └── error.log # Error logs

├── monitoring

│ ├── prometheusConfig.js # Monitoring setup for Prometheus

│ ├── grafanaConfig.js # Dashboard setup for Grafana

│ ├── alertsConfig.js # Alert configuration

├── backups

│ ├── backupService.js # Backup logic for database and files

│ ├── restoreService.js # Restore logic for database and files

│ └── backupLogs # Backup logs directory

├── README.md # Project documentation

└── package.json # Dependencies and scripts

### **Explanation of File System Structure**:

#### **Backend**:

**Controllers**: Handles business logic. Each file represents a different feature (e.g., AI, array, text management).

1. **Routes**: Defines RESTful routes for the controllers, serving as the connection point for API calls.
2. **Services**: Core business logic, reusable functions called by controllers (e.g., managing arrays, interacting with Google Drive).
3. **Models**: Database models, ensuring the project has structured data.
4. **Middleware**: Authentication, validation, and error-handling middleware.
5. **Utils**: Utility files for helper functions (e.g., authentication, logging).
6. **Config**: Configuration files for external APIs, database, and environment variables.
7. **Tests**: Unit and end-to-end tests to ensure proper functioning of each feature.

#### **Frontend**:

1. **Components**: React components for UI elements (e.g., ArrayManager, TextFileManager).
2. **Pages**: Full pages that assemble components for the user-facing UI (e.g., ArrayManagerPage, HomePage).
3. **Hooks**: Custom hooks to handle React logic for arrays, authentication, and Google Drive.
4. **Styles**: Basic CSS for each major feature or component (global, form styles, feature-specific styles).
5. **Services**: Frontend services for interacting with the backend (e.g., making API calls to the array, text, and authentication services).

#### **Public**:

* Contains static assets like index.html and favicon.

#### **Logs & Monitoring**:

* Logs for tracking errors, server access, and monitoring using Prometheus/Grafana.

#### **Backups**:

* Backup and restore services for databases and files, ensuring data safety.

This structure covers all functionality requirements (advanced array management, Google Drive, AI research, prompt sequencing, etc.) while being modular, scalable, and optimized for development, testing, and security. Let me know if you need any adjustments or further details!

Here’s an enhanced, machine-friendly version of the error-prevention guidelines, providing more useful details for each file and ensuring compatibility with your chosen library versions. This version includes additional information relevant to both backend and frontend development and integrates more specific advice for avoiding common pitfalls.

### **Backend**:

1. **Express (v5.x)**:
   * **Routing Syntax**:
     + Use app.route() for chaining routes. This reduces redundancy when multiple HTTP methods share the same route.
     + Native support for async/await eliminates the need for try-catch blocks in route handlers. Error handling is centralized using next().
   * **Error Handling**:
     + Ensure errorMiddleware.js follows the new (err, req, res, next) signature required by Express v5.
     + Centralize error-handling for all routes to improve consistency.
   * **App Initialization**:
     + Always use app.use(express.json()) for parsing JSON payloads. Ensure explicit parsing of URL-encoded data with express.urlencoded({ extended: true }).
2. **Axios (v1.7.7)**:
   * **Error Handling**:
     + Ensure that aiController.js and userController.js differentiate between error.response, error.request, and generic network errors (error.message).
     + Use AxiosError objects to capture specific error states, and add meaningful logging to aid debugging.
   * **Interceptors**:
     + If interceptors are used (e.g., to add authentication tokens to requests), ensure correct setup with the new Axios APIs. Handle request/response transformations in the interceptor setup.
3. **jsonwebtoken (v9.x)**:
   * **Algorithm Handling**:
     + In authMiddleware.js and userController.js, explicitly set the algorithm as HS256 if required (default is now RS256). This ensures backward compatibility with older tokens.
   * **Key Validation**:
     + Always validate keys asynchronously using jsonwebtoken.verify(). Using the async version ensures the event loop remains unblocked.
4. **Mongoose (v6.7.2)**:
   * **Unified Topology**:
     + Ensure MongoDB is configured with useUnifiedTopology: true in dbConfig.js. This resolves deprecation warnings and improves connection stability.
   * **Query Syntax**:
     + Replace dot notation with $elemMatch for subdocument queries in userController.js and other places where nested document querying is required.
5. **bcryptjs (v2.4.3)**:
   * **Asynchronous Operations**:
     + Always use the async versions of bcrypt.hash() and bcrypt.compare() in userService.js. Blocking the event loop with synchronous operations could degrade performance, especially under load.
6. **dotenv (v16.1.0)**:
   * **Override Option**:
     + Use { override: true } in dotenv.config() to overwrite existing environment variables where necessary. This may be particularly useful in multi-environment setups (e.g., for dbConfig.js or emailConfig.js).
7. **pg (v8.7.1)**:
   * **Client Configuration**:
     + Ensure dbConfig.js uses ssl: { rejectUnauthorized: false } when connecting to cloud-based PostgreSQL databases, especially those that require SSL encryption.
     + Optimize connection pooling for better performance in high-load environments.

