
# Project Concept: Westcon Marketing Platform – MDF Activities Dashboard

## Project overview

**Purpose and Background:** The Westcon Marketing Platform – **MDF Activities Dashboard** is designed as a central application to efficiently manage **Marketing Development Funds (MDF)** activities. The solution will replace the current Smartsheet Setup. The goal of the project is to provide an **"Airtable-like" dashboard** in which all MDF activities can be recorded, tracked and edited collaboratively – with strict **role and rights management**, complete **audit trail** history and seamless integration into the existing Microsoft environment.

**Project objectives:**

- **Centralized data collection and maintenance:** All relevant information about MDF activities (e.g., campaigns, budgets, approval status, receipts) is kept in a central **Azure SQL** database and made accessible via a web-based dashboard.
- **Improved collaboration:** Users can collaborate on activities, add comments and attachments (e.g., invoices, receipts) to improve alignment between marketing, operations, finance, and more.
- **Visual Preparation & Productivity:** The dashboard offers **tabular views** and a **Kanban view** to visualize activities according to status or unit. Filters, sorting, and a global search make it possible to quickly find the right records. Users can save personalized **views** to recall needed filter settings with one click.
- **Automated notifications:** Important events (e.g., new comments, status changes, assignments) trigger **email notifications**. In later expansion stages, **Microsoft Teams** notifications via Adaptive Cards will also be possible, including direct **release/approval** actions in Teams.
- **Compliance & transparency:** As a subsidiary of a US listed company (SOXCompliance relevant), the platform meets high requirements for **data integrity and traceability**. Every change to financially-related data should be logged to ensure a complete **audit trail** [1]. In addition, access is strictly controlled on a role-based basis and sensitive data (e.g. budget fields) is only displayed to authorized people.

## Technology stack and system architecture

The project relies on a **modern Microsoft/Azure technology stack** that works seamlessly together (see Fig. 1).

*Fig. 1: Azure Static Web Apps architecture – globally distributed hosting for the React frontend, integrated with serverless Azure Functions as the API backend* [2] [3].

The core components are:

- **Frontend:** The user interface is developed using **React** and **TypeScript**. For a consistent, modern UI design, **Tailwind CSS** is used, supplemented by **ShadCN UI**Component Library (a collection of pre-built, accessible React components based on Radix UI and Tailwind) and **Framer Motion** for smooth animations and interactive effects. The front-end code is implemented as a **single-page application**, which can later also be integrated as a **Teams app** (personal app or tab in Teams) to optionally enable single sign-on within Teams.

**MY-MIND S.R.L.**
*"Better Data, better AI, better Business!"*

Strada Paris 9, Level 2 App. 3,
011813, Bucharest, Romania

**Consultant:** **Armand Sullot**
**Contact:** armand@my-mind.io

- **Backend / API:** The business logic and data access layer are implemented in **Azure Functions** (Node.js with TypeScript). Azure Functions serve as a **serverless API** that is consumed by the frontend via HTTPs calls. Azure Static Web Apps provides an *integrated connection* between the ReactFrontend and the Functions – calls to `/api/*` URLs are automatically routed to the provided Functions without the need for complex CORS configuration [3]. The API layer encapsulates all operations such as reading/writing data in the database, authentication checks, and integrations (e.g., emailing).

- **Database:** Central persistence is an **Azure SQL Database** (relational database). It stores all **MDF activities**, user assignments (units/roles), comments and attachment metadata. The relational structure ensures consistent financial data and enables complex queries (e.g. reporting, total budget per quarter, etc.). For data access in the backend, a suitable **ORM/query builder** (e.g. Prisma or Sequelize) is used to achieve type safety and easy maintainability. Sensitive data in the DB is automatically encrypted by Azure SQL **Transparent Data Encryption (TDE** ).

- **Authentication & Authorization:** The app uses **Azure Active Directory (Entra ID)** for login. The **Microsoft Login** 4 is **integrated via** MSAL.js (Microsoft Authentication Library for JS) in the frontend , so that users log in with their corporate Azure AD accounts (Office 365 accounts). For authorization, Azure AD **app roles** or **Groups: Users are assigned to the** Marketing**,** Marketing Execution**,** Marketing Operations**,** Finance**, and** Administration **application roles based on their AAD roles**. These roles can be mapped in AAD as an App Role or Security Group and come into the application with the JWT token. An Azure Static Web App (SWA) in the **standard tier** allows the use of user-defined roles: A serverless function can change the AD
  Check group memberships via Microsoft Graph and transfer corresponding roles to SWA [5] [6]. This ensures that **role-based access control** extends to the frontend and the API. Unauthorized access is intercepted on the server side; at the UI level, functions or data fields that a user is not allowed to use due to a lack of rights are not even displayed (e.g. financial fields are only visible to Finance).

