My Review (MR) - Project Documentation

Table of Contents

Overview

My Work Review (MWR) is a comprehensive file review and collaboration platform designed for audit and review workflows. The application enables teams to:

- Upload and review PDF documents
- Add highlights, comments, and annotations
- Create structured checklists and sections
- Collaborate with team members and external reviewers
- Manage review templates and workflows
- Integrate with the Friday application for cross-platform functionality
- Generate reports and send automated email notifications

The application is built with React/TypeScript on the frontend and Firebase Cloud Functions (Node.js/Express) on the backend.

Application Structure

Main Routes:

- `/` - Home page (review list and management)
- `/login` - Login page (Google OAuth)
- `/filereview/:id` - File review interface (PDF viewer and annotations)
- `/settings/*` - Settings pages (templates, checklists, teams, users, permissions, integrations)
- `/forbidden` - Access denied page

Settings Sub-routes:

- `/settings/` - Settings home (overview)
- `/settings/templates/*` - Template management (Partner only)
- `/settings/checklists/*` - Checklist management (Partner only)
- `/settings/teams/*` - Team management (Partner only)
- `/settings/users/*` - User management (Partner only)
- `/settings/permissions/*` - Permission management
- `/settings/filereviews/*` - Review settings (flags, temporary access)
- `/settings/integrations/*` - Integration settings (API keys) (Partner only)

File Review Sub-routes:

- `/filereview/:id` - Main review interface
- `/filereview/:id/sections` - Section view
- `/filereview/:id/comparison` - PDF comparison mode

Code Location: `client/src/App.tsx` (lines 706-733)

Architecture

The application follows a client-server architecture:

- **Frontend**: React SPA (Single Page Application) deployed on Netlify
- **Backend**: Firebase Cloud Functions (Express.js) deployed on Firebase
- **Database**: Cloud Firestore (NoSQL)
- **Storage**: Firebase Storage + Google Drive API
- **Authentication**: Firebase Authentication with Google OAuth
- **Integration**: Custom token exchange with Friday application

Architecture Diagram

```
┌─────────────────┐      ┌──────────────────┐
┌─────────────────┐
│   React Client  │──────▶│ Firebase Cloud   │──────▶│    Firestore
│
│   (Netlify)     │      │   Functions      │      │    Database
│
└─────────────────┘      └──────────────────┘
└─────────────────┘
         │                        │                        │
         │                        │                        │
         ▼                        ▼                        ▼
┌─────────────────┐      ┌──────────────────┐
┌─────────────────┐
│ Firebase Auth    │      │ Google Drive     │      │ Firebase
Storage  │
│ (Google OAuth)   │      │      API         │      │
│
└─────────────────┘      └──────────────────┘
└─────────────────┘
         │
         │
         ▼
┌─────────────────┐
│ Friday App      │
│ (Token Exchange) │
└─────────────────┘
```

Technology Stack

Frontend Dependencies

Core Framework:

- `react` : ^18.2.0
- `react-dom` : ^18.2.0
- `react-router-dom` : ^6.14.0
- `typescript` : ^4.9.5

UI Libraries:

- `@mui/material` : ^5.14.4 (Material-UI components)
- `@mui/icons-material` : ^5.14.3
- `@mui/x-data-grid-premium` : ^5.17.26 (Premium data grid)
- `@mui/x-date-pickers-pro` : ^5.0.20
- `@emotion/react` : ^11.11.1
- `@emotion/styled` : ^11.11.0
- `styled-components` : ^5.3.10
- `framer-motion` : ^11.3.17

PDF Handling:

- `react-pdf` : ^7.3.3
- `pdfjs-dist` : ^2.16.105
- `pdf-lib` : ^1.17.1
- `@react-pdf/renderer` : ^3.1.12

Firebase:

- `firebase` : ^10.1.0
- `react-firebase-hooks` : ^5.1.1

Utilities:

- `axios` : ^1.4.0
- `moment` : ^2.29.4
- `date-fns` : ^2.30.0
- `lodash.debounce` : ^4.0.8
- `lodash.throttle` : ^4.1.1

- `uuid` : ^10.0.0

Permissions:

- `@casl/ability` : ^6.5.0
- `@casl/react` : ^3.1.0
- `react-acl` : ^2.0.0

Service Workers:

- `workbox-*` : Various Workbox packages for PWA functionality

Backend Dependencies

Core:

- `firebase-admin` : ^11.10.1
- `firebase-functions` : ^4.4.1
- `express` : ^4.18.2
- `cors` : ^2.8.5
- `body-parser` : ^1.20.2

Google APIs:

- `googleapis` : ^126.0.1
- `@google-cloud/firestore` : ^7.1.0
- `@google-cloud/storage` : ^7.12.1

File Processing:

- `busboy` : ^1.6.0
- `multer` : ^1.4.5-lts.1
- `puppeteer-core` : ^10.4.0
- `chrome-aws-lambda` : ^10.1.0

Email:

- `nodemailer` : ^6.9.5
- `handlebars` : ^4.7.8

Utilities:

- `moment` : ^2.29.4
- `moment-timezone` : ^0.5.44

- `axios` : ^1.5.0
- `dotenv` : ^16.4.1

Authentication System

Authentication Flow

1. **User Login**: Users sign in using Google OAuth via Firebase Authentication
2. **User Verification**: The system checks if the user exists in Firestore (`myworkreview/users/list`)
3. **Status Check**: Verifies user status is "active"
4. **UID Update**: Updates user document with Firebase UID if not already set
5. **Role Assignment**: Fetches user role and permissions from user groups
6. **Context Setup**: Sets up AuthContext with user data, permissions, and teams

Authentication Components

Frontend (`client/src/firebase.tsx`):

- `signInWithGoogle()` : Handles Google OAuth sign-in
- `logout()` : Signs out the user
- `auth` : Firebase Auth instance for My Work Review
- `friday_auth` : Firebase Auth instance for Friday integration

Backend (`functions/index.js`):

- Custom token generation endpoint: `/generate-custom-token`
- Verifies Friday ID tokens and generates custom tokens for MWR

User Management

- Users are stored in Firestore collection: `myworkreview/users/list`
- User fields:
  - `email` : User email (unique identifier)
  - `name` : User display name
  - `photo` : User profile photo URL
  - `uid` : Firebase Auth UID
  - `role` : Reference to user group/role
  - `teams` : Array of team references

- `status` : "active" or "inactive"
- `authProvider` : "google"
- `priorityReviews` : Array of priority review IDs

User Roles and Access Control

The application has **three primary user roles** defined in `lib/permissionTypes.ts` :

1. **Partner** (`Role.PARTNER`):
    - Highest level of access
    - Can access all settings sections
    - Can set deadline dates for tenders
    - Can manage flags and tags
    - Can set WIP values
    - Can archive/unarchive reviews
    - Can manage templates, checklists, teams, and users
    - Can configure review settings
    - Can access integrations settings
    - Protected routes: `ProtectedPartnerRoute` component restricts certain routes to Partners only
2. **Manager** (`Role.MANAGER`):
    - Mid-level access
    - Can manage reviews and checklists
    - Can add flags and tags
    - Can set WIP values
    - Cannot access Partner-only settings
    - Cannot set tender deadline dates
    - Cannot access certain administrative functions
3. **Clerk** (`Role.CLERK`):
    - Basic access level
    - Can view and comment on reviews
    - Can respond to checklist items
    - Cannot add flags or tags
    - Cannot set WIP values
    - Cannot access settings
    - Cannot archive reviews
    - Limited editing capabilities
    - Default role for new users synced from Google Workspace

Role Determination:

- Roles are stored in `myworkreview/userGroups/list` collection
- Each user has a `role` field that references a user group document
- User's `role_name` is derived from the user group's `name` field
- Hook: `useGetUserType()` provides `isClerk`, `isManager`, `isPartner` boolean flags

Code Location:

- Role definitions: `client/src/lib/permissionTypes.ts`
- Role checking: `client/src/hooks/useGetUserType.tsx`
- Route protection: `client/src/ProtectedRoute.tsx` and `client/src/ProtectedPartnerRoute.tsx`

Permission System

The application uses **CASL (Code Access Security Library)** for fine-grained permission management:

Permission Structure:

- **User Groups**: Defined in `myworkreview/userGroups/list`
    - Each user group has a `roles` object defining permissions
    - Permissions are resource-based (checklists, reviews, settings, templates, teams, users)
- **Permission Types** (defined in `permissions.tsx`):
    - `add`: Create new resources
    - `mark_all_resolved`: Mark all items as resolved
    - `change_status`: Change review status
    - `collaborators`: Manage collaborators
    - `private`: Make reviews private
    - `attach_checklist`: Attach checklists to reviews
    - `new_flags`: Create new flags
    - `temporary_access_period`: Configure temporary access
    - `manage`: Full management access
    - `all`: All permissions

Permission Resources:

- `checklists`: Checklist management permissions
- `reviews`: Review management permissions
- `settings`: Settings access permissions
- `templates`: Template management permissions

- `teams` : Team management permissions
- `users` : User management permissions

Permission Checking:

The application uses **two types of permission checks**:

1. **Permission Table-Based Checks** (Firestore):
   - Permissions stored in `myworkreview/userGroups/list` collection
   - Each user group has a `roles` object defining permissions
   - Components check `currentUser.permissions` object
   - CASL Ability: `defineAbilitiesForUser()` function creates ability object based on user roles
   - Example: `if (currentUser?.permissions?.checklists?.add) { ... }`

2. **Inline Permission Checks** (Hardcoded in Components):
   - **Important**: Many components have **inline permission checks** that don't rely on the permission table
   - These checks use `useGetUserType()` hook to get `isClerk` , `isManager` , `isPartner` boolean flags
   - Direct role checks using `currentUser.role_name` (e.g., `currentUser.role_name === "Partner"`)
   - These are hardcoded in the component logic and cannot be changed via the permission table
   - **Examples of inline checks**:
     - `FileReview.tsx` : `checkUserAuthorization()` function checks `isClerk` and `currentUser.role_name === "Partner"` for private review access
     - `Home.tsx` : Field actions check `!isClerk` for tags/WIP, `isPartner` for deadline dates
     - `FlexUpView.tsx` : Multiple inline checks like `if (isClerk) return;` , `if (!isPartner) return;`
     - `SettingsHome.tsx` : Checks `role_name !== "Clerk"` to block settings access
     - `FlagHolder.tsx` : Checks `isPartner` directly for flag permissions
     - `MobileReviewCard.tsx` : Inline checks for Partner-only actions
   - **Why inline checks exist**: Some permissions are role-based business logic that should not be configurable (e.g., only Partners can set deadline dates, Clerks cannot access settings)

Route Protection:

- `ProtectedRoute` component checks user authentication
- `ProtectedPartnerRoute` component restricts routes to Partners only (inline check)

Example Permission Checks:

```
// Permission table-based check
if (currentUser?.permissions?.checklists?.add) {
  // Show add checklist button
}

// Inline role check (hardcoded in component)
const { isPartner, isClerk } = useGetUserType();
if (isPartner) {
  // Show Partner-only features
}

// Inline role check using role_name
if (currentUser.role_name === "Partner") {
  // Partner-specific logic
}

// Inline check blocking Clerks
if (isClerk) return; // Early return for Clerks
if (!isClerk) {
  // Show feature for non-Clerks
}
```

Important Notes on Inline Checks:

- Inline permission checks are **hardcoded** in component logic
- They cannot be overridden by changing the permission table in Firestore
- These checks are intentional for role-based business logic
- To change inline checks, you must modify the component code directly
- Common inline checks:
  - Clerks cannot access settings (`role_name !== "Clerk"`)
  - Clerks cannot add tags or set WIP (`!isClerk`)
  - Only Partners can set deadline dates (`isPartner`)
  - Private review access logic (different for Partners vs Managers/Clerks)

Code Location:

- Permission definitions: `client/src/permissions.tsx`
- Permission types: `client/src/lib/permissionTypes.ts`

- Role checking hook: `client/src/hooks/useGetUserType.tsx`
- Permission checking: Throughout components using `AuthContext`

Components with Inline Permission Checks:

- `FileReview.tsx` : `checkUserAuthorization()` function (lines 1231-1267)
- `Home.tsx` : Field actions and cell click handlers (lines 2528-2552)
- `FlexUpView.tsx` : Multiple inline checks for tags, flags, WIP, deadline dates (lines 2242-2399)
- `SettingsHome.tsx` : Settings access check (blocks Clerks)
- `FlagHolder.tsx` : Flag permission check (checks `isPartner`)
- `MobileReviewCard.tsx` : Partner-only action checks
- `MobileEngagementCard.tsx` : Partner-only action checks
- `ProtectedPartnerRoute.tsx` : Route protection (checks `role_name !== "Partner"`)

Detailed Role-Based Permissions:

Partner Permissions

Full Access:

- ✅ Access all settings sections (Review Settings, Templates, Checklists, Teams, Users, Permissions, Integrations)
- ✅ Create, edit, and delete templates
- ✅ Create, edit, and delete checklists
- ✅ Create, edit, and delete teams
- ✅ Create, edit, and delete users
- ✅ Manage user roles and permissions
- ✅ Configure review settings (temporary access periods, flags, flag managers)
- ✅ Set tender deadline dates
- ✅ Add/remove flags and tags
- ✅ Set WIP (Work in Progress) values
- ✅ Archive/unarchive reviews
- ✅ Make reviews private
- ✅ Manage collaborators (permanent and temporary)
- ✅ Delete attachments (only Partners can delete review attachments)
- ✅ Remove checklists from reviews (only Partners can remove checklists)
- ✅ Toggle ML queries on highlights
- ✅ Resolve and unresolve highlights (full control)
- ✅ Unresolve partially resolved highlights created by others
- ✅ Access private reviews (if in team or collaborator)
- ✅ Set review status (Open/Closed)
- ✅ Mark all checklist items as resolved

- ✅ Access Partner-only routes (`ProtectedPartnerRoute`)

Code Locations:

- Attachment deletion: `FileReviewAttachmentBar.tsx` (line 151)
- Checklist removal: `FileReviewSectionsView.tsx` (line 307)
- ML query toggle: `FileReview.tsx` (line 2809)
- Highlight resolution: `FileReviewSectionsView.tsx` (lines 997-1055)

Manager Permissions

Moderate Access:

- ✅ Access settings (except Partner-only sections)
- ✅ Create and edit reviews
- ✅ Add flags and tags
- ✅ Set WIP values
- ✅ Manage collaborators (permanent and temporary)
- ✅ Resolve highlights (if not already resolved)
- ✅ Unresolve highlights (only if they resolved it themselves)
- ✅ Unresolve partially resolved highlights (only if they created the partial resolution)
- ✅ Add checklists to reviews
- ✅ Respond to checklist items
- ✅ Create highlights and responses
- ✅ Access non-private reviews
- ✅ Access private reviews (if creator, collaborator, or privateBy)
- ❌ Cannot set tender deadline dates (Partner-only)
- ❌ Cannot delete attachments
- ❌ Cannot remove checklists from reviews
- ❌ Cannot toggle ML queries
- ❌ Cannot unresolve highlights resolved by others
- ❌ Cannot unresolve partially resolved highlights created by others
- ❌ Cannot access Partner-only routes
- ❌ Cannot create/edit templates (typically)
- ❌ Cannot create/edit checklists (typically)
- ❌ Cannot manage teams (typically)
- ❌ Cannot manage users (typically)

Code Locations:

- Deadline date restriction: `Home.tsx` (line 2558)
- Checklist addition: `FileReviewSectionsView.tsx` (line 217)
- Highlight resolution: `FileReviewSectionsView.tsx` (lines 997-1055)

Clerk Permissions

Basic Access:

- ✅ View reviews (if creator, collaborator, or non-private)
- ✅ Create highlights (queries)
- ✅ Create responses to highlights
- ✅ Respond to checklist items
- ✅ View review details, sections, and checklists
- ✅ Access private reviews (if creator or collaborator)
- ❌ Cannot access settings (`SettingsHome.tsx` blocks Clerks)
- ❌ Cannot add flags or tags
- ❌ Cannot set WIP values
- ❌ Cannot set tender deadline dates
- ❌ Cannot archive/unarchive reviews
- ❌ Cannot make reviews private
- ❌ Cannot manage collaborators
- ❌ Cannot delete attachments
- ❌ Cannot add or remove checklists
- ❌ Cannot toggle ML queries
- ❌ Cannot resolve highlights
- ❌ Cannot unresolve highlights
- ❌ Cannot change review status
- ❌ Cannot mark checklist items as resolved
- ❌ Cannot create tender reviews (filtered out in template selection)
- ❌ Cannot access Partner-only routes

Code Locations:

- Settings access: `SettingsHome.tsx` (blocks Clerks)
- Tags/WIP restriction: `Home.tsx` (line 2530)
- Checklist addition: `FileReviewSectionsView.tsx` (line 217)
- Highlight resolution: `FileReviewSectionsView.tsx` (line 1001)
- Template filtering: `FileReviewCreateDialog.tsx` (line 104)

Access Control Examples:

1. **Settings Access**:
    - Clerks: Cannot access settings (`SettingsHome.tsx` checks `role_name !== "Clerk"`)
    - Managers/Partners: Can access settings based on permissions
    - Partners: Full access to all settings sections

2. **Review Actions**:
    - Clerks: Cannot add flags, tags, or set WIP values
    - Managers: Can add flags, tags, set WIP values
    - Partners: Can set deadline dates, full review management
3. **Template/Checklist Management**:
    - Access controlled by `currentUser.permissions.templates.add` and `currentUser.permissions.checklists.add`
    - Typically only Partners have these permissions
4. **Highlight Resolution**:
    - Clerks: Cannot resolve or unresolve highlights
    - Managers: Can resolve highlights, can unresolve only if they resolved it
    - Partners: Full control over resolution/unresolution
5. **Attachment Management**:
    - Clerks: Cannot delete attachments
    - Managers: Cannot delete attachments
    - Partners: Can delete attachments (except main review file)
6. **Checklist Management**:
    - Clerks: Cannot add or remove checklists
    - Managers: Can add checklists, cannot remove
    - Partners: Can add and remove checklists
7. **Team/User Management**:
    - Access controlled by `currentUser.permissions.settings.teams.add` and `currentUser.permissions.settings.users.add`
    - Typically only Partners have these permissions

Friday Integration

Overview

My Work Review integrates with the **Friday** application to enable cross-platform functionality. Friday is a separate Firebase application used for engagement management.