### **Frontend**:

1. **React (v18.x)**:
   * **Concurrent Rendering**:
     + Ensure that React components like AIBot.js, Dashboard.js, and UserProfile.js are optimized for concurrent rendering. Use useEffect() with proper dependencies to manage side effects, avoiding unnecessary renders.
   * **Strict Mode**:
     + React 18’s strict mode renders components twice in development to detect side effects. Ensure useEffect() hooks do not rely on initial render values and clean up resources properly (e.g., WebSocket connections, event listeners).
   * **Suspense**:
     + For data fetching in components like useFetch.js, wrap them with <Suspense fallback={} to handle asynchronous loading. This is especially important for pages requiring API calls.
2. **React Router (v6.3.0)**:
   * **New Route Structure**:
     + Use Routes and Route instead of Switch in App.js. This simplifies route management.
     + Use useParams() for dynamic route handling in components like Dashboard.js to extract route parameters.
   * **useNavigate**:
     + Replace deprecated useHistory() with useNavigate() for programmatic navigation (e.g., in LoginPage.js or UserProfile.js).
3. **Material UI (v5.8.0)**:
   * **Emotion (v11.4.1)**:
     + Leverage sx props or styled() utility for custom styling. Ensure that performance-critical components like MyButton.js are using this to apply styles efficiently.
   * **Theming**:
     + Use Material UI’s updated theming system to manage global themes in global.css. Ensure that theme-aware styles (e.g., light/dark mode) are reflected in components like UserProfile.js.