- **File storage for attachments:** Users can upload **screenshots/attachments** per activity (e.g. invoices as PNG/JPG or PDF). These are stored **in** Azure Blob Storage in an **audit-proof** manner. By enabling container immutability (e.g. time-based retention policy of e.g. 7 years), a *WORM* (Write-Once-Read-Many) status can be achieved [7] [8]. This means that attachments cannot be changed or deleted afterwards, which is crucial for audit purposes. Each attachment is also given a digital fingerprint (hash) in the DB to detect manipulation.

- **Email & Notifications: The** Microsoft Graph API (Office 365) **is used for notifications**. For example, the system can send an email on behalf of a service account or the logged-in user by calling the Graph API `sendMail` method [9] [10]. Typical triggers: a comment has been added, an activity has been forwarded for approval, etc. In a later phase, it is planned to integrate **Microsoft Teams** more closely: On the one hand, the entire application could be made available as **a Teams tab** (with SingleSign-On via Teams client), and on the other hand, important events can be posted in Teams chats or channels via Adaptive Cards. For example, a Finance release could be sent directly to the responsible Finance employee as an Adaptive Card, who can then *click Approve/Reject*. Microsoft now offers the **Approvals App API** for this purpose , which can be used to initiate and track approval workflows in Teams from third-party applications [11]. This deep Teams integration is optional and can be added in later expansion stages to meet users where they communicate.

- **Hosting & DevOps:** The solution is hosted as **Azure Static Web App** (SWA). This service hosts the React frontend as a *static site* distributed worldwide (for fast loading times via CDN) and at the same time manages the Azure Functions as a connected API [2]. SWA offers integrated CI/CD: every push to the main branch of the codebase is automatically built and released [12]. Modern tools are used for development: **Visual Studio Code** with Azure Extensions, SWA CLI (for local testing of static web apps incl. functions) and GitHub (or Azure DevOps) for source code control and automatic build/release.

In summary, the result is a modern web application with cloud-native architecture: **single-page frontend** for optimal user experience, **serverless backend** for scalability, and **Azure services** for security, compliance, and seamless integration into the Microsoft ecosystem.

## Detailed phase plan

For step-by-step implementation, the project will be divided into phases. Each phase provides a functional partial result (*snapshot*), which serves as a milestone and can already be presented to the customer team if necessary. In this way, functionality is built up incrementally and risks are detected at an early stage. Below is an overview of the recommended phases:

### PHASE 1: PROJECT SETUP & FRONTEND BASIS

- **Goal:** Building the basic framework of the application and first functional prototype in the frontend. The focus is on the React app with design system and authentication.
- **Description:** Initialization of the monorepo (or two coupled repos) with **frontend** and **backend** directory structure. Setting up the React app (e.g. via Vite or Create React App) in TypeScript. Integration of Tailwind CSS and addition of the first **ShadCN/UI** components (e.g. uniform buttons, dialogs) according to the desired design. Implementation of a **login page** or component and integration of **MSAL.js** for Microsoft login. At this time,
  If necessary, a mock login **can stil** be used if the Azure AD app registration is not yet available – but the structure for Auth (Context, Routing Guard for protected pages) is already being prepared. After logging in, the user will see a blank dashboard page with navigation frames (e.g., sidebar for later filters/navigation, header with user information, and logout).
- **Folder structure:** *repo-root/*
- *frontend/* – React project (e.g. src/components, src/pages for dashboard, etc.)
- *api/* – (still empty or minimal placeholder for Azure Functions)
- *README.md* – Project Documentation
- **Recommended tools:** Vite/CRA for fast React startup, Tailwind configuration (with PostCSS), shadcn/ ui setup according to doc (CLI for adding components), MSAL React or MSAL Browser for Auth, Git (initialize repository).
- **Snapshot point:** At the end of phase 1, there is a login flow: the user can log in with
  Azure AD (in the POC possibly against a demo tenant) and gets to the main dashboard UI (still without real data). The application runs on-premises and ideally already as an Azure Static Web App deployment (for the POC on Azure, with dummy API).