Integration Architecture

1. **Dual Firebase Apps**: The application initializes two Firebase apps:
   - **My Work Review App**: `audit-7ec47` (primary)
   - **Friday App**: `friday-a372b` (secondary)
2. **Token Exchange Flow**:

   ```
   MWR User → Gets ID Token → Sends to Friday Cloud Function
   → Friday verifies token → Generates custom token for MWR
   → MWR signs in with custom token → Access to Friday Firestore
   ```

3. **Backend Token Exchange** (`functions/index.js`):
   - Endpoint: `/generate-custom-token`
   - Verifies Friday ID token using `serviceAccountKeyFriday.json`
   - Generates custom token for My Work Review app
   - Returns token to Friday application

Friday Integration Features

1. **Checklist Import**:
   - Endpoint: `/api/push-mwr-checklist`
   - Friday can push checklists to MWR reviews
   - Files are automatically uploaded to Google Drive
   - Checklist structure is preserved
2. **Review Data Retrieval**:
   - Endpoint: `/api/friday-get-reviews`
   - Returns review names and financial years
   - Requires API key authentication
3. **Checklist Retrieval**:
   - Endpoint: `/api/get-checklists`
   - Returns all checklists for an organization
   - Requires API key authentication

Friday API Authentication

- **API Key System**: Organizations have API keys stored in `apis` collection

- **Key Validation**: Middleware validates API keys for server-to-server requests
- **Header**: `x-server-server` header indicates server-to-server requests
- **Authorization**: Bearer token in Authorization header

Friday Firestore Access

The frontend can access Friday Firestore data:

- **Teams**: `friday/teams/list`
- **Engagements**: Friday engagement data
- **Clients**: Friday client data

Code Location: `client/src/App.tsx` - `handleGetFridayToken()` and `getFridayTeams()`

Firebase Configuration

Firebase Projects

1. **My Work Review Project**:
   - **Project ID**: `audit-7ec47`
   - **Project Alias**: "My Work Review"
   - **Firebase Config** (client):

     ```
     apiKey: "AIzaSyB3hDttJnMDlZr7Vi6OhDQfBPBLQaQxynU"
     authDomain: "audit-7ec47.firebaseapp.com"
     projectId: "audit-7ec47"
     storageBucket: "audit-7ec47.appspot.com"
     ```

2. **Friday Project** (Integration):
   - **Project ID**: `friday-a372b`
   - **Firebase Config** (client):

     ```
     apiKey: "AIzaSyAdtzhTr38NDTj5ATL3EJebHPHusAoAEH0"
     authDomain: "friday-a372b.firebaseapp.com"
     projectId: "friday-a372b"
     ```

Firestore Database Structure

```
myworkreview/
├── users/
│   └── list/              # User documents
├── userGroups/
│   └── list/              # User group/role definitions
├── teams/
│   └── list/              # Team definitions
├── templates/
│   └── list/              # Review templates
├── reviews/
│   └── list/              # Review documents
├── checklists/
│   └── list/              # Checklist templates
├── logs/
│   └── list/              # System logs
├── apis/
│   └── list/              # API keys for organizations
└── removedHighlights/
    └── list/              # Removed highlights archive
```

Storage System

The application uses a **dual storage system** for different types of files:

1. Google Drive Storage

Purpose: Primary storage for review PDFs and review attachments

Upload Process:

- **Review PDFs**: Uploaded via `/api/upload-file` endpoint
    - Creates review folder in Google Drive
    - Creates "attachments" subfolder
    - Uploads PDF to review folder
    - Returns Google Drive file ID (stored in `reviewDetails.fileUrl`)
    - Parent folder: `1LZwInex0XVr6UKqyY_w6G2JeUyeUiiPG`
- **Review Attachments**: Uploaded via `/api/upload-attachment-file` endpoint
    - Uploads to review's attachment folder
    - Returns Google Drive file ID
    - Stored in `reviewDetails.attachments` array

File Retrieval:

- **PDF Retrieval**:
  - Endpoint: `/api/fetch-review-cached` (recommended, uses caching)
  - Endpoint: `/api/fetch-review` (direct, no caching)
  - Process:
    1. Checks Google Cloud Storage cache first
    2. If cached and up-to-date, serves from cache
    3. If not cached or outdated, fetches from Google Drive
    4. Updates cache with new file
    5. Streams PDF to client
  - Caching: Files cached in `cached_reviews` bucket
  - Cache TTL: 24 hours (checks file modification time)
- **Attachment Retrieval**:
  - Endpoint: `/api/get-attachment-file`
  - Process: Fetches file from Google Drive by file ID
  - Returns: File stream with appropriate content type

Drive Folder Structure:

```
Parent Folder (1LZwInex0XVr6UKqyY_w6G2JeUyeUiiPG)/
└── Review Folder (reviewFolderID)/
    ├── Review PDF (fileUrl - Google Drive file ID)
    └── attachments/ (attachmentFolderURL)/
        └── [Attachment files]
```

Code Location:

- Upload: `functions/index.js` - `/api/upload-file`, `/api/upload-attachment-file`
- Retrieval: `functions/index.js` - `/api/fetch-review-cached`, `/api/get-attachment-file`
- Frontend: `client/src/FileReview.tsx` - `loadPdfFromDrive()` function

2. Firebase Storage

Purpose: Storage for chat attachments and general file uploads

Upload Process:

- **Chat Attachments**: Uploaded via `uploadFilesToFirebase()` function

    - Path: `{organisation}/{reviewId}/{filename}_{timestamp}`
    - Uses `uploadBytesResumable` for resumable uploads
    - Returns download URL via `getDownloadURL()`
    - Stored in Firebase Storage bucket: `audit-7ec47.appspot.com`
- **Friday Chat Attachments**: Uploaded via `uploadFridayGeneralFilesToFirebase()`
    - Path: `{clientId}/{engagementId}/{rfiId}/{questionId}/{filename}_{timestamp}`
    - Uses Friday Firebase Storage: `friday-a372b.appspot.com`

File Retrieval:

- **Direct Download URLs**:
    - Files are retrieved using `getDownloadURL()` from Firebase Storage
    - URLs are stored in Firestore and used directly in the application
    - No backend endpoint required for Firebase Storage files

Storage Path Structure:

```
Firebase Storage (audit-7ec47.appspot.com)/
├── {organisation}/
│   └── {reviewId}/
│       └── {filename}_{DDMMYY_HHmmss}  # Chat attachments
│
Friday Storage (friday-a372b.appspot.com)/
└── {clientId}/
    └── {engagementId}/
        └── {rfiId}/
            └── {questionId}/
                └── {filename}_{DDMMYY_HHmmss}  # Friday chat attachments
```

Code Location:

- Upload: `client/src/globalFunctions/uploadFilesToApi.tsx`
- Friday Upload: `client/src/globalFunctions/uploadFridayFilesToApi.tsx`
- Retrieval: Direct download URLs from Firebase Storage

Storage Decision Matrix

File Type	Storage Location	Upload Method	Retrieval Method

File Type	Storage Location	Upload Method	Retrieval Method
Review PDF	Google Drive	`/api/upload-file`	`/api/fetch-review-cached`
Review Attachments	Google Drive	`/api/upload-attachment-file`	`/api/get-attachment-file`
Chat Attachments	Firebase Storage	`uploadFilesToFirebase()`	Direct download URL
Friday Chat Files	Friday Firebase Storage	`uploadFridayGeneralFilesToFirebase()`	Direct download URL
Checklist Response Attachments	Google Drive	`/api/upload-attachment-file`	`/api/get-attachment-file`

Key Differences:

- **Google Drive**: Used for review documents and attachments (organized in folders)
- **Firebase Storage**: Used for chat attachments and temporary files (organized by path)
- **Caching**: Google Drive files are cached in Google Cloud Storage for performance
- **Access**: Google Drive files require backend API calls, Firebase Storage files use direct URLs

Firebase Cloud Functions

Deployment Configuration (`firebase.json`):

```
{
  "functions": {
    "source": "functions",
    "codebase": "default"
  }
}
```

Function Endpoints:

- Base URL: `https://us-central1-audit-7ec47.cloudfunctions.net`
- Main API: `/api/*`
- High Memory API: For resource-intensive operations

Features

1. File Review System

PDF Review Interface (`FileReview.tsx`):

- PDF viewer with zoom controls
- Text and area highlighting
- Comment system with threaded responses
- Drawing/annotation tools
- Page navigation
- Side-by-side comparison mode
- Mobile-responsive design

Key Components:

- `PdfHighlighter` : Main PDF rendering component
- `Highlight` : Text highlight component
- `AreaHighlight` : Area selection component
- `Popup` : Comment popup component
- `Sidebar` : Review sidebar with highlights and comments

Highlight and Annotation System

PDF Library: The application uses **PDF.js** (Mozilla's PDF rendering library) for PDF viewing and annotation.

Library Details:

- **Package**: `pdfjs-dist` (version 2.16.105)
- **Components Used**:
  - `PDFViewer` : Main PDF viewer component
  - `PDFLinkService` : Handles PDF links and navigation
  - `EventBus` : Event system for PDF viewer interactions
  - `PDFDocumentProxy` : PDF document object

Code Location: `client/src/components/PdfHighlighter.tsx`

How Highlights Work:

1. **Text Selection**:
   - User selects text in the PDF using mouse drag
   - Selection is captured via `getBoundingRect()` and `getClientRects()` helper functions

    - Text selection creates a `Range` object from the DOM
    - Multiple text selections on the same page create multiple rects

2. **Area Selection**:
    - User can create area highlights (rectangular selections) without text
    - Area highlights are created by dragging on the PDF canvas
    - Uses `react-rnd` library for resizable/draggable rectangles
    - Area highlights store screenshot of the selected area

3. **Coordinate System**:
    - Highlights use **scaled coordinates** (0-1 range) for position storage
    - Coordinates are stored relative to page dimensions (width, height)
    - This allows highlights to resize correctly when PDF zoom changes
    - Coordinate transformation functions:
        - `viewportToScaled()` : Converts viewport coordinates to scaled coordinates
        - `scaledToViewport()` : Converts scaled coordinates to viewport coordinates
        - `pdfToViewport()` : Converts PDF coordinates to viewport coordinates

4. **Resizing According to Screen Size**:
    - Highlights automatically resize when PDF zoom changes
    - Uses `ResizeObserver` to detect container size changes
    - When viewport changes, highlights are recalculated using:

      ```
      scaledToViewport(scaled, viewport, usePdfCoordinates)
      ```

    - Formula: `x1 = (width * scaled.x1) / scaled.width`
    - Highlights are re-rendered on every scroll and zoom event
    - Debounced resize handler prevents excessive re-renders

5. **Highlight Colors**:
    - **Default Highlight**: `#B0B0B0` (gray) - Standard highlights
    - **Scrolled To Highlight**: `#7F7EFF` (purple) - When highlight is scrolled to from sidebar
    - **Partner Highlight**: `#ff4141` (red) - Highlights created by Partners
    - **Resolved Highlight**: `#44BBA4` (green) - Resolved highlights
    - Color is determined by CSS classes:
        - `.Highlight--scrolledTo` : Purple background
        - `.Highlight--partner` : Red background
        - `.Highlight--resolved` : Green background
        - Default: Gray background

6. **Scroll to Highlight**:

   - When clicking a highlight in the sidebar, the PDF scrolls to that highlight
   - Process:
     1. `scrollTo()` function is called with highlight object
     2. PDF viewer scrolls to the correct page using `scrollPageIntoView()`
     3. Calculates exact position using viewport coordinates
     4. Scrolls to position with 10px margin from top
     5. Sets `scrolledToHighlightId` state to highlight the target
     6. Highlight is rendered with purple color (`Highlight--scrolledTo` class)
     7. After 2-3 seconds, scroll highlight is cleared
   - Special handling for "PDF Editor Generated Query" highlights:
     - Scrolls to page only (no position calculation)
     - Clears scroll highlight after 3 seconds
   - Code Location: `client/src/components/PdfHighlighter.tsx` (lines 846-905)

7. **Highlight Storage**:

   - Highlights are stored in Firestore: `myworkreview/reviews/list/{reviewId}`
   - Structure:

```
{
  highlights: [
    {
      id: "highlight-id",
      position: {
        boundingRect: { x1, y1, x2, y2, width, height, pageNumbe
        rects: [{ x1, y1, x2, y2, width, height, pageNumber }],
        pageNumber: number,
        usePdfCoordinates: boolean
      },
      comment: {
        text: string,
        emoji: string
      },
      content: {
        text?: string,
        image?: string (base64 screenshot for area highlights)
      },
      email: string (creator email),
      resolved: boolean,
      responses: [
        {
                id: "response-id",
                position: { ... },
                comment: { ... },
                email: string,
                datetime: Timestamp
              }
            ]
          }
        ]
      }
```

8. **Highlight Rendering**:
   - Highlights are grouped by page number
   - Each page has a `HighlightLayer` component
   - Highlights are rendered as absolute positioned divs
   - Text highlights use multiple rects (one per line)
   - Area highlights use single bounding rect with screenshot
   - Highlights are re-rendered on:
     - PDF zoom change
     - PDF scroll
     - Page change
     - Highlight updates
9. **Annotation System**:
   - **Comments**: Each highlight can have a comment (text + emoji)
   - **Responses**: Highlights can have threaded responses
   - **Attachments**: Responses can have file attachments
   - **Resolution**: Highlights can be marked as resolved
   - **User Roles**: Highlight color changes based on creator's role (Partner = red)
10. **Deletion Restrictions**:
    - **Highlights/Queries Cannot Be Deleted**: Highlights (queries) cannot be deleted from the system
    - **Archival Instead**: When highlights are removed from the active highlights array, they are archived to `myworkreview/removedHighlights/list/{reviewId}`
    - **Archival Process**:
      - Firestore trigger `updatedReviewTriggerNotification` detects removed highlights
      - Compares old and new highlights arrays
      - Identifies highlights that were removed

        - Archives removed highlights to `removedHighlights` collection
        - Appends to existing archived highlights if document exists
    - **Responses Cannot Be Deleted**: Responses to highlights cannot be deleted
    - **Response Attachments**: Response attachments can be removed (via `handleDeleteSelectedHighlightResponseAttachment`), but responses themselves remain
    - **Why No Deletion**:
        - Maintains audit trail of all queries and responses
        - Preserves historical context of review discussions
        - Ensures data integrity and accountability
    - **Attachment Deletion Restrictions**:
        - Main review file cannot be deleted (enforced in `FileReviewAttachmentBar.tsx`)
        - Attachments in use by highlights or responses cannot be deleted
        - Only Partners can delete attachments
        - System checks if attachment is referenced in highlights or responses before allowing deletion
    - **Code Location**:
        - Highlight archival: `functions/index.js` - `updatedReviewTriggerNotification` (lines 1717-1741)
        - Attachment deletion: `client/src/FileReviewAttachmentBar.tsx` (lines 147-184)

Code Locations:

- PDF Viewer: `client/src/components/PdfHighlighter.tsx`
- Highlight Component: `client/src/components/Highlight.tsx`
- Area Highlight: `client/src/components/AreaHighlight.tsx`
- Coordinate Functions: `client/src/lib/coordinates.ts`
- Highlight Styles: `client/src/style/Highlight.css`

2. Review Management

Review Creation (`FileReviewCreateDialog.tsx`):

- Create new reviews with templates
- Upload PDF files
- Set financial year
- Assign teams
- Configure tender details (if applicable)

Review Properties:

- `groupName` : Review name
- `financialYear` : Financial year
- `template` : Review template reference
- `team` : Team reference
- `status` : Open/Closed
- `flags` : Status flags (Open, Closed, Missed, Declined, etc.)
- `collaborators` : List of collaborators
- `sections` : Review sections/checklists
- `highlights` : PDF highlights and comments
- `attachments` : File attachments

3. Checklist System

Checklist Features:

- Structured section items
- Response tracking
- File attachments per response
- Resolution status
- Email notifications
- PDF report generation

Checklist Structure:

```
{
  name: "Checklist Name",
  cloud_id: "checklist-id",
  section_items: [
    {
      id: "item-id",
      question: "Question text",
      responses: [
        {
          user: "user-email",
          response: "Response text",
          datetime: Timestamp,
          attachment: { url, name, type }
        }
      ],
      resolved: boolean
```

```
        }
      ]
    }
```

4. Collaboration Features

Collaborators:

- **Permanent Collaborators**: Full access with no expiration
- **Temporary Collaborators**: Time-limited access with expiration date
- Email notifications on collaborator changes
- Automatic access removal for expired temporary collaborators

Collaboration Tools:

- Real-time updates via Firestore listeners
- Comment threads
- Notification system
- Review sharing

Chat Synchronization System

How Chats Are Synced When Linked to Friday:

1. **Chat Collections**:
    - **Review Chat**: `myworkreview/reviews/list/{reviewId}/chat`
    - **Friday Chat**: `friday/engagements/list/{engagementId}/chat`
    - When a review is linked to a Friday engagement, both chat collections are synced
2. **Chat Sync Process**:
    - **In Review Chat** (`FlexUpChat.tsx`):
        - If review has `friday_link` , subscribes to both chat collections
        - Review chat: `myworkreview/reviews/list/{reviewId}/chat`
        - Friday chat: `friday/engagements/list/{friday_link}/chat`
        - Messages from both collections are merged and sorted by datetime
        - Messages are tagged: `isFridayMessage: true/false`
    - **In Friday Chat** (`FlexUpFridayChat.tsx`):
        - If engagement has `review_link` , subscribes to both chat collections
        - Friday chat: `friday/engagements/list/{engagementId}/chat`

        - Review chat: `myworkreview/reviews/list/{review_link}/chat`
        - Messages from both collections are merged and sorted by datetime
        - Messages are tagged: `isReviewMessage: true/false`

3. **Real-Time Synchronization**:

    - Uses Firestore `onSnapshot()` listeners for real-time updates
    - Both chat collections are monitored simultaneously
    - When a message is added to either collection, it appears in both interfaces
    - Messages are sorted by `datetime` field (ascending order)

4. **Message Merging**:

    - Messages from both collections are combined into a single array
    - Messages are sorted by `datetime` timestamp
    - Each message includes:
        - `cloud_id` : Document ID
        - `user_email` : Sender email
        - `message` : Message text
        - `datetime` : Timestamp
        - `photo` : User photo (from users map)
        - `isFridayMessage` or `isReviewMessage` : Source indicator
        - `attachments` : File attachments (if any)

5. **Chat Icon Color Updates**:

    - When new messages are detected, chat icon color is updated
    - Uses `updateChatIconColor()` callback function
    - Updates based on last message timestamp
    - Color indicates unread messages

6. **Unsubscribe Logic**:

    - When component unmounts or selection changes, both listeners are unsubscribed
    - Prevents memory leaks and unnecessary updates
    - Cleanup function returns unsubscribe functions

Code Locations:

- Review Chat: `client/src/components/FlexUpChat.tsx` (lines 427-549)
- Friday Chat: `client/src/components/FlexUpFridayChat.tsx` (lines 428-552)
- Chat Sync Logic: Both components handle bidirectional sync

5. Template System

Review Templates:

- Pre-configured review structures
- Section definitions
- Checklist templates
- Template management interface

Template Types:

- **Standard Templates**: Regular review templates
- **Tender Templates**: Templates for tender reviews
- **Friday Templates**: Templates linked to Friday engagements

Template Structure:

```
{
  name: string,
  cloud_id: string,
  template_type: "standard" | "tender" | "friday",
  sections: [...], // Checklist sections
  description: string
}
```

How Reviews Are Linked to Templates:

1. **Template Assignment**:
    - When creating a review, user selects a template
    - Template is stored in review document: `review.template = { name, cloud_id, default_checklists: [...], ... }`
    - Template type is stored: `review.template_type = "standard" | "tender" | "friday"`
    - Default checklists are loaded from template's `default_checklists` array
2. **Checklist-Template Matching**:
    - **Template Structure**: Each template has a `default_checklists` array containing checklist document IDs
    - **Checklist Storage**: Checklists are stored in `myworkreview/checklists/list` collection
    - **Matching Process**:

        1. Template's `default_checklists` array contains checklist IDs (e.g., `["checklist-id-1", "checklist-id-2"]`)
        2. When creating a review, system fetches each checklist by ID from `myworkreview/checklists/list`
        3. Each checklist is copied to the review's `sections` array
        4. Checklist items are initialized with empty responses, unresolved status
        5. Review's `template.default_checklists` is updated to match template's default checklists
    - **Code Location**: `client/src/FileReviewCreateDialog.tsx` - `handleGetDefaultChecklists()` (lines 114-159)

3. **Default Checklists in Reviews**:

    - **On Review Creation**:
        - System calls `handleGetDefaultChecklists()` function
        - Fetches checklists from `template.default_checklists` array
        - Creates review with `sections` array populated with default checklists
        - Each checklist is copied with initialized section items (empty responses, unresolved)
    - **Checklist Initialization**:
        - Section items are initialized with:
            - `user: ""`
            - `email: ""`
            - `datetime: moment().format("DD/MM/YY HH:mm")`
            - `showResponses: true`
            - `responses: []`
            - `resolved: false`
            - `resolvedBy: ""`
            - `resolvedDate: ""`
    - **Template Tracking**: Review's `template.default_checklists` array tracks which checklists came from the template

4. **Adding Additional Checklists**:

    - **Process**:
        1. Users can add additional checklists to a review after creation
        2. System fetches checklist from `myworkreview/checklists/list` by ID
        3. Checklist is copied to review's `sections` array
        4. Review's `template.default_checklists` is updated to include new checklist ID
        5. New checklist is initialized with empty responses
    - **Permissions**:

        - Clerks: Cannot add checklists
        - Managers: Can add checklists
        - Partners: Can add and remove checklists
    - **Code Location**: `client/src/FileReviewSectionsView.tsx` - `handleAddChecklist()` (lines 213-303)

5. **Removing Checklists**:
    - **Process**:
        - Only Partners can remove checklists from reviews
        - System removes checklist from review's `sections` array
        - System removes checklist ID from `template.default_checklists` array
        - System checks if remaining checklists have `emailChecklist` flag and updates `hasEmailChecklist` field
    - **Permissions**: Only Partners can remove checklists
    - **Code Location**: `client/src/FileReviewSectionsView.tsx` - `handleDeleteChecklist()` (lines 305-375)

6. **Template Change**:
    - When template changes, new default checklists are loaded
    - **Process**:
        1. System compares old and new template's `default_checklists` arrays
        2. Identifies checklists that are in new template but not in review
        3. Fetches new checklists from `myworkreview/checklists/list`
        4. Adds new checklists to review's `sections` array via `arrayUnion()`
        5. Old checklists are preserved (not deleted)
        6. Review's `template.default_checklists` is updated
    - **Code Location**: `client/src/components/ReviewTemplateManage.tsx` - `handleGetDefaultChecklists()` (lines 113-164)

7. **Friday Template Linking**:
    - **Friday Templates** (`template_type === "friday"`) require a Friday engagement link
    - When selecting a Friday template, user must provide a Friday engagement document ID
    - Friday engagement ID is validated:
        - Checks if engagement exists in Friday Firestore
        - Checks if engagement is already linked to another review
        - Validates engagement ID format
    - Friday link is stored: `review.friday_link = "engagement-document-id"`
    - Friday engagement is updated: `friday_engagement.review_link = "review-document-id"`

8. **Friday Engagement Link Process**:
   - **Creating Review with Friday Template**:
     1. User selects Friday template
     2. User enters Friday engagement ID (or URL with ID extracted)
     3. System validates engagement exists: `friday/engagements/list/{engagementId}`
     4. System checks if engagement is already linked
     5. If valid, creates review with `friday_link` field
     6. Updates Friday engagement: `friday/engagements/list/{engagementId}.review_link = reviewId`
   - **Updating Review Template**:
     1. If changing from Friday to non-Friday template:
        - Removes `friday_link` from review
        - Removes `review_link` from Friday engagement
     2. If changing to different Friday engagement:
        - Removes old `review_link` from old engagement
        - Adds new `review_link` to new engagement
        - Updates review `friday_link` field
9. **Bidirectional Linking**:
   - Review → Friday: `review.friday_link = "engagement-id"`
   - Friday → Review: `friday_engagement.review_link = "review-id"`
   - Both links are maintained for cross-platform access
   - When link is removed, both sides are updated

Code Locations:

- Review Creation: `client/src/FileReviewCreateDialog.tsx` (lines 209-351)
- Template Management: `client/src/components/ReviewTemplateManage.tsx` (lines 207-286)
- Friday Link Validation: `client/src/FileReview.tsx` (lines 1087-1127)

Template Management (`TemplateManagement.tsx`):

- Create/edit templates
- Define sections
- Configure template properties

- Template type selection (standard, tender, etc.)