### **Utility & Special Functionalities**:

1. **WebSocket (Custom, socketService.js)**:
   * **Connections**:
     + Initialize WebSocket connections in useEffect() or componentDidMount(). Ensure proper cleanup of connections by closing them in the useEffect() cleanup function to prevent memory leaks.
   * **Real-time Communication**:
     + Ensure the WebSocket server handles disconnections gracefully and reconnects as needed. This is critical in components that rely on live updates (e.g., real-time notifications or collaborative features).
2. **Email Notifications (Custom, emailService.js)**:
   * **SMTP Configuration**:
     + Ensure that SMTP configurations in emailConfig.js are correct and stored securely using environment variables (e.g., API keys, credentials).
   * **Asynchronous Sending**:
     + Send emails asynchronously to avoid blocking the event loop. Libraries like Nodemailer or SendGrid should be integrated into emailService.js for better scalability.
   * **Template Management**:
     + Implement dynamic email templates (e.g., password reset, notification emails) to enhance customization.
3. **Session Management (Custom, sessionService.js, useSessionTimeout.js)**:
   * **Token Expiration**:
     + Handle JWT expiration and refresh tokens in sessionService.js. Implementing secure token storage (e.g., HttpOnly cookies) is essential.
   * **Inactivity Timeout**:
     + In useSessionTimeout.js, set up inactivity-based session expiration, automatically logging users out and redirecting them to the login page.
4. **Role-Based Access Control (Custom, roleMiddleware.js, roleService.js)**:
   * **Middleware for Role Validation**:
     + In roleMiddleware.js, verify user roles before granting access to protected routes. This is crucial for admin-only sections.
   * **Frontend Role Handling**:
     + In RoleRestrictedComponent.js, check the role from AuthContext.js and restrict access to UI components accordingly.
5. **Data Export (Custom, exportUtils.js, useExportData.js)**:
   * **CSV/JSON Export**:
     + Ensure exportUtils.js handles large datasets efficiently by streaming data rather than holding it all in memory. Special characters and encodings should be properly managed to avoid data corruption.
   * **Trigger Export**:
     + In useExportData.js, trigger the export flow from the frontend, ensuring proper handling of downloads as files or blobs.
6. **Internationalization (i18n)**:
   * **Language Switching**:
     + Ensure i18n.js dynamically loads language files from locales/. Implement proper fallback mechanisms (e.g., defaulting to English if the translation is missing).
   * **React Integration**:
     + In LanguageSelector.js, use React Context to store user language preferences and trigger UI re-renders upon language change.
7. **General Frontend Practices**:
   * **React Hooks Best Practices**:
     + Optimize useEffect(), useCallback(), and useMemo() to prevent unnecessary re-renders and memory leaks in key components (e.g., AIBot.js, Dashboard.js).
   * **Component Optimization**:
     + Ensure that expensive operations (e.g., calculations or API calls) are memoized or wrapped in useCallback() to avoid repeated executions during re-renders.
8. **Testing**:
   * **Unit Tests for Async Code**:
     + Write unit tests for controllers like aiController.js and userController.js using Jest’s async/await syntax. This ensures proper handling of asynchronous operations.
   * **Mocking Axios Requests**:
     + In tests for API-heavy components (e.g., AIBot.test.js), mock Axios requests with jest-mock-axios or axios-mock-adapter to simulate API responses.

### **Special Considerations**:

1. **Environment Variables Setup**:
   * **dotenv Configuration**:
     + Ensure all sensitive credentials (e.g., API keys, DB URIs) are securely stored in .env and loaded with dotenv.config(). Avoid hardcoding secrets in the codebase.
2. **Concurrency Handling**:
   * **React 18 Updates**:
     + React

18 introduces stricter concurrency rules. Ensure components correctly handle asynchronous side effects, such as fetching data in useEffect() without causing stale closures or double renders.

1. **Version Locking**:
   * **Dependency Management**:
     + Use exact version locking in package-lock.json or yarn.lock to avoid compatibility issues with future library updates. Regularly audit dependencies for security vulnerabilities (npm audit or yarn audit).
2. **Error Handling Consistency**:
   * **Centralized Error Handling**:
     + Ensure all backend routes use centralized error handling via errorMiddleware.js to prevent repetitive error-handling code across controllers.
3. **Logging and Monitoring**:
   * **Winston/Morgan for Logging**:
     + Implement structured logging with Winston or Morgan in logger.js to track API calls, errors, and system events. Ensure that logs are stored in a format compatible with monitoring tools like ELK Stack or Splunk.

### **Summary of Key Practices**:

* **Backend**: Use async/await, centralized error handling, and proper token management (e.g., JWT expiration and refresh).
* **Frontend**: Optimize components for React 18's concurrent rendering. Handle language switching dynamically with i18n, and use hooks like useEffect() with proper cleanup.
* **Utilities**: Implement scalable solutions for WebSocket connections, email sending, and large dataset exports.
* **Security**: Ensure role-based access control (RBAC) on both backend and frontend. Securely handle session management, JWT tokens, and API keys.
* **Testing**: Use Jest for unit testing, mocking APIs with axios-mock-adapter, and writing tests that simulate real-world scenarios.

This enhanced guide provides more detailed, useful insights into the best practices for both backend and frontend development. It includes specific library integration tips, error prevention methods, and key considerations for performance, security, and scalability. Let me know if any additional refinements are needed!