### PHASE 2: BACKEND FOUNDATION (AZURE FUNCTIONS API) & FRONTEND CONNECTION

- **Goal:** Setup of the serverless backend API and connection of the frontend to the first API end points (with dummy data).
- **Description:** In this phase, the *api/* directory is populated with Azure Functions. First, one or two exemplary **HTTP trigger functions** are created, e.g. `GetActivities` (GET /api/ activities) and `AddActivity` (POST /api/activities). In phase 2, they still work with static **demo data** (e.g. an in-memory array within the function or a JSON file in the project) to test the end-to-end chain. An important focus is authentication at the **API endpoint**:
  The Azure Function should only be callable if a valid Azure AD JWT token is given. Azure Static Web Apps makes this easier with *Easy Auth* – but in on-premises or outside SWA, the token verification must be configured (e.g. using **@azure/msal-node** or **passport-azure-ad**, or simply using the SWA mechanism in standard hosting). In the frontend, an initial data display is implemented: After logging in, the React dashboard calls the
  `GetActivities` API and displays the returned (demo) activities in a table, e.g. a simple table or a list view, styled with Tailwind/ ShadCN.
- **Folder:**
- *frontend/src/api/* -- Optional: Definitions for API calls (e.g. a apiClient.js/ts with Axios or Fetch wrapper)
- *api/GetActivities/index.ts* – Azure Function Code for GET (returns list of activities)
- *api/AddActivity/index.ts* - Azure Function Code for POST (takes new activity, stores it in memory for now)
- *api/shared/* – if necessary, common code (data model interfaces for activities etc., can be `shared between frontend/backend` via npm workspaces)
- **Recommended tools:** Azure Functions Core Tools (for on-premises execution), SWA CLI (for on-premises execution)
  Frontend+Functions to test together locally), Axios or Fetch API for Data Calls, Jest

(preparation for later testing), ESLint/Prettier (code quality from the beginning). For Auth in Azure Function, we can first use **SWA Auth Emulator** or implement a simple JWT check.

- **Snapshot point:** End of Phase 2, a simple end-to-end app is available: Login with Microsoft Account, then a hardcoded activity list is fetched from the API and stored in the frontend displayed. This proves the interaction between the frontend, auth and backend. All data is still static, but the basic framework is in place.

## PHASE 3: AZURE SQL INTEGRATION (PERSISTENCE LAYER)

- **Goal:** Connection of a real **Azure SQL database** and persistence of the MDF data. CRUDOperations (create, read, update activities) now work permanently.
- **Description:** In this phase, the previously static API is connected to the database. To do this, data modeling is required first . The most important table is **Activities**: Fields could be *ActivityID (PK), Title*, *Description*, *BudgetAllocated*, *BudgetSpent*, *Status*, *Owner*, *Unit*, *CreatedDate*, *ModifiedDate*, etc. In addition, tables for **Comments** (with foreign key on ActivityID, Text, Author, Timestamp) and **attachments** (foreign key on ActivityID or CommentID, path to blob, uploader, timestamp). If necessary, also a **Users/Teams** table, unless everything is dynamically sourced from Azure AD – or at least an assignment of Azure AD user to "unit" (marketing, finance, etc.), unless this can be derived 1:1 from AD groups. The Azure SQL DB is created in Azure and connected via a connection string (or better: via the **Managed Identity** of the function). In the backend code, we replace dummy data with real DB queries. Use of an **ORM** (e.g. Prism): we define the schema accordingly and generate models. The Functions `GetActivities` and `AddActivity` now access the DB – `AddActivity` performs a INSERT, `GetActivities` a SELECT (possibly with JOINs, e.g. if the owner name is pulled from a user table). For local development, an **Azure SQL Edge** or LocalDB with an identical schema can be used. Little changes in the frontend, except that the data is now real. In addition, we implement rudimentary **input validation**: The function checks, for example, that required fields are not empty, that values are within the valid range, etc., to ensure clean data in the DB.
- **Folder:**
- *api/* – additional config for DB access, e.g. *api/db/* (for prism schema `schema.prisma` and generated client classes)
- *.env* (on-premises) / Azure App Settings – contains DB connection string or uses Managed Identity (Azure Functions can access the DB via Azure AD Identity, so there is no password in the code)
- *frontend/src/models/* – optional definition of common interfaces for Activity, Comment, etc., if not already available.
- **Recommended tools:** Prisma ORM (automates a lot of DB access and migrations), Azure Data Studio or SQL Server Management Studio (for creating the schema and testing queries), Faker (for generating seed data for tests), DotEnv (for local configuration).
- **Snapshot Point:** End of Phase 3, the application can **store real data**. A user can create a new activity in the UI (possibly via a simple form) – this process calls the real API (Azure Function) that writes to Azure SQL. `GetActivities` then shows the advanced List including the new activity. All data remains persistent in the Azure SQL DB. This is a good time to demonstrate an initial *end-to-end use case* to the specialist department (e.g. creating an activity).