6. Team Management

Teams:

- Team creation and management
- Partner assignments
- User assignments
- Team-based review filtering

7. User Management

User Features:

- User creation and editing
- Role assignment
- Team assignment
- Status management (active/inactive)
- Profile management

Daily User Synchronization (`pubSub_dailyUserSync`):
The application automatically syncs users daily from Google Workspace via Google Apps Script.

Sync Process:

1. **Trigger**: Pub/Sub topic `daily-user-sync` (scheduled daily via Cloud Scheduler)
2. **API Call**:
   - URL: `https://script.google.com/macros/s/AKfycbyqQp6ZLZn_vSNi4dxZxWHa1yEoFEcKMoWunP3Mytx2sXL9N0g/exec`
   - Authentication: `USER_EMAIL_SYNC_KEY` passed as `Authorization` query parameter
   - Method: GET request
3. **Response Format**: Array of user objects:

```
[
  {
    name: "User Name",
    email: "user@example.com",
    photo: "https://photo-url.com/photo.jpg" | null
  }
]
```

4. **Sync Logic**:
    - **Users to Add**: Users in Google Apps Script but not in Firestore
        - Creates new user document with default role "A3KQ4WpdhLj2Vo2oSS2x" (Clerk)
        - Sets status to "active"
        - Initializes empty teams array
    - **Users to Remove**: Users in Firestore but not in Google Apps Script
        - Deletes user document from Firestore
        - Removes user from all teams (partners and users arrays)
        - Deletes user from Firebase Authentication if UID exists
    - **Users to Update**: Users with changed name or photo
        - Updates name and photo fields in Firestore
5. **Batch Operations**: Uses Firestore batch writes for efficiency
6. **Team Cleanup**: Automatically removes deleted users from all teams

Code Location: `functions/index.js` - `pubSub_dailyUserSync` function (lines 1254-1350)

8. Email Notifications

Notification Types:

1. **Review Creation**: Notifies team partners when a review is created
2. **Status Changes**: Notifies when review status changes (Open/Closed)
3. **Collaborator Changes**: Notifies when collaborators are added/removed
4. **Tender Deadlines**: Notifies about upcoming tender deadlines
5. **Weekly Checklists**: Sends weekly checklist PDF reports
6. **Custom Notifications**: Manual notifications from review interface

Email Configuration:

- Service: Gmail (nodemailer)
- From: `audit@l-inc.co.za`
- HTML templates with styling

9. PDF Report Generation

Weekly Checklist Reports:

- Generates PDF reports from checklist data
- Uses Puppeteer for PDF generation
- Handlebars templates for HTML rendering
- Email distribution to assigned users
- Cached PDF generation for performance

10. Google Drive Integration

Drive Features:

- File upload to Google Drive
- Folder structure creation
- File replacement/updates
- File retrieval with caching
- Attachment management

Drive Structure:

```
Parent Folder/
└── Review Folder/
    ├── Review PDF
    └── attachments/
        └── [Attachment files]
```

11. Search and Filtering

Review Filtering:

- Filter by status (Open/Closed)
- Filter by team
- Filter by template
- Filter by financial year
- Search by review name
- Priority reviews
- Archive/history views

12. Mobile Support

Mobile Features:

- Responsive design
- Mobile-optimized review cards
- Touch-friendly interactions
- Mobile navigation drawer
- Progressive Web App (PWA) support

13. Review Stages and Filtering

Review Stages (`Home.tsx`):

- **Active**: Currently open reviews
- **Priority**: User's priority reviews (stored in `priorityReviews` array)
- **History**: Closed/completed reviews
- **Office Active**: Office-level active reviews (if applicable)
- **Office History**: Office-level historical reviews (if applicable)
- **Archive**: Archived reviews

Review Filtering:

- Filter by status (Open/Closed)
- Filter by team
- Filter by template
- Filter by financial year
- Search by review name
- Hide empty reviews (reviews with no highlights/comments)
- Filter by flags
- Filter by tags

Review Actions:

- Archive/unarchive reviews
- Set priority reviews
- Add/remove flags
- Add/remove tags
- Set WIP (Work in Progress) values
- Set deadline dates (for tenders)
- Link to Friday engagements

14. Chat and Messaging System

Review Chat (`FlexUpChat.tsx`):

- Real-time chat for each review
- Message threading with replies
- File attachments in chat
- User mentions and notifications
- Chat history persistence
- Mobile swipe navigation between reviews

Friday Chat Integration (`FlexUpFridayChat.tsx`):

- Integrated chat with Friday engagements

- Cross-platform messaging
- Shared chat interface
- Friday engagement linking

FlexUp View (`FlexUpView.tsx`):

- Unified view of reviews and Friday engagements
- Side-by-side comparison
- Chat integration
- Review and engagement management
- Data grid with detail panels

Chat Features:

- Real-time message updates via Firestore listeners
- File uploads to Firebase Storage
- Message timestamps
- User avatars
- Reply-to functionality
- Message selection and actions
- Scroll-to-bottom functionality

15. AI Tool Integration

Guru the Robot (`AiTool.tsx`):

- AI-powered document analysis tool
- Integration with external AI service
- Opens PDF in AI analysis interface
- URL: `https://gurutherobot.web.app/#`
- Encodes Google Drive file ID for AI processing
- Accessible from review interface

Usage:

- Click AI icon on review to open in Guru
- Automatically formats Google Drive URL
- Opens in new tab/window

16. Settings and Configuration

Settings Sections (`SettingsHome.tsx`):

1. **Review Settings** (`SettingsFileReviewSettings.tsx`):

- Temporary review access period (days)
- Review flags management
- Tender flags management
- Flags managers assignment (who can set flags)
- Default flag configurations

2. **Templates Management** (`TemplatesHome.tsx`):

    - Create/edit review templates
    - Template types (standard, tender)
    - Section definitions
    - Checklist assignments
    - Template properties

3. **Checklists Management** (`ChecklistsHome.tsx`):

    - Create/edit checklists
    - Section items with questions
    - Response tracking setup
    - Email checklist configuration
    - Checklist templates

4. **Teams Management** (`TeamsHome.tsx`):

    - Create/edit teams
    - Assign partners to teams
    - Assign users to teams
    - Team-based review filtering
    - Team notifications

5. **Users Management** (`UsersHome.tsx`):

    - View all users
    - Edit user details
    - Assign roles
    - Assign teams
    - Set user status (active/inactive)
    - Note: Users are primarily synced from Google Apps Script

6. **Permissions Management** (`PermissionsHome.tsx`):

    - User group/role definitions
    - Permission configuration
    - Route-based permissions
    - Feature-based permissions

    - Role hierarchy

7. **Integrations** (`SettingsIntegrations.tsx`):
    - API key generation
    - Organization API keys
    - Friday integration settings
    - External service configurations

17. Flags and Tags System

Review Flags:

- Custom flags for review status
- Predefined flags: Open, Closed, Missed, Declined
- Custom flag creation in settings
- Flag-based filtering
- Flag managers (who can set flags)
- Tender-specific flags

Tags System:

- Custom tags for reviews
- Tag-based organization
- Tag filtering
- Multiple tags per review
- Tag management interface

18. Work in Progress (WIP) Values

WIP Management:

- Set WIP values for reviews
- Track work progress
- WIP-based filtering
- WIP value updates
- Integration with review workflow

19. Tender Management

Tender Features:

- Tender-specific review templates

- Deadline date tracking
- Tender flags (Open, Missed, Declined)
- Deadline notifications (7 days before, on deadline)
- Missed deadline tracking
- Tender details dialog
- Deadline date management

Tender Notifications:

- Daily checks for upcoming deadlines
- Notifications 7 days before deadline
- Notifications on deadline day
- Automatic "Missed" flag assignment
- Team partner notifications

20. Priority Reviews

Priority System:

- Users can mark reviews as priority
- Stored in user's `priorityReviews` array
- Priority tab in Home view
- Quick access to important reviews
- Priority indicator in review list

21. Section Reports

Report Generation (`SectionReport.tsx` , `GetSectionReport.tsx`):

- Generate reports for review sections
- PDF report export
- Section-specific data
- Report templates
- Report distribution

22. PDF Comparison Mode

Comparison Features (`PdfComparison.tsx`):

- Side-by-side PDF comparison
- Synchronized scrolling (toggle on/off)
- Comparison file selection from review attachments

- Highlight comparison (highlights visible on both PDFs)
- Comment comparison (general comments drawer)
- Toggle comparison mode
- Independent zoom controls for each PDF
- Page navigation for both PDFs

How It Works:

1. User opens comparison mode from review interface
2. Main PDF is loaded on the left side
3. User selects a comparison file from review attachments
4. Comparison PDF is loaded on the right side
5. Synchronized scrolling can be toggled on/off
6. When enabled, scrolling one PDF scrolls the other proportionally
7. General comments drawer is available for both PDFs
8. Highlights from the review are visible on both PDFs

Synchronized Scrolling:

- Uses `requestAnimationFrame` for smooth scrolling
- Calculates scroll position proportionally between PDFs
- Respects PDF boundaries (doesn't scroll beyond limits)
- Can be toggled on/off during comparison

Code Location: `client/src/PdfComparison.tsx`

23. ML Query Consolidation

ML Features (`FileReviewMLQueryConsol.tsx`):

- Management Letter (ML) query consolidation
- Query management interface
- ML-powered query suggestions
- Query response tracking
- Consolidates queries marked with `ml_query: true` flag
- Integrates with Friday RFI (Request for Information) system
- Displays queries from both MWR reviews and Friday engagements
- Query source indicators (MWR vs Friday)
- Copy consolidated queries to clipboard
- Filter queries by source (MWR or Friday)

How It Works:

1. Queries are marked with `ml_query: true` flag in highlights

2. ML Query Consolidation dialog displays all ML queries
3. Queries from Friday engagements are fetched via RFI collection
4. Queries are grouped and displayed in a consolidated view
5. Users can copy consolidated queries for external use

Code Location: `client/src/components/FileReviewMLQueryConsol.tsx`

25. General Comments

General Comments Feature:

- Comments that are not tied to specific PDF highlights
- Stored with `fileId: "general"` (no position data)
- Can be added from sidebar or comparison mode
- Support responses and attachments
- Can be resolved/unresolved like regular highlights
- Clerks cannot add general comments

How It Works:

1. Users can add general comments via sidebar or comparison mode
2. General comments are stored in highlights collection with `fileId: "general"`
3. Position data is empty (pageNumber: 0, no bounding rect)
4. General comments appear in sidebar under "General Comments" section
5. Support threaded responses and file attachments
6. Can be resolved/unresolved by Managers and Partners

Use Cases:

- Overall review comments
- General questions about the document
- Comments not specific to a PDF location
- High-level observations

Code Locations:

- General comment creation: `client/src/globalFunctions/FileReviewCalcs.tsx` - `addGeneralHighlight()` (line 172)
- General comments drawer: `client/src/PdfComparison.tsx` - `GeneralCommentsDrawer` (line 45)
- General comments sidebar: `client/src/Sidebar.tsx` (line 207)

26. Partial Resolution

Partial Resolution Feature:

- Highlights can be marked as "partially resolved" (not fully resolved)
- Different from full resolution (`resolved: true`)
- Tracks who created the partial resolution (`partialResolvedBy`)
- Tracks when partial resolution was created (`partialResolvedDate`)
- Only Partners can unresolve partially resolved highlights created by others
- Managers can unresolve their own partial resolutions

How It Works:

1. Managers or Partners can mark a highlight as partially resolved
2. System sets `partialResolved: true` , `partialResolvedBy: email` , `partialResolvedDate: timestamp`
3. Highlight remains unresolved (`resolved: false`) but shows as partially resolved
4. Only the creator of the partial resolution (or Partners) can unresolve it
5. Partial resolution can be upgraded to full resolution

Permission Rules:

- **Clerks**: Cannot resolve or unresolve highlights (including partial)
- **Managers**:
  - Can create partial resolutions
  - Can unresolve their own partial resolutions
  - Cannot unresolve partial resolutions created by others
- **Partners**:
  - Can create partial resolutions
  - Can unresolve any partial resolution (their own or others')

Code Location: `client/src/FileReviewSectionsView.tsx` - `canResolveHighlight()` and `canUnresolveHighlight()` (lines 997-1055)

27. Push Queries to Friday (RFI Integration)

RFI (Request for Information) Features:

- Push MWR review queries to Friday engagements as RFIs
- Creates RFI questions in Friday engagement
- Links MWR highlights to Friday RFI questions
- Supports file attachments in RFI questions
- Bidirectional integration between MWR and Friday

How It Works:

1. User selects highlights/queries from MWR review
2. User selects target Friday engagement
3. System creates RFI questions in Friday engagement
4. Each highlight becomes an RFI question
5. Files can be attached to RFI questions
6. RFI questions are linked back to MWR highlights via `friday_query: true` flag
7. RFI questions appear in Friday engagement's RFI collection

RFI Structure in Friday:

- Collection: `friday/engagements/list/{engagementId}/rfis`
- Each RFI question has:
  - Question text (from highlight comment)
  - Status (Open, Answered, Closed)
  - Linked MWR review ID
  - Linked MWR highlight ID
  - File attachments (if any)

Code Location: `client/src/components/PushQueriesToMyFridayDialog.tsx`

28. Offline Support

Progressive Web App (PWA):

- **Service Worker**: Registered in `serviceWorkerRegistration.ts`
- **Manifest**: `client/public/manifest.json` defines PWA metadata
- **Caching Strategies**: Workbox libraries for caching
- **Offline Data Access**: Firestore persistent cache (unlimited size)
- **Background Sync**: Workbox background sync for offline actions
- **Update Notifications**: Service worker update listener for new versions

PWA Configuration:

- **Short Name**: "My Review"
- **Display Mode**: Standalone
- **Theme Color**: #000000
- **Background Color**: #ffffff
- **Icons**: Multiple sizes (16x16 to 512x512)

Project Structure

```
myworkview/
├── client/                    # React frontend application
│   ├── public/               # Static assets
│   ├── src/
│   │   ├── components/       # Reusable components
│   │   ├── globalFunctions/  # Utility functions
│   │   ├── hooks/           # Custom React hooks
│   │   ├── lib/             # Type definitions and utilities
│   │   ├── style/           # Styled components and CSS
│   │   ├── App.tsx          # Main app component
│   │   ├── firebase.tsx     # Firebase configuration
│   │   └── ...
│   ├── package.json
│   └── tsconfig.json
│
├── functions/                # Firebase Cloud Functions
│   ├── index.js             # Main functions file
│   ├── collaboratorReviewChanges.js
│   ├── emailFunctions.js
│   ├── weeklyChecklistEmails.js
│   ├── userUtils.js
│   ├── serviceAccountKey.json
│   ├── serviceAccountKeyFriday.json
│   ├── serviceAccountCloud.json
│   └── package.json
│
├── firebase.json             # Firebase configuration
├── .firebaserc              # Firebase project aliases
├── server.js                # Local development server
└── package.json
```

Key Frontend Files

- `App.tsx` : Main application component, routing, authentication setup
- `FileReview.tsx` : PDF review interface (5280 lines)
- `FileReviewCreateDialog.tsx` : Review creation dialog
- `Home.tsx` : Review listing and management
- `firebase.tsx` : Firebase initialization and auth functions
- `AuthContext.tsx` : Global authentication context

Key Backend Files

- `functions/index.js` : Main Cloud Functions file (2043 lines)
- `collaboratorReviewChanges.js` : Collaborator change notifications
- `weeklyChecklistEmails.js` : Weekly checklist email generation
- `userUtils.js` : Friday user management utilities (for future use)
- `emailFunctions.js` : Email utility functions (currently empty)

Note: `server.js` in root directory is an outdated local development server and is not used in production. Cloud Functions in `functions/index.js` handle all backend operations.

UI Components and Styling

Frontend Component Overview

This section provides a comprehensive overview of all frontend components, their purpose, and why they exist in the application.

Main Application Components

Core Application Components

1. `App.tsx` - Main Application Component

   - **Purpose**: Root component that sets up routing, authentication context, and global state
   - **Reason**: Centralizes application initialization, manages global data (users, teams, templates), handles Friday token exchange, and provides context to all child components
   - **Key Features**: Lazy loading, service worker updates, offline detection, global snackbar notifications

2. `Login.tsx` - Authentication Component

   - **Purpose**: Handles user authentication via Google OAuth
   - **Reason**: Entry point for user access, validates user exists in Firestore before allowing access
   - **Key Features**: Google sign-in, user validation, error handling

3. `Home.tsx` - Main Dashboard

   - **Purpose**: Displays list of reviews with filtering, sorting, and management capabilities

- **Reason**: Primary interface for users to view and manage their reviews, provides review overview and quick actions
- **Key Features**: Data grid with detail panels, filtering by status/team/template/flags/tags, priority reviews, archive management, mobile-responsive cards

4. `FileReviewHome.tsx` - File Review Router

   - **Purpose**: Routes to different file review views (PDF viewer, sections, checklist)
   - **Reason**: Organizes file review functionality into separate views for better UX
   - **Key Features**: Nested routing, view switching

5. `FileReview.tsx` - PDF Review Interface

   - **Purpose**: Main PDF viewing and annotation interface
   - **Reason**: Core functionality of the app - allows users to view PDFs, create highlights, add comments, and manage review details
   - **Key Features**: PDF viewer, highlight system, comment threads, sidebar, zoom controls, comparison mode

6. `SettingsHome.tsx` - Settings Router

   - **Purpose**: Routes to different settings sections
   - **Reason**: Organizes administrative functions into accessible sections
   - **Key Features**: Accordion-based navigation, permission-based access

PDF and Annotation Components

7. `PdfHighlighter.tsx` - PDF Viewer and Highlighter

   - **Purpose**: Wraps PDF.js viewer and manages highlight rendering, selection, and interaction
   - **Reason**: Core component for PDF viewing and annotation functionality, handles coordinate transformations and highlight lifecycle
   - **Key Features**: PDF rendering, text/area selection, highlight rendering, scroll-to-highlight, zoom handling, resize observer

8. `Highlight.tsx` - Text Highlight Component

   - **Purpose**: Renders text highlights on PDF pages
   - **Reason**: Visual representation of text selections with color coding based on status and user role
   - **Key Features**: Multiple rect rendering, color classes (default/partner/resolved/scrolled), emoji display

9. `AreaHighlight.tsx` - Area Selection Component

- **Purpose**: Renders rectangular area highlights (non-text selections)
- **Reason**: Allows users to highlight areas of PDF without text selection, useful for images or diagrams
- **Key Features**: Resizable rectangles, screenshot storage, position tracking

10. `HighlightLayer.tsx` - Highlight Container

    - **Purpose**: Groups and renders highlights for a specific PDF page
    - **Reason**: Organizes highlights by page for efficient rendering and updates
    - **Key Features**: Page-specific highlight grouping, viewport coordinate transformation

11. `Popup.tsx` - Comment Popup

    - **Purpose**: Displays comment popup when clicking on highlights
    - **Reason**: Provides interface for viewing and interacting with highlight comments
    - **Key Features**: Comment display, emoji, user information

12. `Tip.tsx` - Comment Tip Component

    - **Purpose**: Shows comment input tooltip when creating new highlights
    - **Reason**: Provides inline comment creation interface during highlight creation
    - **Key Features**: Comment input, emoji picker, mobile keyboard handling

13. `TipContainer.tsx` - Tip Container

    - **Purpose**: Manages positioning and display of tip components
    - **Reason**: Handles tip positioning relative to PDF pages and viewport
    - **Key Features**: Position calculation, viewport-aware rendering

14. `MouseSelection.tsx` - Mouse Selection Handler

    - **Purpose**: Handles mouse drag selection for creating highlights
    - **Reason**: Captures user selection gestures and converts to highlight coordinates
    - **Key Features**: Mouse event handling, selection rectangle, coordinate calculation

15. `MouseMonitor.tsx` - Mouse Event Monitor

    - **Purpose**: Tracks mouse movements for selection and interaction
    - **Reason**: Provides mouse position data for highlight creation and interaction
    - **Key Features**: Mouse position tracking, event delegation

16. `PdfLoader.tsx` - PDF Loading Component

    - **Purpose**: Displays loading state while PDF is being fetched
    - **Reason**: Provides user feedback during PDF loading, especially for large files
    - **Key Features**: Loading spinner, progress indication

17. `PdfEditor.tsx` - PDF Editor Interface

    - **Purpose**: Allows users to draw annotations directly on PDF pages
    - **Reason**: Provides drawing/annotation tools for marking up PDFs with freehand drawings
    - **Key Features**: Canvas drawing, brush tool, undo/redo, page navigation, save annotations

18. `PdfComparison.tsx` - PDF Comparison View

    - **Purpose**: Displays two PDFs side-by-side for comparison
    - **Reason**: Allows users to compare different versions of documents or related documents
    - **Key Features**: Dual PDF viewer, synchronized scrolling, side-by-side layout

Sidebar and Review Management Components

19. `Sidebar.tsx` - Review Sidebar

    - **Purpose**: Displays review details, highlights list, sections, and review actions
    - **Reason**: Provides comprehensive review information and quick access to review features
    - **Key Features**: Highlights list with filtering, sections/checklist view, review details, tags/flags, mobile-responsive

20. `SidebarReviewDetails.tsx` - Review Details Panel

    - **Purpose**: Shows review metadata and properties
    - **Reason**: Displays review information in organized sections for quick reference
    - **Key Features**: Review name, status, team, template, dates, collaborators

21. `FileReviewMenu.tsx` - Review Menu Bar

    - **Purpose**: Provides toolbar with review actions and tools
    - **Reason**: Quick access to common review actions and tools
    - **Key Features**: Zoom controls, view options, AI tool, comparison mode, menu items

22. `FileReviewLinksBar.tsx` - Review Links Bar

    - **Purpose**: Displays Friday engagement link and related information
    - **Reason**: Shows integration with Friday platform and provides access to linked engagement
    - **Key Features**: Friday link display, engagement details, link validation

23. `FileReviewAttachmentBar.tsx` - Attachment Bar

    - **Purpose**: Displays and manages review attachments
    - **Reason**: Provides interface for viewing and managing file attachments associated with reviews
    - **Key Features**: Attachment list, upload, delete, preview

24. `FileReviewSectionsView.tsx` - Sections View

    - **Purpose**: Displays review sections/checklists in a dedicated view
    - **Reason**: Provides focused interface for managing checklist sections and responses
    - **Key Features**: Section list, item responses, file attachments, resolution status

Dialog and Form Components

25. `FileReviewCreateDialog.tsx` - Create Review Dialog

    - **Purpose**: Dialog for creating new reviews
    - **Reason**: Provides structured interface for review creation with template selection and file upload
    - **Key Features**: Template selection, file upload, team assignment, Friday link validation, tender details

26. `FileReviewCollaboratorDialog.tsx` - Collaborator Management Dialog

    - **Purpose**: Manages review collaborators (permanent and temporary)
    - **Reason**: Allows review owners to grant access to other users with expiration options
    - **Key Features**: Add/remove collaborators, temporary access periods, email notifications

27. `FileReviewFlagDialog.tsx` - Flag Management Dialog

    - **Purpose**: Manages review flags (status indicators)
    - **Reason**: Allows users to set custom flags for review organization and filtering
    - **Key Features**: Flag selection, custom flags, permission-based access

28. `FileReviewTagsDialog.tsx` - Tags Management Dialog

    - **Purpose**: Manages review tags for organization
    - **Reason**: Allows users to tag reviews for better organization and filtering
    - **Key Features**: Tag selection, multiple tags, tag creation

29. `FileReviewNotificationDialog.tsx` - Notification Dialog

    - **Purpose**: Sends custom notifications to review participants
    - **Reason**: Allows users to send targeted notifications about review updates
    - **Key Features**: Email selection, custom message, review link

30. `FileReviewChecklistDialog.tsx` - Checklist Management Dialog

    - **Purpose**: Manages checklist sections for reviews
    - **Reason**: Allows users to add, edit, and manage checklist sections
    - **Key Features**: Section management, item configuration, email checklist setup

31. `FileReviewTemplateChoice.tsx` - Template Selection Component
    - **Purpose**: Allows users to change review template
    - **Reason**: Provides interface for updating review template and Friday link
    - **Key Features**: Template selection, Friday link validation, checklist updates
32. `FileReviewDateDialog.tsx` - Date Selection Dialog
    - **Purpose**: Sets review dates (published date, deadline, etc.)
    - **Reason**: Allows users to set important dates for reviews, especially for tenders
    - **Key Features**: Date picker, multiple date fields, validation
33. `FileReviewValueDialog.tsx` - Value Input Dialog
    - **Purpose**: Sets review values (WIP, fee, etc.)
    - **Reason**: Allows users to set numeric values associated with reviews
    - **Key Features**: Number input, validation, multiple value types
34. `FileReviewInfoBubble.tsx` - Info Tooltip
    - **Purpose**: Displays informational tooltips
    - **Reason**: Provides contextual help and information to users
    - **Key Features**: Tooltip display, contextual information
35. `FileReviewTenderDetailsHolder.tsx` - Tender Details Component
    - **Purpose**: Displays and manages tender-specific details
    - **Reason**: Shows tender information like deadline dates and flags
    - **Key Features**: Deadline display, tender flags, deadline tracking
36. `FileReviewToolTip.tsx` - Tooltip Component
    - **Purpose**: Generic tooltip component
    - **Reason**: Provides reusable tooltip functionality across the application
    - **Key Features**: Position-aware tooltips, content display
37. `FileReviewMLQueryConsol.tsx` - ML Query Console
    - **Purpose**: Displays AI/ML query results and interactions
    - **Reason**: Provides interface for AI-powered document analysis queries
    - **Key Features**: Query display, results rendering, interaction interface

Chat Components

38. `FlexUpChat.tsx` - Review Chat Component

- **Purpose**: Real-time chat interface for reviews
- **Reason**: Enables communication between review participants within the review context
- **Key Features**: Real-time messaging, file attachments, Friday chat sync, message threading, mobile swipe navigation

39. `FlexUpFridayChat.tsx` - Friday Chat Component

    - **Purpose**: Chat interface for Friday engagements with review sync
    - **Reason**: Enables cross-platform chat between Friday engagements and linked reviews
    - **Key Features**: Friday chat display, review chat sync, message merging, bidirectional updates

40. `FlexUpView.tsx` - Unified Review/Engagement View

    - **Purpose**: Displays reviews and Friday engagements in a unified interface
    - **Reason**: Provides integrated view of MWR reviews and Friday engagements for cross-platform workflow
    - **Key Features**: Side-by-side comparison, chat integration, data grid with detail panels, flag management

41. `FlexUpReport.tsx` - Report View Component

    - **Purpose**: Displays report data for reviews/engagements
    - **Reason**: Provides reporting interface for review and engagement data
    - **Key Features**: Report generation, data visualization, export options

42. `GetFlexUpReport.tsx` - Report Generator Component

    - **Purpose**: Generates reports from review/engagement data
    - **Reason**: Creates formatted reports for reviews and engagements
    - **Key Features**: Report formatting, data aggregation, export functionality

Settings and Management Components

43. `SettingsFileReviewSettings.tsx` - Review Settings

    - **Purpose**: Manages review-related settings
    - **Reason**: Allows administrators to configure review behavior and defaults
    - **Key Features**: Temporary access periods, flag management, flag managers assignment

44. `TemplatesHome.tsx` - Templates Management Page

    - **Purpose**: Main page for template management
    - **Reason**: Provides interface for viewing and accessing template management
    - **Key Features**: Template list, navigation to template details

45. `TemplateManagement.tsx` - Template Editor
    - **Purpose**: Create and edit review templates
    - **Reason**: Allows administrators to define review structures and default checklists
    - **Key Features**: Template creation, section definitions, template type selection
46. `TemplateDetails.tsx` - Template Details View
    - **Purpose**: Displays template details and properties
    - **Reason**: Shows template information and allows editing
    - **Key Features**: Template properties, section list, edit functionality
47. `TemplateDetailsDialog.tsx` - Template Details Dialog
    - **Purpose**: Dialog for viewing/editing template details
    - **Reason**: Provides modal interface for template management
    - **Key Features**: Template editing, section management, save/cancel
48. `ChecklistsHome.tsx` - Checklists Management Page
    - **Purpose**: Main page for checklist management
    - **Reason**: Provides interface for viewing and accessing checklist management
    - **Key Features**: Checklist list, navigation to checklist details
49. `ChecklistsManagement.tsx` - Checklist Editor
    - **Purpose**: Create and edit checklists
    - **Reason**: Allows administrators to define checklist structures with questions and responses
    - **Key Features**: Checklist creation, section items, email checklist configuration
50. `ChecklistDetails.tsx` - Checklist Details View
    - **Purpose**: Displays checklist details and items
    - **Reason**: Shows checklist information and allows editing
    - **Key Features**: Checklist properties, item list, edit functionality
51. `TeamsHome.tsx` - Teams Management Page
    - **Purpose**: Main page for team management
    - **Reason**: Provides interface for viewing and accessing team management
    - **Key Features**: Team list, navigation to team details
52. `TeamsManagement.tsx` - Team Editor
    - **Purpose**: Create and edit teams
    - **Reason**: Allows administrators to define teams and assign partners/users

- **Key Features**: Team creation, partner assignment, user assignment

53. `TeamsDetails.tsx` - Team Details View

    - **Purpose**: Displays team details and members
    - **Reason**: Shows team information and allows editing
    - **Key Features**: Team properties, member list, edit functionality

54. `UsersHome.tsx` - Users Management Page

    - **Purpose**: Main page for user management
    - **Reason**: Provides interface for viewing and accessing user management
    - **Key Features**: User list, navigation to user details

55. `UsersManagement.tsx` - User Editor

    - **Purpose**: Create and edit users
    - **Reason**: Allows administrators to manage user accounts, roles, and teams
    - **Key Features**: User creation, role assignment, team assignment, status management

56. `UsersDetails.tsx` - User Details View

    - **Purpose**: Displays user details and properties
    - **Reason**: Shows user information and allows editing
    - **Key Features**: User properties, role display, team memberships, edit functionality

57. `PermissionsHome.tsx` - Permissions Management Page

    - **Purpose**: Main page for permission management
    - **Reason**: Provides interface for viewing and accessing permission management
    - **Key Features**: Permission list, navigation to permission details

58. `PermissionManagement.tsx` - Permission Editor

    - **Purpose**: Create and edit user groups/permissions
    - **Reason**: Allows administrators to define roles and their associated permissions
    - **Key Features**: Permission creation, role definition, permission configuration

59. `PermissionDetails.tsx` - Permission Details View

    - **Purpose**: Displays permission details and configuration
    - **Reason**: Shows permission information and allows editing
    - **Key Features**: Permission properties, role display, edit functionality

60. `SettingsIntegrations.tsx` - Integrations Settings

- **Purpose**: Manages integration settings and API keys
- **Reason**: Allows administrators to configure external integrations and API access
- **Key Features**: API key generation, Friday integration settings, external service configuration

Utility and Helper Components

61. `NavBar.tsx` - Navigation Bar
    - **Purpose**: Main navigation component
    - **Reason**: Provides consistent navigation across the application
    - **Key Features**: Menu items, user profile, logout, responsive design
62. `SideDrawer.tsx` - Side Drawer Navigation
    - **Purpose**: Mobile navigation drawer
    - **Reason**: Provides mobile-friendly navigation menu
    - **Key Features**: Drawer menu, navigation links, mobile optimization
63. `Loader.tsx` - Loading Spinner
    - **Purpose**: Displays loading state
    - **Reason**: Provides visual feedback during async operations
    - **Key Features**: Spinner animation, overlay display
64. `Spinner.tsx` - Spinner Component
    - **Purpose**: Circular loading indicator
    - **Reason**: Lightweight loading indicator for inline use
    - **Key Features**: Circular spinner, customizable size
65. `Splash.tsx` - Splash Screen
    - **Purpose**: Initial loading screen
    - **Reason**: Displays while application initializes
    - **Key Features**: Branding, loading indicator
66. `NoConnection.tsx` - Offline Indicator
    - **Purpose**: Displays when user is offline
    - **Reason**: Informs users about connectivity issues
    - **Key Features**: Offline message, retry functionality
67. `Forbidden.tsx` - Access Denied Page
    - **Purpose**: Displays when user lacks permission

- **Reason**: Provides clear feedback for unauthorized access attempts
- **Key Features**: Error message, navigation options

68. `ConfirmDialog.tsx` - Confirmation Dialog
    - **Purpose**: Generic confirmation dialog
    - **Reason**: Reusable confirmation interface for destructive actions
    - **Key Features**: Yes/No buttons, customizable message
69. `ConfirmDialogPassword.tsx` - Password Confirmation Dialog
    - **Purpose**: Password confirmation for sensitive actions
    - **Reason**: Adds extra security layer for critical operations
    - **Key Features**: Password input, validation
70. `UserAvatar.tsx` - User Avatar Component
    - **Purpose**: Displays user profile picture
    - **Reason**: Provides consistent user avatar display across the application
    - **Key Features**: Image display, fallback initials, size options
71. `StatusLabel.tsx` - Status Label Component
    - **Purpose**: Displays status indicators
    - **Reason**: Provides consistent status display (active/inactive, open/closed, etc.)
    - **Key Features**: Color coding, text display, icon support
72. `EmptyContainer.tsx` - Empty State Component
    - **Purpose**: Displays empty state messages
    - **Reason**: Provides user feedback when no data is available
    - **Key Features**: Empty state message, icon, action buttons
73. `CircularProgressLabel.tsx` - Circular Progress Indicator
    - **Purpose**: Displays progress with label
    - **Reason**: Shows progress percentage with descriptive text
    - **Key Features**: Circular progress, label display, percentage
74. `LinearProgressLabel.tsx` - Linear Progress Indicator
    - **Purpose**: Displays linear progress with label
    - **Reason**: Shows progress bar with descriptive text
    - **Key Features**: Linear progress bar, label display, percentage
75. `HomeLinearProgressLabel.tsx` - Home Progress Label

- **Purpose**: Progress indicator for home page
- **Reason**: Shows review completion progress on home page
- **Key Features**: Review-specific progress, home page styling

76. `FridayLinearProgressLabel.tsx` - Friday Progress Label

    - **Purpose**: Progress indicator for Friday engagements
    - **Reason**: Shows engagement completion progress
    - **Key Features**: Engagement-specific progress, Friday styling

77. `HomeTabs.tsx` - Home Page Tabs

    - **Purpose**: Tab navigation for home page sections
    - **Reason**: Organizes home page content into tabs (Active, Priority, History, etc.)
    - **Key Features**: Tab navigation, section switching

78. `SearchReviewField.tsx` - Search Input Component

    - **Purpose**: Search input for reviews
    - **Reason**: Provides search functionality for filtering reviews
    - **Key Features**: Search input, debounced search, placeholder text

Mobile Components

79. `MobileReviewCard.tsx` - Mobile Review Card

    - **Purpose**: Mobile-optimized review card display
    - **Reason**: Provides touch-friendly review cards for mobile devices
    - **Key Features**: Mobile layout, swipe gestures, compact design

80. `MobileEngagementCard.tsx` - Mobile Engagement Card

    - **Purpose**: Mobile-optimized Friday engagement card display
    - **Reason**: Provides touch-friendly engagement cards for mobile devices
    - **Key Features**: Mobile layout, swipe gestures, compact design

81. `MobileResizableRectangle.tsx` - Mobile Resizable Rectangle

    - **Purpose**: Resizable rectangle component for mobile
    - **Reason**: Provides touch-friendly resizing for area highlights on mobile
    - **Key Features**: Touch gestures, resize handles, mobile optimization

Friday Integration Components

82. `ReviewTemplateManage.tsx` - Review Template Manager

    - **Purpose**: Manages review template and Friday link
    - **Reason**: Allows users to change review template and validate Friday engagement links
    - **Key Features**: Template selection, Friday link validation, bidirectional linking

83. `ReviewChecklistManage.tsx` - Review Checklist Manager

    - **Purpose**: Manages checklist sections for a review
    - **Reason**: Allows users to add, edit, and configure checklist sections
    - **Key Features**: Section management, item configuration, email setup

84. `FridayTeamDialog.tsx` - Friday Team Dialog

    - **Purpose**: Displays Friday team information
    - **Reason**: Shows team details from Friday platform
    - **Key Features**: Team display, member list, Friday integration

85. `FridayDueDateDialog.tsx` - Friday Due Date Dialog

    - **Purpose**: Manages due dates for Friday engagements
    - **Reason**: Allows users to set and manage engagement due dates
    - **Key Features**: Date picker, validation, Friday sync

86. `PushQueriesToMyFridayDialog.tsx` - Push Queries Dialog

    - **Purpose**: Pushes review queries/highlights to Friday platform
    - **Reason**: Enables cross-platform query sharing between MWR and Friday
    - **Key Features**: Query selection, Friday push, synchronization

Attachment and File Components

87. `ResponseAttachment.tsx` - Response Attachment Component

    - **Purpose**: Displays attachments in checklist responses
    - **Reason**: Shows file attachments associated with checklist item responses
    - **Key Features**: File display, download, preview

88. `ResponseAttachmentLoad.tsx` - Response Attachment Loader

    - **Purpose**: Loads and displays response attachments
    - **Reason**: Handles attachment loading and display logic
    - **Key Features**: File loading, error handling, preview

89. `ResponseAttachmentPreview.tsx` - Response Attachment Preview

- **Purpose**: Preview component for response attachments
- **Reason**: Provides preview interface for attached files
- **Key Features**: File preview, download, delete

90. `FlexAttachmentPreview.tsx` - Flex Attachment Preview

    - **Purpose**: Preview component for FlexUp chat attachments
    - **Reason**: Provides preview interface for chat attachments
    - **Key Features**: File preview, download, chat context

91. `QuickViewAttachment.tsx` - Quick View Attachment

    - **Purpose**: Quick preview for attachments
    - **Reason**: Provides fast preview without full dialog
    - **Key Features**: Quick preview, modal display, file types

92. `ReviewAttachmentChoiceDialog.tsx` - Attachment Choice Dialog

    - **Purpose**: Dialog for selecting review attachments
    - **Reason**: Allows users to choose attachments from review files
    - **Key Features**: Attachment list, selection, preview

93. `AreaHighlightImageLoad.tsx` - Area Highlight Image Loader

    - **Purpose**: Loads images for area highlights
    - **Reason**: Handles image loading for area highlight screenshots
    - **Key Features**: Image loading, error handling, display

94. `ConvertDataUrlFile.tsx` - Data URL Converter

    - **Purpose**: Converts data URLs to files
    - **Reason**: Utility for converting base64 images to file objects
    - **Key Features**: Data URL parsing, file conversion

Report Components

95. `SectionReport.tsx` - Section Report Component

    - **Purpose**: Generates reports for review sections
    - **Reason**: Creates formatted reports for checklist sections
    - **Key Features**: Report generation, data formatting, export

96. `GetSectionReport.tsx` - Section Report Generator

    - **Purpose**: Generates section reports from data

    - **Reason**: Creates report data from section information
    - **Key Features**: Data aggregation, report formatting

Flag and Tag Components

97. `FlagHolder.tsx` - Flag Display Component

    - **Purpose**: Displays and manages review flags
    - **Reason**: Provides interface for viewing and setting review flags
    - **Key Features**: Flag display, flag selection, permission checks

98. `TagHolder.tsx` - Tag Display Component

    - **Purpose**: Displays and manages review tags
    - **Reason**: Provides interface for viewing and setting review tags
    - **Key Features**: Tag display, tag selection, tag creation

Other Utility Components

99. `ResponseChoiceBox.tsx` - Response Choice Input

    - **Purpose**: Input component for checklist responses
    - **Reason**: Provides structured input for checklist item responses
    - **Key Features**: Text input, file attachment, response submission

100. `Sections.tsx` - Sections Display Component
    - **Purpose**: Displays review sections/checklists
    - **Reason**: Shows checklist sections in organized view
    - **Key Features**: Section list, item display, response tracking

101. `TenderDetailsDialog.tsx` - Tender Details Dialog
    - **Purpose**: Manages tender-specific details
    - **Reason**: Allows users to set tender deadline dates and flags
    - **Key Features**: Deadline date, tender flags, validation

102. `RetryHighlightDialog.tsx` - Retry Highlight Dialog
    - **Purpose**: Dialog for retrying failed highlight operations
    - **Reason**: Provides recovery interface for failed highlight saves
    - **Key Features**: Retry functionality, error display

103. `HeaderText.tsx` - Header Text Component
    - **Purpose**: Reusable header text component

- **Reason**: Provides consistent header styling across the application
- **Key Features**: Typography, styling, size options

104. **`StyledTextField.tsx`** - Styled Text Field
    - **Purpose**: Custom styled text input component
    - **Reason**: Provides consistent text input styling
    - **Key Features**: Custom styling, validation, error display

Component Architecture

Main Pages:

- `Home.tsx` - Review list and management dashboard
- `Login.tsx` - Google OAuth login page
- `FileReview.tsx` - Main PDF review interface (5280 lines)
- `FileReviewHome.tsx` - File review routing wrapper
- `SettingsHome.tsx` - Settings routing wrapper
- `NavBar.tsx` - Navigation bar component

Reusable Components (`client/src/components/`):

- `FlexUpView.tsx` - Unified view for reviews and Friday engagements
- `FlexUpChat.tsx` - Chat interface for reviews
- `FlexUpFridayChat.tsx` - Chat interface for Friday engagements
- `PdfHighlighter.tsx` - PDF rendering and highlighting component
- `Highlight.tsx` - Text highlight component
- `AreaHighlight.tsx` - Area selection component
- `Popup.tsx` - Comment popup component
- `MobileReviewCard.tsx` - Mobile review card component
- `MobileEngagementCard.tsx` - Mobile Friday engagement card
- `FileReviewTagsDialog.tsx` - Tags management dialog
- `FileReviewValueDialog.tsx` - WIP value dialog
- `FileReviewDateDialog.tsx` - Deadline date dialog
- `FileReviewMLQueryConsol.tsx` - ML query consolidation component
- `StatusLabel.tsx` - Status label component
- `UserAvatar.tsx` - User avatar component
- `Loader.tsx` - Loading spinner component
- `EmptyContainer.tsx` - Empty state component
- `HomeTabs.tsx` - Home page tabs component
- `SearchReviewField.tsx` - Review search field

- `AiTool.tsx` - AI tool integration component

Code Location: `client/src/components/`

Styling System

Theme Configuration (`client/src/index.tsx`):

- **Font Family**: Poppins (sans-serif)
- **Primary Color**: `Colors.dark_blue` (#293241)
- **Secondary Color**: `Colors.darkest_purple` (#10002b)
- **Typography**: Custom font weights and sizes
- **Material-UI Theme**: Custom theme with component overrides

Color Palette (`client/src/style/styled.container.ts`):

- **Primary Colors**: Dark blue, purple variants
- **Status Colors**: Green (success), Orange (warning), Red (error)
- **Grey Scale**: Light to dark grey variants
- **Blue Scale**: Light to dark blue variants
- **Purple Scale**: Light to darkest purple variants

Styled Components:

- Uses `styled-components` library
- Custom styled components in `client/src/style/` directory
- Material-UI component overrides in theme
- CSS files for specific components (PDF viewer, highlights, etc.)

Styling Files:

- `styled.container.ts` - Container components and color palette
- `styled.flexup.tsx` - FlexUp view styled components
- `styled.navbar.ts` - Navigation bar styles
- `styled.login.ts` - Login page styles
- `styled.sidebar.tsx` - Sidebar styles
- `styled.settings.tsx` - Settings page styles
- `styled.templates.tsx` - Template management styles
- `styled.icons.tsx` - Icon styled components
- `App.css` - Global application styles
- Component-specific CSS files (Highlight.css, AreaHighlight.css, etc.)

Code Location: `client/src/style/`

UI Patterns

Data Grid:

- Uses Material-UI Data Grid Premium (`@mui/x-data-grid-premium`)
- Custom columns and cell renderers
- Inline editing support
- Filtering and sorting
- Row selection and actions
- Detail panels for expanded views

Dialogs:

- Material-UI Dialog components
- Custom styled dialogs with rounded corners
- Form dialogs for data entry
- Confirmation dialogs for destructive actions

Forms:

- Material-UI form components
- Autocomplete for user/team selection
- Date pickers for deadline dates
- Text fields with validation
- File upload components

Navigation:

- React Router for routing
- Protected routes with authentication checks
- Partner-only routes with role checks
- Navigation bar with user menu
- Breadcrumbs for nested routes

Responsive Design:

- Mobile-responsive components
- Breakpoint-based layouts
- Mobile-specific components (MobileReviewCard, MobileEngagementCard)
- Responsive data grid columns
- Touch-friendly interactions

Code Location: Throughout components using Material-UI and custom styled components

---

Backend Functions

HTTP Endpoints

All HTTP endpoints are available through two Cloud Functions:

- `api` : Standard HTTP API (default memory and timeout)
- `highMemoryApi` : High memory HTTP API (4GB memory, 540s timeout, max 10 instances)

Base URLs:

- Standard API: `https://us-central1-audit-7ec47.cloudfunctions.net/api`
- High Memory API: `https://us-central1-audit-7ec47.cloudfunctions.net/highMemoryApi`

Authentication:

- Server-to-server requests require `x-server-server` header and `Authorization: Bearer {MWR_SERVER_KEY}` header
- Client requests use CORS validation against allowed origins

Health Check

- **GET** `/api/hello` : Health check endpoint
  - Returns: `{ response: "Hello From My Review" }`
  - No authentication required

Authentication

- **POST** `/generate-custom-token` : Generate custom token for Friday integration
  - **Request Body**: `{ idToken: string }` (Friday ID token)
  - **Response**: `{ customToken: string }` (MWR custom token)
  - **Process**: Verifies Friday ID token, creates custom MWR token for cross-platform authentication
  - **Code Location**: `functions/index.js` (lines 196-213)

Review Management

- **POST** `/api/friday-get-reviews` : Get all reviews for Friday integration

  - **Request Body**: `{ idToken: string }` (Friday ID token)
  - **Response**: Array of review objects

    - **Process**: Verifies Friday token, retrieves all reviews from Firestore
    - **Code Location**: `functions/index.js` (lines 215-252)

- **POST** `/api/get-checklists` : Get all checklists
    - **Request Body**: `{ idToken: string }` (Friday ID token)
    - **Response**: Array of checklist objects
    - **Process**: Verifies Friday token, retrieves checklists from Firestore
    - **Code Location**: `functions/index.js` (lines 254-292)

- **POST** `/api/push-mwr-checklist` : Push checklist from Friday to MWR
    - **Request Body**: `{ idToken: string, checklist: object }`
    - **Response**: `{ success: boolean }`
    - **Process**: Verifies Friday token, creates/updates checklist in MWR Firestore
    - **Code Location**: `functions/index.js` (lines 294-342)

File Operations

- **POST** `/api/upload-file` : Upload review PDF to Google Drive
    - **Request**: Multipart form data with `file` and `fileName`
    - **Response**: `{ url: string, reviewFolderID: string, attachmentsFolderID: string }`
    - **Process**:
        1. Creates review folder in Google Drive parent folder
        2. Creates "attachments" subfolder
        3. Uploads PDF to review folder
        4. Returns Google Drive file IDs
    - **Code Location**: `functions/index.js` (lines 474-581)

- **POST** `/api/replace-drive-file` : Replace existing Drive file
    - **Request Body**: `{ fileId: string, fileBuffer: Buffer, fileName: string }`
    - **Response**: `{ url: string }` (new file ID)
    - **Process**: Deletes old file, uploads new file to same location
    - **Code Location**: `functions/index.js` (lines 645-717)

- **POST** `/api/fetch-review` : Fetch PDF from Google Drive (direct, no caching)
    - **Request Body**: `{ fileId: string }`
    - **Response**: PDF file stream

    - **Process**: Fetches file directly from Google Drive
    - **Code Location**: `functions/index.js` (lines 743-812)

- **POST `/api/fetch-review-cached`**: Fetch PDF with caching
    - **Request Body**: `{ fileId: string }`
    - **Response**: PDF file stream
    - **Process**:
        1. Checks Google Cloud Storage cache
        2. If cached and up-to-date, serves from cache
        3. If not cached or outdated, fetches from Drive and updates cache
    - **Cache**: Stored in `cached_reviews` bucket
    - **Cache TTL**: 24 hours (checks file modification time)
    - **Code Location**: `functions/index.js` (lines 814-870)

- **POST `/api/get-attachment-file`**: Get attachment file from Google Drive
    - **Request Body**: `{ fileId: string }`
    - **Response**: File stream with appropriate content type
    - **Process**: Fetches file from Google Drive by file ID
    - **Code Location**: `functions/index.js` (lines 872-916)

- **POST `/api/upload-attachment-file`**: Upload attachment to Drive
    - **Request**: Multipart form data with `file`, `fileName`, and `attachmentFolderID`
    - **Response**: `{ url: string }` (Google Drive file ID)
    - **Process**: Uploads file to specified attachment folder in Google Drive
    - **Code Location**: `functions/index.js` (lines 952-1040)

- **POST `/api/delete-attachment-file`**: Delete attachment from Drive
    - **Request Body**: `{ fileId: string }`
    - **Response**: `{ success: boolean }`
    - **Process**: Deletes file from Google Drive
    - **Code Location**: `functions/index.js` (lines 1042-1055)

Notifications

- **POST `/api/send-notification`**: Send custom notification email
    - **Request Body**: `{ reviewId: string, notificationText: string, emails: string[] }`
    - **Response**: `{ success: boolean }`

  - **Process**: Sends HTML email notification to specified recipients
  - **Code Location**: `functions/index.js` (lines 1057-1117)

- **POST `/api/send-bug-report`**: Send bug report email
  - **Request Body**: `{ email: string, message: string, reviewId?: string }`
  - **Response**: `{ success: boolean }`
  - **Process**: Sends bug report email to `audit@l-inc.co.za`
  - **Code Location**: `functions/index.js` (lines 918-950)

Cloud Functions (Pub/Sub Triggers)

All Pub/Sub functions are triggered by Cloud Scheduler jobs that publish messages to Pub/Sub topics.

1. **`pubSub_temporaryReviewAccessScheduler`**:
   - **Topic**: `temporary-access`
   - **Trigger**: Cloud Scheduler (typically daily)
   - **Purpose**: Removes expired temporary collaborators from reviews
   - **Process**:
     1. Iterates through all reviews in Firestore
     2. Filters collaborators by type (permanent vs temporary)
     3. Checks expiry date for temporary collaborators
     4. Removes expired temporary collaborators from `collaborators` array
     5. Removes expired collaborator cloud IDs from `collaboratorCloudIds` array
     6. Updates review document with cleaned collaborator lists
   - **Code Location**: `functions/index.js` (lines 1160-1232)
   - **Configuration**: Default memory and timeout

2. **`pubSub_dailyUserSync`**:
   - **Topic**: `daily-user-sync`
   - **Trigger**: Cloud Scheduler (daily at 2:00 AM Africa/Johannesburg)
   - **Purpose**: Synchronizes users from Google Workspace to Firestore
   - **Process**:
     1. Calls Google Apps Script API with `USER_EMAIL_SYNC_KEY` token
     2. Retrieves user list (name, email, photo) from Google Workspace
     3. Compares with existing Firestore users
     4. **Adds new users**: Creates new user documents with default role (Clerk), status (active)
     5. **Removes users**: Deletes users not in Google Workspace
        - Removes user from team memberships (partners and users arrays)
        - Deletes Firebase Auth account if exists

    6. **Updates users**: Updates name and photo for existing users if changed
    7. Uses Firestore batch writes (max 500 operations per batch)
  - **Authentication**: Uses `USER_EMAIL_SYNC_KEY` environment variable
  - **API Endpoint**: Google Apps Script URL (see Google Apps Script Integration section)
  - **Default Role**: Clerk (`A3KQ4WpdhLj2Vo2oSS2x`)
  - **Default Status**: `active`
  - **Code Location**: `functions/index.js` (lines 1254-1350)
  - **Configuration**: Default memory and timeout

3. **`pubsub_dailyBackup`** :

   - **Topic**: `daily-backup`
   - **Trigger**: Cloud Scheduler (daily at 3:00 AM Africa/Johannesburg)
   - **Purpose**: Creates daily Firestore database backup
   - **Process**:
     1. Exports all Firestore collections
     2. Stores backup in Google Cloud Storage bucket: `gs://mwrdailybackups`
     3. Logs backup operation to Firestore logs collection
     4. Returns operation name for tracking
   - **Backup Location**: `gs://mwrdailybackups`
   - **Backup Format**: Firestore export format
   - **Code Location**: `functions/index.js` (lines 1246-1252, 134-168)
   - **Configuration**: Default memory and timeout

4. **`pubSub_generateChecklistPDFEmails`** :

   - **Topic**: `generate-checklist-pdf`
   - **Trigger**: Cloud Scheduler (weekly on Mondays at 8:00 AM Africa/Johannesburg)
   - **Purpose**: Generates and emails weekly checklist PDF reports
   - **Process**:
     1. Queries reviews with `status == true` , `stage != "archive"` , `hasEmailChecklist == true`
     2. Filters sections with `emailChecklist == true`
     3. Fetches HTML template from `https://myworkreview.netlify.app/checklist_template.html`
     4. For each section:
        - Generates PDF using Puppeteer (headless Chrome)
        - Calculates days outstanding from published date
        - Sends email with PDF attachment to assigned users
     5. Processes sections concurrently (limit: 3 at a time)
     6. Adds 5-second delay between batches to prevent rate limiting

- **Memory**: 1GB
- **Timeout**: 540 seconds (9 minutes)
- **Concurrency**: 3 sections at a time
- **Code Location**: `functions/index.js` (lines 1912-2019)
- **Helper Function**: `functions/weeklyChecklistEmails.js` - `sendEmailsForSection()`

5. `checkTenderReviewNotificationsDue`:

    - **Topic**: `daily-tender-notifications`
    - **Trigger**: Cloud Scheduler (daily at 9:00 AM Africa/Johannesburg)
    - **Purpose**: Sends email notifications for tender reviews due within 7 days
    - **Process**:
        1. Queries tender reviews (`template.template_type == "tender"`) with:
            - `status == true` (Open)
            - `deadline_date >= today` and `<= today + 7 days`
            - `flags` array contains "Open"
        2. For each review:
            - Calculates days until deadline
            - Gets team partners from team document
            - Filters out inactive users
            - Sends email notification to team partners and reviewer
            - Email includes review name, deadline date, days until deadline, and link
    - **Email Recipients**: Team partners (active users only) + reviewer
    - **Code Location**: `functions/index.js` (lines 1378-1532)
    - **Configuration**: Default memory and timeout

6. `checkTenderReviewsMissedDeadline`:

    - **Topic**: `daily-tender-missed-deadline`
    - **Trigger**: Cloud Scheduler (daily at 10:00 AM Africa/Johannesburg)
    - **Purpose**: Updates review flags for missed tender deadlines
    - **Process**:
        1. Queries tender reviews with:
            - `template.template_type == "tender"`
            - `status == true` (Open)
            - `deadline_date == yesterday` (missed deadline)
        2. For each review:
            - Checks if flags already contain "Missed" or "Declined"
            - If not, updates flags to `["Missed"]`
    - **Timezone**: Africa/Johannesburg

- **Code Location**: `functions/index.js` (lines 1534-1593)
- **Configuration**: Default memory and timeout

Firestore Triggers

Firestore triggers automatically execute when documents are created or updated in Firestore.

1. **`newReviewerTriggerNotification`**:
   - **Trigger**: `myworkreview/reviews/list/{reviewId}` onCreate
   - **Purpose**: Sends email notification when a new review is created
   - **Process**:
     1. Waits 3 seconds for data consistency (all fields to be populated)
     2. Extracts review details (name, reviewer, team, template, status)
     3. Gets team partners from team document
     4. Filters out inactive users
     5. Sends HTML email notification to all active team partners
     6. Email includes review name, template, publisher, status, and link
   - **Email Recipients**: Team partners (active users only)
   - **Delay**: 3 seconds (for data consistency)
   - **Code Location**: `functions/index.js` (lines 1595-1680)
   - **Configuration**: Default memory and timeout

2. **`updatedReviewTriggerNotification`**:
   - **Trigger**: `myworkreview/reviews/list/{reviewId}` onUpdate
   - **Purpose**: Sends notifications and archives data when reviews are updated
   - **Process**:
     1. **Highlights Archival**:
        - Compares old and new highlights arrays
        - Identifies removed highlights
        - Archives removed highlights to `myworkreview/removedHighlights/list/{reviewId}`
        - Appends to existing archived highlights if document exists
     2. **Status Change Notifications**:
        - Detects if `status` field changed (Open/Closed)
        - Gets team partners from team document
        - Filters out inactive users
        - Sends HTML email notification to team partners and reviewer
        - Email includes review name, new status, and link
     3. **Collaborator Change Notifications**:
        - Compares old and new collaborators arrays

        - Compares old and new `collaboratorCloudIds` arrays
        - Calls `processCollaboratorChanges()` helper function
        - Sends email notifications for added/removed collaborators
    - **Email Recipients**: Team partners (active users only) + reviewer (for status changes)
    - **Code Location**: `functions/index.js` (lines 1682-1824)
    - **Helper Function**: `functions/collaboratorReviewChanges.js` - `processCollaboratorChanges()`
    - **Configuration**: Default memory and timeout

Function Configuration

Standard API:

- Memory: Default
- Timeout: Default
- Max Instances: Default

High Memory API:

- Memory: 4GB
- Timeout: 540 seconds (9 minutes)
- Max Instances: 10

Database Schema

Collections Structure

`myworkreview/users/list`

```
{
  email: string,                // Unique identifier
  name: string,
  photo: string | null,
  uid: string,                  // Firebase Auth UID
  role: string,                 // Reference to userGroup
  teams: string[],              // Array of team references
  status: "active" | "inactive",
  authProvider: "google",
  priorityReviews: string[] // Array of review IDs
}
```

myworkreview/userGroups/list

```
{
  name: string,
  roles: string[],           // Permission array
  description: string
}
```

myworkreview/teams/list

```
{
  name: string,
  partners: [{ email, name, photo }],
  users: [{ email, name, photo }]
}
```

myworkreview/templates/list

```
{
  name: string,
  template_type: "standard" | "tender",
  sections: [...],
  description: string
}
```

myworkreview/reviews/list

```
{
  groupName: string,
  financialYear: string,
  template: { cloud_id, name, template_type },
  team: { cloud_id, name },
  status: boolean,          // true = Open, false = Closed
  flags: string[],           // ["Open", "Closed", "Missed", "Declined"]
  collaborators: [{
    cloud_id: string,
    email: string,
    name: string,
    type: "permanent" | "temporary",
    expires: Timestamp | null
  }],
  collaboratorCloudIds: string[],
  sections: [...],            // Checklists/sections
  highlights: [...],           // PDF highlights
  fileUrl: string,             // Google Drive file ID
  attachmentFolderURL: string, // Google Drive folder ID
  attachments: [...],
  published: boolean,
  published_date: string,
  closed_date: string,
  deadline_date: string,      // For tenders
  tender_details: {...},
  email: string,               // Reviewer email
  user: string,               // Reviewer name
  firstManager: string,
  stage: "active" | "history" | "archive",
  hasEmailChecklist: boolean
}
```

myworkreview/checklists/list

```
  {
    name: string,
    section_items: [{
      id: string,
      question: string,
      responses: [{
        user: string,
        response: string,
        datetime: Timestamp,
        attachment: { url, name, type } | null
      }],
      resolved: boolean
    }]
  }
```

myworkreview/apis/list

```
  {
    key: string,                // API key
    name: string                // Organization name (collection name)
  }
```

Deployment

Frontend Deployment

Platform: Netlify
URL: `https://myworkreview.netlify.app`

Build Process:

1. Install dependencies: `npm install`
2. Build: `npm run build`
3. Optional obfuscation: `npm run build:prod` (includes obfuscation)

Build Scripts:

- `npm start` : Development server (runs on `http://localhost:3000`)
- `npm run build` : Production build
- `npm run build:prod` : Production build with JavaScript obfuscation

Code Obfuscation:

- Uses `javascript-obfuscator` package
- Obfuscates all JavaScript files in `build/static/js/`
- Configuration in `client/obfuscate.js`
- Features: control flow flattening, dead code injection, string array encoding (RC4)

Backend Deployment

Platform: Firebase Cloud Functions
Region: `us-central1`

Deployment Commands:

```
cd functions
npm install
firebase deploy --only functions
```

Function URLs:

- Main API: `https://us-central1-audit-7ec47.cloudfunctions.net/api`

- High Memory API: `https://us-central1-audit-7ec47.cloudfunctions.net/highMemoryApi`

Environment Setup

Required Files:

- `functions/serviceAccountKey.json` : MWR Firebase service account
- `functions/serviceAccountKeyFriday.json` : Friday Firebase service account
- `functions/serviceAccountCloud.json` : Google Cloud service account (for Drive API)

Environment Variables (`.env` in functions directory):

- `MWR_SERVER_KEY` : Server-to-server API key
- `AUDIT_EMAIL_PASS` : Gmail app password for notifications
- `USER_EMAIL_SYNC_KEY` : Authorization key for Google Apps Script user sync

Frontend Environment Variables (`.env` in client directory):

- `REACT_APP_MUI` : Material-UI Premium license key (required)
- `REACT_APP_FRIDAY_ROLE_PARTNER` : Friday role ID for Partner role
- `REACT_APP_FRIDAY_ROLE_MANAGER` : Friday role ID for Manager role

Environment Variables

Frontend

Environment Variables (`.env` in `client/` directory):

```
# Material-UI Premium License Key (required for @mui/x-data-grid-premium)
REACT_APP_MUI=your-mui-license-key-here

# Friday Role IDs (for role mapping in Friday integration)
REACT_APP_FRIDAY_ROLE_PARTNER=your-friday-partner-role-id
REACT_APP_FRIDAY_ROLE_MANAGER=your-friday-manager-role-id
```

Note: Firebase configuration is hardcoded in `client/src/firebase.tsx` and does not require environment variables.

Backend (`functions/.env`)

```
# Server-to-server API key
MWR_SERVER_KEY=your-api-key-here

# Gmail credentials for email notifications
AUDIT_EMAIL_PASS=your-gmail-app-password

# User sync API key (for Google Apps Script integration)
USER_EMAIL_SYNC_KEY=your-sync-key-here
```

Google Cloud Storage

- **Backup Bucket**: `gs://mwrdailybackups`
- **Cache Bucket**: `cached_reviews` (for PDF caching)

Google Drive Configuration

- **Parent Folder ID**: `1LZwInex0XVr6UKqyY_w6G2JeUyeUiiPG` (hardcoded in functions)
- **Drive API Scopes**: `https://www.googleapis.com/auth/drive.file`
- **Service Account**: Uses `serviceAccountCloud.json` for Drive API access

Google Apps Script Integration

User Email Synchronization:

- **Script URL**: `https://script.google.com/macros/s/AKfycbyqQp6ZLZn_vSNi4dxZxWHa1yEoFEcKMoWunP3Mytx2sXL9N0g/exec`
- **Authentication**:
  - Method: GET request
  - Authorization: `USER_EMAIL_SYNC_KEY` passed as query parameter: `?Authorization={USER_EMAIL_SYNC_KEY}`
  - Token stored in environment variable `USER_EMAIL_SYNC_KEY`
- **Purpose**: Syncs user email list from Google Workspace
- **Returns**: Array of user objects:

```
[
  {
      name: "User Name",       // Trimmed string
      email: "user@example.com", // Lowercase, trimmed
      photo: "https://photo-url.com/photo.jpg" | null // Trimmed or nu
    }
  ]
```

- **Used By**: `pubSub_dailyUserSync` Cloud Function
- **Frequency**: Daily synchronization via Cloud Scheduler
- **Error Handling**: Logs errors and continues with available users
- **Data Processing**:
  - Trims all string fields
  - Converts emails to lowercase
  - Handles null photo values
- **Sync Details**: See "Daily User Synchronization" section in Features for complete process flow

---

Setting Up a New Staging/Dev Environment

This section provides a comprehensive guide for setting up a new staging or development environment from scratch. This is essential when creating a separate environment for testing or development purposes.

Overview

Setting up a new environment requires:

1. Creating new Firebase projects (MWR and Friday)
2. Setting up Google Drive folders
3. Configuring Google Authentication
4. Setting up Firestore indexes
5. Configuring Storage buckets
6. Setting up environment variables
7. Creating service account keys
8. Configuring CORS
9. Setting up Google Apps Script (optional for dev)
10. Configuring Cloud Scheduler
11. Updating code configuration

Step 1: Create Firebase Projects

1.1 Create MWR Firebase Project

1. **Go to Firebase Console**: https://console.firebase.google.com/

2. **Create New Project**:

   - Click "Add project"
   - Enter project name (e.g., `mwr-staging` or `mwr-dev`)
   - Choose project ID (e.g., `mwr-staging-abc123`)
   - Disable Google Analytics (optional for dev)
   - Click "Create project"

3. **Enable Services**:

   - **Authentication**:
     - Go to Authentication > Sign-in method
     - Enable "Google" provider
     - Add authorized domains (your staging domain, localhost for dev)
     - Configure OAuth consent screen
   - **Firestore Database**:
     - Go to Firestore Database
     - Click "Create database"
     - Choose "Start in production mode" (security handled at application level)
     - Select location (e.g., `us-central1`)
   - **Storage**:
     - Go to Storage
     - Click "Get started"
     - Start in production mode
     - Select location (e.g., `us-central1`)
   - **Cloud Functions**:
     - Go to Functions
     - Enable billing (required for Cloud Functions)
     - Select location (e.g., `us-central1`)

4. **Get Firebase Config**:

   - Go to Project Settings > General
   - Scroll to "Your apps"
   - Click "Web" icon (`</>`)
   - Register app (e.g., "My Work Review Staging")
   - Copy Firebase configuration:

     ```
     {
       apiKey: "YOUR_API_KEY",
       authDomain: "YOUR_PROJECT_ID.firebaseapp.com",
     ```

```
    projectId: "YOUR_PROJECT_ID",
    storageBucket: "YOUR_PROJECT_ID.appspot.com",
    messagingSenderId: "YOUR_SENDER_ID",
    appId: "YOUR_APP_ID",
    measurementId: "YOUR_MEASUREMENT_ID" // Optional
  }
```

5. **Update Frontend Config**:
    - Edit `client/src/firebase.tsx`
    - Replace `firebaseConfig` with your new staging config:

```
const firebaseConfig = {
  apiKey: "YOUR_STAGING_API_KEY",
  authDomain: "YOUR_STAGING_PROJECT_ID.firebaseapp.com",
  projectId: "YOUR_STAGING_PROJECT_ID",
  storageBucket: "YOUR_STAGING_PROJECT_ID.appspot.com",
  messagingSenderId: "YOUR_STAGING_SENDER_ID",
  appId: "YOUR_STAGING_APP_ID",
};
```

1.2 Create Friday Firebase Project (If Using Friday Integration)

1. **Create Second Firebase Project**:
    - Follow same steps as MWR project
    - Name: `friday-staging` or `friday-dev`
    - Project ID: `friday-staging-xyz789`
2. **Enable Same Services**:
    - Authentication (Google provider)
    - Firestore Database
    - Storage
    - Cloud Functions
3. **Update Frontend Config**:
    - Edit `client/src/firebase.tsx`
    - Replace `friday_firebaseConfig` with your new Friday staging config:

```
const friday_firebaseConfig = {
  apiKey: "YOUR_FRIDAY_STAGING_API_KEY",
  authDomain: "YOUR_FRIDAY_STAGING_PROJECT_ID.firebaseapp.com",
  projectId: "YOUR_FRIDAY_STAGING_PROJECT_ID",
  storageBucket: "YOUR_FRIDAY_STAGING_PROJECT_ID.appspot.com",
  messagingSenderId: "YOUR_FRIDAY_STAGING_SENDER_ID",
  appId: "YOUR_FRIDAY_STAGING_APP_ID",
};
```

Step 2: Set Up Google Drive

2.1 Create Google Drive Parent Folder

1. **Create Parent Folder**:
    - Go to Google Drive
    - Create a new folder (e.g., "MWR Staging Reviews" or "MWR Dev Reviews")
    - Right-click folder > "Get link" > "Copy link"
    - Extract folder ID from URL: `https://drive.google.com/drive/folders/{FOLDER_ID}`
    - Save folder ID (e.g., `1ABC123xyz...`)
2. **Share Folder with Service Account**:
    - Right-click folder > "Share"
    - Add service account email (from `serviceAccountCloud.json`)
    - Grant "Editor" permission
    - Click "Send"

2.2 Update Code with New Folder ID

1. **Update Backend Code**:
    - Edit `functions/index.js`
    - Find line with parent folder ID (around line 517):

      ```
      parents: ["1LZwInex0XVr6UKqyY_w6G2JeUyeUiiPG"], // Production fo
      ```

    - Replace with your staging folder ID:

      ```
      parents: ["YOUR_STAGING_FOLDER_ID"], // Staging folder ID
      ```

2. **Verify Folder Structure**:
    - The application will create subfolders automatically
    - Structure: `Parent Folder > Review Folder > attachments/`

Step 3: Configure Google Authentication

3.1 Set Up OAuth Consent Screen

1. **Go to Google Cloud Console**:
   - Navigate to your Firebase project's Google Cloud Console
   - Go to "APIs & Services" > "OAuth consent screen"
2. **Configure OAuth Consent Screen**:
   - User Type: "Internal" (for organization) or "External" (for public)
   - App name: "My Work Review Staging"
   - User support email: Your email
   - Developer contact: Your email
   - Click "Save and Continue"
3. **Add Scopes**:
   - Click "Add or Remove Scopes"
   - Add required scopes:
     - `.../auth/userinfo.email`
     - `.../auth/userinfo.profile`
   - Click "Update" > "Save and Continue"
4. **Add Test Users** (if External):
   - Add test user emails
   - Click "Save and Continue"
5. **Summary**:
   - Review settings
   - Click "Back to Dashboard"

3.2 Configure Authorized Domains

1. **In Firebase Console**:
   - Go to Authentication > Settings > Authorized domains
   - Add domains:
     - `localhost` (for local development)
     - Your staging domain (e.g., `mwr-staging.netlify.app`)
     - Your custom domain (if applicable)

2. **In Google Cloud Console**:
    - Go to "APIs & Services" > "Credentials"
    - Click on your OAuth 2.0 Client ID
    - Add authorized JavaScript origins:
        - `http://localhost:3000` (for local dev)
        - `https://your-staging-domain.netlify.app`
    - Add authorized redirect URIs:
        - `http://localhost:3000` (for local dev)
        - `https://your-staging-domain.netlify.app`

Step 4: Set Up Firestore Indexes

4.1 Required Indexes

The application uses several Firestore queries that require composite indexes. Create these indexes:

1. **Users Collection**:
    - Collection: `myworkreview/users/list`
    - Index: `email` (ascending) - Single field index (auto-created)
2. **Reviews Collection**:
    - Collection: `myworkreview/reviews/list`
    - Indexes needed:
        - `status` (ascending), `stage` (ascending) - Composite index
        - `status` (ascending), `stage` (ascending), `hasEmailChecklist` (ascending) - Composite index
        - `team.cloud_id` (ascending), `status` (ascending) - Composite index
        - `user` (ascending), `status` (ascending) - Composite index
3. **Friday Engagements** (if using Friday):
    - Collection: `friday/engagements/list`
    - Indexes needed:
        - `user` (array-contains), `stage` (ascending), `status` (ascending) - Composite index
        - `team` (ascending), `stage` (ascending), `status` (ascending) - Composite index

4.2 Creating Indexes

Method 1: Via Firebase Console:

1. Go to Firestore Database > Indexes
2. Click "Create Index"
3. Select collection
4. Add fields and order
5. Click "Create"

Method 2: Via Firebase CLI:

1. Create `firestore.indexes.json`:

```
{
  "indexes": [
    {
      "collectionGroup": "list",
      "queryScope": "COLLECTION",
      "fields": [
        {
          "fieldPath": "status",
          "order": "ASCENDING"
        },
        {
          "fieldPath": "stage",
          "order": "ASCENDING"
        }
      ]
    },
    {
      "collectionGroup": "list",
      "queryScope": "COLLECTION",
      "fields": [
        {
          "fieldPath": "status",
          "order": "ASCENDING"
        },
        {
          "fieldPath": "stage",
          "order": "ASCENDING"
        },
        {
          "fieldPath": "hasEmailChecklist",
          "order": "ASCENDING"
        }
      ]
    }
  ],
  "fieldOverrides": []
}
```

2. Deploy indexes:

```
firebase deploy --only firestore:indexes
```

Method 3: Let Firestore Create Automatically:

- Firestore will prompt you to create indexes when queries fail
- Click the link in the error message
- Firestore Console will open with pre-filled index configuration
- Click "Create Index"

Step 5: Set Up Storage Buckets

5.1 Create Storage Buckets

1. **Firebase Storage Bucket** (Auto-created):
   - Bucket name: `{project-id}.appspot.com`
   - Already created when you enabled Storage
   - No additional setup needed
2. **Google Cloud Storage Buckets** (For caching and backups):

   Cache Bucket:

   ```
   gsutil mb -p YOUR_PROJECT_ID -c STANDARD -l us-central1 gs://cached_r
   ```

   - Bucket name: `cached_reviews_staging` (or `cached_reviews_dev`)
   - Location: `us-central1` (or your preferred region)
   - Storage class: `STANDARD`

   Backup Bucket:

   ```
   gsutil mb -p YOUR_PROJECT_ID -c STANDARD -l us-central1 gs://mwr_stag
   ```

   - Bucket name: `mwr_staging_backups` (or `mwr_dev_backups`)
   - Location: `us-central1`
   - Storage class: `STANDARD`

3. **Update Code with Bucket Names**:

   - Edit `functions/index.js`
   - Find `REVIEW_BUCKET_NAME` (around line 37):

     ```
     const REVIEW_BUCKET_NAME = 'cached_reviews'; // Production
     ```

   - Replace with staging bucket:

     ```
     const REVIEW_BUCKET_NAME = 'cached_reviews_staging'; // Staging
     ```

   - Find backup bucket (around line 33):

     ```
     const bucket = "gs://mwrdailybackups"; // Production
     ```

   - Replace with staging bucket:

     ```
     const bucket = "gs://mwr_staging_backups"; // Staging
     ```

5.2 Set Bucket Permissions

1. **Grant Service Account Access**:

   ```
   gsutil iam ch serviceAccount:YOUR_SERVICE_ACCOUNT_EMAIL:roles/storage
   gsutil iam ch serviceAccount:YOUR_SERVICE_ACCOUNT_EMAIL:roles/storage
   ```

Step 6: Create Service Account Keys

6.1 Create Service Accounts

1. **MWR Firebase Service Account**:

   - Go to Google Cloud Console > IAM & Admin > Service Accounts
   - Click "Create Service Account"
   - Name: `mwr-functions-service`
   - Description: "Service account for MWR Cloud Functions"
   - Click "Create and Continue"
   - Grant roles:
     - "Firebase Admin SDK Administrator Service Agent"
     - "Cloud Datastore User"
     - "Storage Admin"
   - Click "Continue" > "Done"

2. **Friday Firebase Service Account** (if using Friday):
    - Create similar service account for Friday project
    - Name: `friday-functions-service`
3. **Google Cloud Service Account** (for Drive API):
    - Create service account in MWR project
    - Name: `mwr-drive-service`
    - Grant roles:
        - "Storage Admin" (for GCS buckets)
        - "Firestore Admin" (for backups)

6.2 Generate Service Account Keys

1. **MWR Firebase Key**:
    - Go to Service Accounts
    - Click on `mwr-functions-service`
    - Go to "Keys" tab
    - Click "Add Key" > "Create new key"
    - Choose "JSON"
    - Download and save as `functions/serviceAccountKey.json`
2. **Friday Firebase Key**:
    - Repeat for Friday service account
    - Save as `functions/serviceAccountKeyFriday.json`
3. **Google Cloud Key** (for Drive API):
    - Repeat for Drive service account
    - Save as `functions/serviceAccountCloud.json`
4. **Enable Google Drive API**:
    - Go to "APIs & Services" > "Library"
    - Search "Google Drive API"
    - Click "Enable"
    - Grant Drive API access to service account
5. **Share Drive Folder with Service Account**:
    - Go to Google Drive
    - Right-click your staging parent folder
    - Click "Share"

- Add service account email (from `serviceAccountCloud.json`)
- Grant "Editor" permission

Step 7: Set Up Environment Variables

7.1 Backend Environment Variables

1. **Create `.env` file in `functions/` directory**:

   ```
   # Server-to-server API key (generate a secure random string)
   MWR_SERVER_KEY=your-secure-random-api-key-here

   # Gmail app password for email notifications
   # Generate at: https://myaccount.google.com/apppasswords
   AUDIT_EMAIL_PASS=your-gmail-app-password

   # User sync API key (for Google Apps Script integration)
   # Generate a secure random string
   USER_EMAIL_SYNC_KEY=your-secure-sync-key-here
   ```

2. **Generate Secure Keys**:

   ```
   # Generate MWR_SERVER_KEY (32+ characters)
   openssl rand -hex 32

   # Generate USER_EMAIL_SYNC_KEY (32+ characters)
   openssl rand -hex 32
   ```

3. **Set Gmail App Password**:
   - Go to https://myaccount.google.com/apppasswords
   - Sign in with Gmail account (`audit@l-inc.co.za` or your staging email)
   - Select "Mail" and "Other (Custom name)"
   - Enter name: "MWR Staging"
   - Click "Generate"
   - Copy 16-character password
   - Add to `.env` as `AUDIT_EMAIL_PASS`

4. **Set Environment Variables in Firebase** (for deployed functions):

```
firebase functions:config:set mwr.server_key="YOUR_SERVER_KEY"
firebase functions:config:set audit.email_pass="YOUR_EMAIL_PASS"
firebase functions:config:set user.email_sync_key="YOUR_SYNC_KEY"
```

Note: For Firebase Functions v2, use `.env` file or Secret Manager:

```
# Using Secret Manager (recommended)
echo -n "YOUR_SERVER_KEY" | gcloud secrets create mwr-server-key --da
echo -n "YOUR_EMAIL_PASS" | gcloud secrets create audit-email-pass --
echo -n "YOUR_SYNC_KEY" | gcloud secrets create user-email-sync-key -
```

7.2 Frontend Environment Variables

1. **Create `.env` file in `client/` directory**:

```
# Material-UI Premium License Key (required)
REACT_APP_MUI=your-mui-license-key-here

# Friday Role IDs (if using Friday integration)
REACT_APP_FRIDAY_ROLE_PARTNER=your-friday-partner-role-id
REACT_APP_FRIDAY_ROLE_MANAGER=your-friday-manager-role-id
```

2. **Get Material-UI License Key**:
   - Go to https://mui.com/store/
   - Purchase or use existing license
   - Copy license key

3. **Get Friday Role IDs** (if using Friday):
   - Go to Friday Firestore: `friday/userGroups/list`
   - Find Partner and Manager role documents
   - Copy their `cloud_id` values

Step 8: Configure CORS

8.1 Update Allowed Origins

1. **Edit `functions/index.js`**:
   - Find `allowedOrigins` array (around line 75):

```
const allowedOrigins = [
  "http://localhost:3000",
  "https://localhost:3000",
  "localhost:3000",
  "https://www.myworkreview.netlify.app",
  // ... other origins
];
```

- Add your staging origins:

```
const allowedOrigins = [
  "http://localhost:3000",
  "https://localhost:3000",
  "localhost:3000",
  "https://www.myworkreview.netlify.app", // Production
  "https://mwr-staging.netlify.app", // Staging
  "https://mwr-dev.netlify.app", // Dev
  // ... other origins
];
```

Step 9: Set Up Google Apps Script (Optional for Dev)

9.1 Create Google Apps Script

1. **Create New Script**:
    - Go to https://script.google.com/
    - Click "New project"
    - Name: "MWR User Sync Staging"
2. **Write Script**:

```
function doGet(e) {
  const authKey = e.parameter.Authorization;
  const expectedKey = 'YOUR_USER_EMAIL_SYNC_KEY'; // From .env

  if (authKey !== expectedKey) {
    return ContentService.createTextOutput(JSON.stringify({ error: '
      .setMimeType(ContentService.MimeType.JSON);
  }

  // Get users from Google Workspace
```

```
    const users = AdminDirectory.Users.list({
      domain: 'your-domain.com',
      maxResults: 500
    });

    const userList = users.users.map(user => ({
      name: user.name.fullName.trim(),
      email: user.primaryEmail.toLowerCase().trim(),
      photo: user.thumbnailPhotoUrl ? user.thumbnailPhotoUrl.trim() :
    }));

    return ContentService.createTextOutput(JSON.stringify(userList))
      .setMimeType(ContentService.MimeType.JSON);
  }
```

3. **Enable Admin SDK**:
   - Go to "Services" > "Add a service"
   - Add "Admin SDK API"
   - Enable
4. **Deploy as Web App**:
   - Click "Deploy" > "New deployment"
   - Type: "Web app"
   - Execute as: "Me"
   - Who has access: "Anyone"
   - Click "Deploy"
   - Copy web app URL
5. **Update Code with Script URL**:
   - Edit `functions/index.js`
   - Find `getEmailList()` function (around line 170)
   - Update URL:

     ```
     const response = await axios.get(
       'YOUR_STAGING_SCRIPT_URL',
       {
         params: {
           Authorization: process.env.USER_EMAIL_SYNC_KEY
         }
       }
     );
     ```

Note: For local development, you can skip this step and manually create users in Firestore.

Step 10: Set Up Cloud Scheduler

10.1 Create Scheduled Jobs

1. **Daily User Sync**:

```
gcloud scheduler jobs create pubsub daily-user-sync-staging \
  --schedule="0 2 * * *" \
  --topic=daily-user-sync \
  --message-body='{"environment":"staging"}' \
  --time-zone="Africa/Johannesburg"
```

2. **Daily Backup**:

```
gcloud scheduler jobs create pubsub daily-backup-staging \
  --schedule="0 3 * * *" \
  --topic=daily-backup \
  --message-body='{"environment":"staging"}' \
  --time-zone="Africa/Johannesburg"
```

3. **Daily Tender Notifications**:

```
gcloud scheduler jobs create pubsub daily-tender-notifications-stagi
  --schedule="0 9 * * *" \
  --topic=daily-tender-notifications \
  --message-body='{"environment":"staging"}' \
  --time-zone="Africa/Johannesburg"
```

4. **Daily Tender Missed Deadline**:

```
gcloud scheduler jobs create pubsub daily-tender-missed-deadline-sta
  --schedule="0 10 * * *" \
  --topic=daily-tender-missed-deadline \
  --message-body='{"environment":"staging"}' \
  --time-zone="Africa/Johannesburg"
```

5. **Weekly Checklist PDF Emails**:

```
gcloud scheduler jobs create pubsub generate-checklist-pdf-staging \
  --schedule="0 8 * * 1" \
  --topic=generate-checklist-pdf \
  --message-body='{"environment":"staging"}' \
  --time-zone="Africa/Johannesburg"
```

Step 11: Initialize Firestore Data

11.1 Create Initial Collections

1. **Create Collection Structure**:
   - Go to Firestore Database
   - Create collections:
     - `myworkreview/users/list`
     - `myworkreview/userGroups/list`
     - `myworkreview/teams/list`
     - `myworkreview/templates/list`
     - `myworkreview/reviews/list`
     - `myworkreview/checklists/list`
     - `myworkreview/logs/list`
     - `myworkreview/apis/list`
     - `myworkreview/settings/list`
     - `myworkreview/removedHighlights/list`
2. **Create Default User Groups**:
   - Create document in `myworkreview/userGroups/list`
   - Document ID: `A3KQ4WpdhLj2Vo2oSS2x` (or generate new ID)
   - Data:

     ```
     {
       "name": "Clerk",
       "roles": {
         "reviews": {
           "mark_all_resolved": false,
           "change_status": false
         }
       },
       "description": "Basic user role"
     }
     ```

  - Create similar documents for "Manager" and "Partner" roles

3. **Create Settings Document**:

   - Create document in `myworkreview/settings/list`
   - Document ID: `settings` (or your preferred ID)
   - Data:

     ```
     {
       "temporary_review_access_period": 30,
       "flags_managers": [],
       "tender_flags": ["Open", "Missed", "Declined"]
     }
     ```

4. **Create Test User**:

   - Create document in `myworkreview/users/list`
   - Data:

     ```
     {
       "email": "your-email@example.com",
       "name": "Your Name",
       "photo": null,
       "uid": "",
       "role": "A3KQ4WpdhLj2Vo2oSS2x", // Reference to user group
       "teams": [],
       "status": "active",
       "authProvider": "google",
       "priorityReviews": []
     }
     ```

Step 12: Deploy and Test

12.1 Deploy Backend

1. **Install Dependencies**:

   ```
   cd functions
   npm install
   ```

2. **Deploy Functions**:

```
firebase deploy --only functions
```

3. **Verify Deployment**:
   - Check Firebase Console > Functions
   - Verify all functions are deployed
   - Check function logs for errors

12.2 Deploy Frontend

1. **Install Dependencies**:

```
cd client
npm install
```

2. **Build**:

```
npm run build
```

3. **Deploy to Netlify** (or your hosting):
   - Connect repository to Netlify
   - Set build command: `cd client && npm run build`
   - Set publish directory: `client/build`
   - Add environment variables in Netlify dashboard:
     - `REACT_APP_MUI`
     - `REACT_APP_FRIDAY_ROLE_PARTNER` (if applicable)
     - `REACT_APP_FRIDAY_ROLE_MANAGER` (if applicable)

12.3 Test Environment

1. **Test Authentication**:
   - Visit staging URL
   - Click "Login with Google"
   - Verify OAuth flow works
   - Check if user is created in Firestore
2. **Test File Upload**:
   - Create a test review
   - Upload a PDF

- Verify file appears in Google Drive folder
- Verify file can be viewed in app

3. **Test Cloud Functions**:

   - Check function logs in Firebase Console
   - Test API endpoints manually
   - Verify email notifications work

4. **Test Integrations**:

   - Test Friday integration (if applicable)
   - Test Google Apps Script sync (if applicable)
   - Verify scheduled jobs run

Step 13: Environment-Specific Configuration

13.1 Create Environment Config File

1. **Create** `client/src/config.ts`:

```
const config = {
  environment: process.env.REACT_APP_ENV || 'development',
  apiUrl: process.env.REACT_APP_API_URL || 'https://us-central1-YOUR_
  // Add other environment-specific configs
};

export default config;
```

2. **Use in Code**:

```
import config from './config';

const apiUrl = config.apiUrl;
```

13.2 Separate Firebase Projects

Recommended Approach:

- **Production**: `audit-7ec47` (MWR), `friday-a372b` (Friday)
- **Staging**: `mwr-staging-abc123` (MWR), `friday-staging-xyz789` (Friday)
- **Development**: `mwr-dev-123456` (MWR), `friday-dev-789012` (Friday)

Use Firebase Project Aliases:

1. **Create `.firebaserc`**:

   ```
   {
     "projects": {
       "default": "audit-7ec47",
       "production": "audit-7ec47",
       "staging": "mwr-staging-abc123",
       "dev": "mwr-dev-123456"
     }
   }
   ```

2. **Deploy to Specific Environment**:

   ```
   firebase use staging
   firebase deploy --only functions
   ```

Step 14: Security Checklist

- [ ] Service account keys are not committed to version control
- [ ] `.env` files are in `.gitignore`
- [ ] Environment variables are set in deployment platform
- [ ] Google Drive folder permissions are correct
- [ ] OAuth consent screen is configured
- [ ] Authorized domains are set
- [ ] CORS origins are restricted
- [ ] API keys are secure and rotated regularly
- [ ] Firestore indexes are created
- [ ] Storage bucket permissions are correct
- [ ] Cloud Scheduler jobs are configured
- [ ] Backup bucket exists and is accessible

Step 15: Documentation Updates

1. **Update README**:
   - Document staging environment URLs
   - Document staging-specific configurations
   - Update environment variable examples

2. **Create Environment-Specific Docs**:
    - Create `STAGING_SETUP.md` with staging-specific notes
    - Document any staging-specific workarounds
    - Note any differences from production

Troubleshooting New Environment

Common Issues:

1. **"Permission denied" errors**:
    - Check service account permissions
    - Verify Google Drive folder sharing
    - Check Firestore rules (if any)
2. **"CORS policy" errors**:
    - Verify origin in `allowedOrigins` array
    - Check browser console for exact origin
    - Update CORS configuration
3. **"Missing or insufficient permissions"**:
    - Check Firestore indexes
    - Verify service account roles
    - Check Storage bucket permissions
4. **Authentication failures**:
    - Verify OAuth consent screen configuration
    - Check authorized domains
    - Verify redirect URIs
5. **Google Drive upload failures**:
    - Verify Drive API is enabled
    - Check service account has Drive access
    - Verify folder sharing permissions

Code Locations for Environment Setup:

- Firebase config: `client/src/firebase.tsx`
- Google Drive folder ID: `functions/index.js` (line 517)
- CORS origins: `functions/index.js` (line 75)

- Bucket names: `functions/index.js` (lines 33, 37)
- Environment variables: `functions/.env`, `client/.env`

Development Setup

Prerequisites

- Node.js 20+ (for functions)
- npm or yarn
- Firebase CLI: `npm install -g firebase-tools`
- Firebase account with project access
- Google Cloud account for Drive API

Local Development

1. **Clone Repository**:

   ```
   git clone <repository-url>
   cd myworkview
   ```

2. **Install Frontend Dependencies**:

   ```
   cd client
   npm install
   ```

3. **Install Backend Dependencies**:

   ```
   cd ../functions
   npm install
   ```

4. **Configure Firebase**:

   - Copy service account JSON files to `functions/` directory:
     - `serviceAccountKey.json` (MWR Firebase)
     - `serviceAccountKeyFriday.json` (Friday Firebase)
     - `serviceAccountCloud.json` (Google Cloud for Drive API)
   - Create `.env` file in `functions/` with required variables (see Environment Variables section)

5. **Configure Frontend Environment**:
   - Create `.env` file in `client/` directory
   - Add `REACT_APP_MUI` with your Material-UI Premium license key
   - Add `REACT_APP_FRIDAY_ROLE_PARTNER` and `REACT_APP_FRIDAY_ROLE_MANAGER` if using Friday integration
6. **Start Frontend Development Server**:

   ```
   cd client
   npm start
   ```

   - Runs on `http://localhost:3000`
7. **Start Backend Locally** (optional):

   ```
   cd functions
   firebase emulators:start --only functions
   ```

Firebase Emulators

To run Firebase emulators locally:

```
firebase emulators:start
```

This starts:

- Firestore emulator
- Functions emulator
- Authentication emulator (if configured)

Testing

Frontend Tests:

```
cd client
npm test
```

Progressive Web App (PWA) Features

Overview

The application is a **Progressive Web App (PWA)** with offline support and service worker functionality.

PWA Configuration

Manifest (`client/public/manifest.json`):

- **Short Name**: "My Review"
- **Display Mode**: `standalone` (app-like experience)
- **Theme Color**: `#000000`
- **Background Color**: `#ffffff`
- **Icons**: Multiple sizes (16x16, 24x24, 32x32, 64x64, 192x192, 512x512)
- **Start URL**: `.` (root)

Service Worker

Service Worker (`client/src/service-worker.ts`):

- Uses **Workbox** for service worker management
- **Precaching**: All build assets are precached
- **App Shell Routing**: Navigation requests fulfilled with `index.html`
- **Runtime Caching**:
  - Images: `StaleWhileRevalidate` strategy (max 50 entries)
  - Firebase Storage: `StaleWhileRevalidate` strategy (30-day cache, max 50 entries)
  - PDF Fetch: `NetworkOnly` strategy (always fetch fresh)

Caching Strategies:

1. **Precache**: All static assets from build
2. **StaleWhileRevalidate**: Images and Firebase Storage files
3. **NetworkOnly**: PDF fetch endpoint (always fresh)

Service Worker Registration (`client/src/serviceWorkerRegistration.ts`):

- Registers service worker in production only
- Handles update notifications
- Auto-reloads on update ready

Update Listener (`client/src/App.tsx`):

- Listens for service worker updates

- Shows update notification to users
- Auto-reloads when update is ready
- Disabled in development mode

Offline Functionality

Offline Detection:

- Uses `useOnlineStatus()` hook to detect connection status
- Shows snackbar message when offline: "You are connected with limited functionality. Please check your internet connection."
- Blocks certain actions when offline (e.g., creating reviews, adding queries)

Offline Behavior:

- Cached assets available offline
- Firebase Storage files cached for offline access
- PDFs require online connection (NetworkOnly strategy)
- Review viewing limited when offline
- Error messages shown for offline-restricted actions

Code Location:

- Service worker: `client/src/service-worker.ts`
- Registration: `client/src/serviceWorkerRegistration.ts`
- Update listener: `client/src/App.tsx` (lines 87-126)
- Offline detection: `client/src/useCheckConnection.tsx`

Error Handling and Logging

Logging System

Firestore Logging (`useLogger` hook):

- Logs user actions to Firestore collection: `myworkreview/logs/list`
- Log entries include:
  - `review` : Review ID or reference
  - `message` : Action description
  - `email` : User email
  - `log_time` : Timestamp (YYYY-MM-DD HH:mm:ss format)
- Used throughout the application for audit trails

Activity Logging Types:

- **Review Actions**: Review creation, updates, status changes
- **User Actions**: User login, logout, profile updates
- **Collaboration Actions**: Collaborator additions/removals
- **System Actions**: Automated processes, scheduled tasks
- **Error Logging**: Errors captured and logged for debugging

Log Collection Structure:

```
myworkreview/logs/list/
├── {logId}/
│   ├── review: string (review ID or reference)
│   ├── message: string (action description)
│   ├── email: string (user email)
│   └── log_time: string (YYYY-MM-DD HH:mm:ss)
```

Bug Reporting:

- Endpoint: `/api/send-bug-report`
- Sends bug reports via email to `imimoosa@gmail.com`
- Includes highlight data and error details
- HTML formatted email with JSON preview

Error Handling:

- Try-catch blocks throughout components
- Error messages displayed via snackbar notifications
- Console logging for debugging
- Error boundaries for React components (if implemented)

Code Location:

- Logger hook: `client/src/useLogger.tsx`
- Bug report endpoint: `functions/index.js` (lines 918-950)

Security

API Key Validation

Server-to-Server Requests:

- Middleware: `validateApiKey()` function

- Header: `x-server-server` indicates server-to-server request
- Authorization: `Authorization: Bearer {MWR_SERVER_KEY}`
- Validation: Checks token against `process.env.MWR_SERVER_KEY`
- Response: 403 Forbidden if invalid

Organization API Keys:

- Stored in Firestore: `myworkreview/apis/list`
- Used for Friday integration (`/api/push-mwr-checklist`)
- Validated against Firestore before processing

CORS Configuration

Allowed Origins (`functions/index.js`):

```
const allowedOrigins = [
  "http://localhost:3000",
  "https://localhost:3000",
  "localhost:3000",
  "https://www.myworkreview.netlify.app",
  "https://fridayinc.netlify.app",
  "https://myworkreview.netlify.app",
  "https://myworkreview.netlify.app/",
  "https://us-central1-friday-1363b.cloudfunctions.net",
  "https://us-central1-audit-7ec47.cloudfunctions.net/",
  "https://us-central1-friday-a372b.cloudfunctions.net",
];
```

CORS Logic:

- Server-to-server requests: CORS disabled (`origin: false`)
- Client requests: Origin checked against allowed origins list
- Allowed origins: CORS enabled (`origin: true`)
- Blocked origins: CORS disabled (`origin: false`)

Code Location: `functions/index.js` (lines 75-132)

Authentication Security

Firebase Authentication:

- Google OAuth only

- Token validation on backend
- Custom token generation for Friday integration
- User status check (only active users can authenticate)

Private Review Access:

- Authorization checks in `FileReview.tsx`
- Different logic for Partners vs Managers/Clerks
- Collaborator-based access control
- Team-based access for Partners

Data Security

Firestore Security Rules:

- **Note**: The application does not use Firestore security rules files
- Security is enforced at the **application level** through:
    - Authentication checks in components
    - Authorization logic in `FileReview.tsx` (`checkUserAuthorization()`)
    - Permission checks using `useGetUserType()` hook
    - Inline role checks throughout components
- All Firestore operations require authenticated users
- Access control is implemented in React components and Cloud Functions

Storage Security Rules:

- **Note**: The application does not use Storage security rules files
- Security is enforced at the **application level** through:
    - Authentication required for all uploads
    - Authorization checks before file operations
    - Service account authentication for Google Drive operations
- Firebase Storage files use direct download URLs (authenticated via Firebase Auth)
- Google Drive files require backend API calls (authenticated via service account)

Service Account Keys:

- **MWR Firebase**: `functions/serviceAccountKey.json`
    - Used for Firestore operations
    - Used for Firebase Authentication
    - Used for Firebase Storage operations
- **Friday Firebase**: `functions/serviceAccountKeyFriday.json`
    - Used for Friday Firestore access
    - Used for Friday token generation

- **Google Cloud**: `functions/serviceAccountCloud.json`
    - Used for Google Drive API operations
    - Used for Google Cloud Storage operations
    - Used for Firestore backups
- **Security**: Service account keys are stored in `functions/` directory and should never be committed to version control
- **Permissions**: Service accounts have minimal required permissions (principle of least privilege)

Code Location:

- Authentication checks: Throughout components using `AuthContext`
- Authorization logic: `client/src/FileReview.tsx` (lines 1231-1267)
- Service account usage: `functions/index.js` (lines 1-56)

Performance Optimizations

Caching Strategies

Google Drive PDF Caching:

- Cache location: Google Cloud Storage bucket `cached_reviews`
- Cache path: `pdfs/{driveFileId}.pdf`
- Metadata cache: `metadata/{driveFileId}.json`
- Cache validation: Checks file modification time
- Cache TTL: 24 hours (checks `modifiedTime`)
- Strategy: Serve from cache if up-to-date, otherwise fetch and update

Firebase Storage Caching:

- Service worker caching: 30-day cache, max 50 entries
- Strategy: `StaleWhileRevalidate`
- Cache name: `firebase-storage-cache`

Frontend Caching:

- Workbox precaching for static assets
- Image caching: Max 50 entries
- App shell caching for navigation

React Optimizations

Code Splitting:

- Lazy loading: Components loaded on demand using `React.lazy()`
- Suspense boundaries: Loading states for lazy-loaded components
- Examples: `LazyFileReviewFlagDialog`, `LazyFileReviewCreateDialog` in `FlexUpView.tsx`

Memoization:

- `useMemo`: Memoizes computed values (e.g., `columnsReviews`, `findReviewById`)
- `useCallback`: Memoizes callback functions (e.g., `handleCellEditCommit`, `getDetailPanelContent`)
- Prevents unnecessary re-renders and recalculations

Performance Optimizations:

- Component memoization to prevent unnecessary re-renders
- Callback memoization to prevent function recreation
- Computed value memoization for expensive calculations
- Lazy loading for heavy components

Code Location:

- Lazy loading: `client/src/components/FlexUpView.tsx` (lines 145-147)
- Memoization: Throughout components using `useMemo` and `useCallback`

Batch Operations

Firestore Batch Writes:

- Daily user sync uses batch writes
- Team cleanup uses batch operations
- Efficient for multiple document updates

Concurrency Limits:

- PDF generation: Concurrency limit of 3 (configurable)
- Prevents resource exhaustion
- Configurable in `pubSub_generateChecklistPDFEmails`

Progressive Loading

PDF Loading:

- Progressive loading with progress updates
- Streaming PDF data
- Progress percentage displayed to user
- Chunked reading for large files

Code Location:

- PDF caching: `functions/index.js` (lines 814-870)
- Progressive loading: `client/src/FileReview.tsx` (lines 1657-1782)

Backup and Recovery

Daily Backups

Automated Backups (`pubsub_dailyBackup`):

- Trigger: Pub/Sub topic `daily-backup` (scheduled daily)
- Destination: Google Cloud Storage bucket `gs://mwrdailybackups`
- Scope: All Firestore collections
- Format: Firestore export format
- Retention: Managed by GCS bucket lifecycle policies

Backup Process:

1. Cloud Scheduler triggers Pub/Sub topic
2. Cloud Function exports all Firestore collections
3. Backup stored in GCS with timestamp
4. Operation name logged for tracking

Manual Backup (`createBackup()` function):

- Can be triggered manually via Cloud Function
- Same process as automated backup
- Returns operation name for tracking

Code Location: `functions/index.js` (lines 134-149, 1824-1840)

Recovery Procedures

Restoring from Backup:

1. **Locate Backup**: Find backup in GCS bucket `gs://mwrdailybackups`

2. **Identify Backup**: Backups are timestamped and stored in Firestore export format
3. **Restore Process**:
    - Use Firebase Console or `gcloud` CLI to import backup
    - Command: `gcloud firestore import gs://mwrdailybackups/{backup-path}`
    - Verify import completion
4. **Post-Restore**:
    - Verify data integrity
    - Check user authentication
    - Test critical functions
    - Monitor for errors

Partial Recovery:

- Can restore specific collections
- Use Firestore import with `collectionIds` parameter
- Restore only affected collections

Data Recovery:

- Firestore: Restore from GCS backups
- Google Drive: Files remain in Drive (no backup needed)
- Firebase Storage: Files remain in Storage (no backup needed)
- User data: Restored from Firestore backup

Disaster Recovery Plan

Recovery Objectives:

- **RTO (Recovery Time Objective)**: 4-8 hours
- **RPO (Recovery Point Objective)**: 24 hours (daily backups)

Recovery Steps:

1. **Assessment**: Identify scope of data loss
2. **Backup Selection**: Choose most recent backup before incident
3. **Restore Process**: Follow recovery procedures above
4. **Verification**: Test all critical functions
5. **Communication**: Notify users of recovery status

Prevention Measures:

- Daily automated backups
- Service account key backups (secure storage)

- Environment variable documentation
- Code version control (Git)
- Configuration documentation

Emergency Contacts:

- Firebase Support: https://firebase.google.com/support
- Google Cloud Support: https://cloud.google.com/support
- Service Account Key Recovery: Contact Firebase project administrator

---

Email Notification System

Email Service

Provider: Gmail (via Nodemailer)

- **From Address**: `audit@1-inc.co.za`
- **Authentication**: App password (`AUDIT_EMAIL_PASS`)
- **Transport**: Nodemailer with Gmail service

Email Types

1. **Review Creation Notifications**:
    - Trigger: Firestore trigger `newReviewerTriggerNotification`
    - Recipients: Team partners (active users only)
    - Content: Review name, template, publisher, status, link
    - HTML formatted with styling
2. **Review Update Notifications**:
    - Trigger: Firestore trigger `updatedReviewTriggerNotification`
    - Recipients: Team partners and collaborators
    - Content: Review changes, status updates, link
3. **Collaborator Change Notifications**:
    - Trigger: Firestore trigger on collaborator changes
    - Recipients: Affected users
    - Content: Access granted/removed, expiration dates
4. **Tender Deadline Notifications**:

    - Trigger: Pub/Sub `checkTenderReviewNotificationsDue`
    - Recipients: Team partners
    - Timing: 7 days before deadline, on deadline day
    - Content: Deadline date, review details, link

5. **Custom Notifications**:

    - Endpoint: `/api/send-notification`
    - Manual notifications from review interface
    - Recipients: Selected users
    - Content: Custom message, review link

6. **Weekly Checklist PDF Emails**:

    - Trigger: Pub/Sub `pubSub_generateChecklistPDFEmails`
    - Recipients: Users assigned to checklist sections
    - Content: PDF report generated via Puppeteer
    - Template: HTML template from `checklist_template.html`

7. **Bug Reports**:

    - Endpoint: `/api/send-bug-report`
    - Recipients: `imimoosa@gmail.com`
    - Content: Highlight data, error details

Email Templates

HTML Templates:

- Styled with inline CSS
- Black header with review name
- Verdana font family
- Responsive design
- Links to review pages

Code Location:

- Email functions: `functions/index.js` (lines 918-1117, 1595-1823)
- Checklist emails: `functions/weeklyChecklistEmails.js`
- Collaborator notifications: `functions/collaboratorReviewChanges.js`

Code Obfuscation

Production Build Obfuscation

Process:

- Script: `client/obfuscate.js`
- Runs after production build
- Obfuscates all JavaScript files in `build/static/js/`

Obfuscation Options:

- `compact: true` : Minifies code
- `controlFlowFlattening: true` : Flattens control flow (75% threshold)
- `deadCodeInjection: true` : Injects dead code (40% threshold)
- `disableConsoleOutput: true` : Removes console statements
- `mangle: true` : Mangles variable names
- `renameGlobals: true` : Renames global variables
- `selfDefending: true` : Protects against deobfuscation
- `stringArray: true` : Encodes strings (RC4 encoding, 75% threshold)

Build Command:

```
npm run build:prod
```

Code Location: `client/obfuscate.js`

Troubleshooting

Authentication Issues

User Cannot Login:

- **Symptom**: Login fails or redirects to login page
- **Causes**:
  - User not in Firestore `users/list` collection
  - User status is "inactive"
  - Google OAuth not configured correctly
  - Firebase Auth domain not whitelisted
- **Solutions**:
  1. Check if user exists in `myworkreview/users/list`

2. Verify user `status` is "active"
3. Check Firebase Auth configuration
4. Verify Google OAuth consent screen settings
5. Check browser console for errors

Permission Denied Errors:

- **Symptom**: "Permission denied" or "Forbidden" errors
- **Causes**:
    - User role not set correctly
    - Permission table missing permissions
    - Inline permission check blocking access
- **Solutions**:
    1. Verify user `role` field references valid user group
    2. Check user group `roles` object in `userGroups/list`
    3. Review inline permission checks in component code
    4. Check `currentUser.permissions` object

Token Expiration:

- **Symptom**: User logged out unexpectedly
- **Causes**:
    - Firebase Auth token expired
    - Token refresh failed
- **Solutions**:
    1. Check Firebase Auth token refresh logic
    2. Verify `onAuthStateChanged` listener is active
    3. Check network connectivity

Data Access Issues

Cannot View Review:

- **Symptom**: Review not loading or "Access Denied" message
- **Causes**:
    - Private review access restrictions
    - User not in team or collaborator list
    - Clerk role restrictions
- **Solutions**:
    1. Check `checkUserAuthorization()` logic in `FileReview.tsx`
    2. Verify user is collaborator or team member
    3. Check review `private` field and `privateBy` field

4. Verify user role (Clerks have restricted access)

Firestore Permission Errors:

- **Symptom**: "Missing or insufficient permissions" error
- **Causes**:
  - User not authenticated
  - Application-level security checks failing
- **Solutions**:
  1. Verify user is authenticated (`currentUser` exists)
  2. Check authorization logic in component
  3. Review inline permission checks
  4. Check Firestore query filters

Storage Access Errors:

- **Symptom**: Cannot upload or download files
- **Causes**:
  - Firebase Storage permissions
  - Google Drive API permissions
  - Service account key issues
- **Solutions**:
  1. Verify Firebase Storage bucket exists
  2. Check Google Drive API credentials
  3. Verify service account has required permissions
  4. Check file size limits

Backend Function Issues

Cloud Function Timeout:

- **Symptom**: Function execution timeout errors
- **Causes**:
  - Function timeout too short
  - Long-running operations
  - Resource-intensive tasks
- **Solutions**:
  1. Increase function timeout in `functions.runWith()`
  2. Use high memory configuration for heavy operations
  3. Break operations into smaller chunks
  4. Use Pub/Sub for long-running tasks

Cloud Function Errors:

- **Symptom**: Function returns error or fails
- **Causes**:
  - Missing environment variables
  - Service account key issues
  - API rate limits
  - Invalid request data
- **Solutions**:
  1. Check Cloud Function logs in Firebase Console
  2. Verify all environment variables are set
  3. Check service account key files exist
  4. Review request payload format
  5. Check API quotas and limits

Pub/Sub Trigger Not Firing:

- **Symptom**: Scheduled tasks not running
- **Causes**:
  - Cloud Scheduler not configured
  - Pub/Sub topic not created
  - Function not deployed
- **Solutions**:
  1. Verify Cloud Scheduler job exists
  2. Check Pub/Sub topic exists
  3. Verify function is deployed
  4. Check function logs for errors

Performance Issues

Slow PDF Loading:

- **Symptom**: PDFs take long to load
- **Causes**:
  - Large file sizes
  - Cache not working
  - Network issues
  - Google Drive API rate limits
- **Solutions**:
  1. Check cache bucket `cached_reviews` exists
  2. Verify cache metadata is being updated
  3. Check network connectivity
  4. Monitor Google Drive API usage
  5. Consider file size optimization

Slow Page Loads:

- **Symptom**: Application loads slowly
- **Causes**:
  - Large bundle size
  - Too many Firestore queries
  - Network latency
  - Service worker issues
- **Solutions**:
  1. Check bundle size in build output
  2. Review Firestore query patterns
  3. Enable code splitting (lazy loading)
  4. Check service worker cache
  5. Use React DevTools Profiler

High Memory Usage:

- **Symptom**: Browser crashes or slow performance
- **Causes**:
  - Memory leaks
  - Large data sets in memory
  - Too many listeners
- **Solutions**:
  1. Check for memory leaks in React DevTools
  2. Review Firestore listener cleanup
  3. Implement pagination for large lists
  4. Use `useMemo` and `useCallback` appropriately

Integration Issues

Friday Integration Not Working:

- **Symptom**: Cannot access Friday data or token exchange fails
- **Causes**:
  - Friday Firebase app not initialized
  - Token exchange endpoint failing
  - API key validation failing
- **Solutions**:
  1. Verify Friday Firebase config in `firebase.tsx`
  2. Check token exchange endpoint logs
  3. Verify `serviceAccountKeyFriday.json` exists
  4. Check Friday API key in Firestore

Google Drive Integration Issues:

- **Symptom**: Cannot upload or fetch files from Drive
- **Causes**:
  - Google Drive API not enabled
  - Service account key missing
  - Drive API permissions
- **Solutions**:
  1. Verify Google Drive API is enabled in GCP
  2. Check `serviceAccountCloud.json` exists
  3. Verify service account has Drive API access
  4. Check Drive API quotas

Google Apps Script Sync Failing:

- **Symptom**: Daily user sync not working
- **Causes**:
  - `USER_EMAIL_SYNC_KEY` incorrect
  - Apps Script not accessible
  - Pub/Sub trigger not firing
- **Solutions**:
  1. Verify `USER_EMAIL_SYNC_KEY` environment variable
  2. Test Apps Script URL manually
  3. Check Pub/Sub trigger logs
  4. Verify Apps Script permissions

Common Error Messages

"Permission denied":

- User lacks required permissions
- Check user role and permissions table
- Review inline permission checks

"Missing or insufficient permissions":

- Firestore security (application-level)
- Check user authentication
- Verify authorization logic

"Invalid API Key":

- Server-to-server API key incorrect
- Check `MWR_SERVER_KEY` environment variable

- Verify `x-server-server` header

"CORS policy":

- Origin not in allowed origins list
- Check `allowedOrigins` array in `functions/index.js`
- Verify request origin

"Service worker registration failed":

- Service worker file not found
- Check build output
- Verify service worker path

"Network request failed":

- Offline or network issues
- Check internet connection
- Verify API endpoints are accessible

Getting Help

Documentation:

- This README file
- Firebase Documentation: https://firebase.google.com/docs
- React Documentation: https://react.dev
- Material-UI Documentation: https://mui.com

Logs and Debugging:

- **Frontend**: Browser DevTools console
- **Backend**: Firebase Cloud Functions logs
- **Firestore**: Firebase Console > Firestore > Usage tab
- **Storage**: Firebase Console > Storage > Files tab

Support Channels:

- Firebase Support: https://firebase.google.com/support
- Google Cloud Support: https://cloud.google.com/support
- GitHub Issues: (if repository is public)

Code Locations for Common Issues:

- Authentication: `client/src/firebase.tsx`, `client/src/App.tsx`
- Authorization: `client/src/FileReview.tsx` (lines 1231-1267)

- API Endpoints: `functions/index.js`
- Service Worker: `client/src/service-worker.ts`
- Error Handling: Throughout components using try-catch

Debugging

Frontend:

- Browser DevTools console
- React DevTools
- Service Worker DevTools
- Network tab for API calls
- Application tab for service worker status

Backend:

- Firebase Cloud Functions logs
- Cloud Scheduler logs
- Pub/Sub logs
- Firestore logs
- Google Cloud Logging

Code Location:

- Logging throughout codebase via `console.log` and `console.error`

Backend Tests:

- Manual testing via Firebase Functions logs
- Use Firebase emulators for local testing

Important Notes

Security Considerations

1. **Service Account Keys**: Never commit service account JSON files to version control
2. **API Keys**: Store API keys in environment variables
3. **CORS**: Configured for specific origins only
4. **Authentication**: All routes require authentication
5. **Permissions**: Role-based access control enforced

Performance Considerations

1. **Firestore Caching**: Unlimited cache size configured for offline support
2. **PDF Caching**: Google Cloud Storage caching for PDF files
3. **Lazy Loading**: React components lazy-loaded for code splitting
4. **Service Workers**: PWA caching for offline access

Known Limitations

1. **File Size**: Large PDF files may take time to load
2. **Concurrent Users**: Firestore read/write limits apply
3. **Email Rate Limits**: Gmail has daily sending limits
4. **Function Timeout**: Maximum 9 minutes for Cloud Functions

Maintenance Tasks

1. **Daily Backups**: Automatic Firestore backups to GCS
2. **User Sync**: Daily user synchronization from external source
3. **Temporary Access Cleanup**: Automatic removal of expired collaborators
4. **Cache Management**: PDF cache TTL is 24 hours

Integration Points

1. **Friday Application**:
    - Token exchange for cross-app authentication
    - Checklist import/export
    - Review data retrieval
    - Firestore data access
2. **Google Apps Script**:
    - User email list synchronization
    - URL: `https://script.google.com/macros/s/AKfycbyqQp6ZLZn_vSNi4dxZxWHa1yEoFEcKMoWunP3Mytx2sXL9N0g/exec`
    - Returns user list with name, email, and photo
3. **Google Drive API**:
    - File storage and management
    - Folder structure creation
    - File upload/download
    - Parent folder: `1LZwInex0XVr6UKqyY_w6G2JeUyeUiiPG`

4. **Gmail (Nodemailer)**:
    - Email notifications
    - From: `audit@l-inc.co.za`
    - HTML email templates
5. **Material-UI Premium**:
    - Requires license key (`REACT_APP_MUI`)
    - Used for Data Grid Premium component

Troubleshooting

Common Issues:

1. **Authentication Failures**:
    - Check user status in Firestore (must be "active")
    - Verify Firebase Auth configuration
    - Check CORS settings in Cloud Functions
    - Verify user exists in `myworkreview/users/list` collection
2. **File Upload Issues**:
    - Verify Google Drive API credentials (`serviceAccountCloud.json`)
    - Check service account permissions in Google Cloud Console
    - Verify parent folder ID: `1LZwInex0XVr6UKqyY_w6G2JeUyeUiiPG`
    - Check Drive API quotas and limits
3. **Email Not Sending**:
    - Check Gmail app password (`AUDIT_EMAIL_PASS` environment variable)
    - Verify email service configuration
    - Check user status (inactive users don't receive emails)
    - Verify Gmail sending limits (500 emails/day for free accounts)
4. **Friday Integration Issues**:
    - Verify Friday service account key (`serviceAccountKeyFriday.json`)
    - Check API key in `apis` collection
    - Verify CORS configuration
    - Check Friday Firebase project access
5. **Build/Obfuscation Issues**:

    - Ensure `REACT_APP_MUI` environment variable is set
    - Check obfuscation script permissions
    - Verify all dependencies are installed

6. **User Sync Issues**:

    - Verify `USER_EMAIL_SYNC_KEY` environment variable
    - Check Google Apps Script permissions
    - Verify script URL is accessible
    - Check Firestore write permissions

7. **PWA/Service Worker Issues**:

    - Clear browser cache and service worker
    - Check `manifest.json` configuration
    - Verify service worker registration
    - Check HTTPS requirement for PWA features

Support and Contact

For issues or questions:

- Check Firebase Console logs
- Review Cloud Functions logs
- Check Firestore data structure
- Verify environment variables

Additional Resources

Firebase Documentation

- Firebase Authentication
- Cloud Firestore
- Cloud Functions
- Firebase Storage

Google APIs

- Google Drive API
- Google Cloud Storage

React/TypeScript

- React Documentation
- TypeScript Documentation
- Material-UI Documentation

Quick Start Guide for New Developers

First Steps

1. **Get Access**:
   - Firebase project access (audit-7ec47)
   - Google Cloud Console access
   - Netlify account access
   - Material-UI Premium license key
2. **Obtain Required Credentials**:
   - Service account JSON files from Firebase Console
   - Environment variables from previous developer
   - Google Apps Script URL and authorization key
   - Gmail app password for notifications
3. **Set Up Local Environment**:
   - Clone repository
   - Install dependencies (frontend and backend)
   - Configure service account files
   - Set up environment variables
   - Test local development server
4. **Understand Key Files**:
   - `client/src/firebase.tsx` : Firebase configuration
   - `client/src/App.tsx` : Main app logic and routing
   - `functions/index.js` : All backend endpoints
   - `client/src/FileReview.tsx` : Core PDF review functionality
5. **Test Critical Paths**:
   - User authentication

    - PDF upload and review
    - Checklist creation
    - Email notifications
    - Friday integration

Important Files to Review First

1. **Authentication Flow**: `client/src/firebase.tsx` and `client/src/App.tsx`
2. **Backend API**: `functions/index.js` (all endpoints)
3. **PDF Review**: `client/src/FileReview.tsx` (main review interface)
4. **Database Schema**: Review Firestore collections in Firebase Console
5. **Integration Points**: Friday token exchange and Google Apps Script

Common Tasks

- **Adding a New Feature**: Start in `client/src/` for frontend, `functions/index.js` for backend
- **Debugging**: Check Firebase Console logs, browser console, and Cloud Functions logs
- **Deploying**: Frontend via Netlify, Backend via `firebase deploy --only functions`
- **User Management**: Users synced daily from Google Apps Script, managed in Firestore

---

Last Updated: 7 November 2025
Maintained By: Prolific Solutions