## PHASE 4: ROLE & RIGHTS MANAGEMENT, DEEPEN AZURE AD INTEGRATION

- **Goal:** Implementation of **fine-grained access rights** in frontend and backend based on the Azure AD roles/groups (**Marketing**, **Execution**, **Operations**, **Finance**, **Admin**). Ensure that **all security requirements** are met (only authorized users see/edit certain content).
- **Description:** In phase 4, we link the authentication information of the logged-in Users with the application. To do this, Azure AD defines App Roles for App Registration (e.g., Finance, Value, finance, MarketingOps, Value, marketingOps, etc.) and assigns appropriate user groups. When logging in, MSAL.js receives an ID token with the **roles** claims. Alternatively, you can also evaluate Azure AD group memberships. In Azure Static Web App (Standard Plan) can be set up via **custom roles** Function, which is assigned to the User is assigned a Static Web App Role [5] [6]. In the backend, we check the permission for each API operation: e.g. `AddActivity` may be done by marketing or admin, but

**Finance** roles may only be allowed to read. For read operations such as `GetActivities`, data filters **could** apply: Marketing only sees its own unit data, Admin sees all, etc. **Columns** can also be restricted: e.g. Finance-specific fields (e.g. *Costs Released* or *Reimbursement Amount*) are only delivered by the API if the calling user has the role Finance or Admin; in the frontend, such fields are given a corresponding visibility condition. Technically, this is achieved by evaluating the JWT token claims in the function, or by using the **/.auth/me** Endpoint of SWA that provides the user information. In addition, the frontend menu is Role adjusted (e.g. admins see an "admin panel" area, normal users don't). In this phase, the **unit** concept should also be anchored: Users belong to a unit (stored either in Azure AD as an attribute or as a separate table). This can be used, for example, to provide a My Unit filter in the activity list, or for workflows (an activity may move from Marketing to Finance for review, etc.).

- **Folder:**
- *api/Auth/* – possibly Function *GetUserRoles* if custom role assignment is used via Function (this would call Graph API, require **User.Read.All** rights and read group memberships to derive roles) [13].
- *frontend/src/context/AuthContext.ts* – Contains the user info including roles after login, makes it available app-wide.
- *frontend/src/components/ProtectedRoute.tsx* – Higher-order component or route wrapper that intercepts unauthorized access (e.g. redirect if role is missing).
- *frontend/src/config/roles.ts* – Defines which rights have which role if necessary (mapping to use in code).
- **Recommended tools:** Microsoft Graph SDK (for Node, e.g. to retrieve group information if needed), JWT library (e.g. **jsonwebtoken** in Node to check token claims on the server side if SWA doesn't take care of everything), Azure AD portal (for App Roles setup), Azure CLI/PowerShell (for automated deployment of app settings including roles).
- **Snapshot point:** At the end of phase 4, the security architecture is implemented. When logging in, the user receives exactly the rights to which he is entitled. If he tries to carry out an unauthorized action (e.g. Finance user creates marketing activity or Marketing user sees Finance field), this is prevented. This milestone is critical for SOX compliance: only authorized persons can view/modify data. This can be demonstrated, for example, by using two
  test users with different roles and compares their views/action options.

**PHASE 5: ADVANCED FEATURES (KANBAN, FILTERS, VIEWS)**

- **Goal:** To increase ease of use and visualization options through additional front-end features: **Kanban board view**, extended **filter and search functions** as well as **saved views**.
- **Description:** A tabular representation of the activities is expected to exist until phase 4. In
  Phase 5 implements an alternate **Kanban view**. Users can switch between table and Kanban view. For example, the Kanban board shows columns by status (scheduled, under review, approved, rejected, completed) and lists the activities as **cards** that can be dragged and dropped between columns. The move would update the status in the background (Azure Function call). Here, **Framer Motion** can be used for smooth map animations during drag and drop, or libraries such as **React DnD Kit** for the logical side of drag and drop. In addition, **filter options** have been expanded: e.g. filters by unit, by status, by time period. A **global search** (input field that searches across all important text fields – title, description, comments) is integrated. To avoid users having to set complex filters every time, there is also the option **of saving user-defined views**: e.g. a user can define a filter "My open activities in Q1" and save it under a name. These saved views are either stored locally in the browser or (better) stored in the backend per user profile, so that they are available on every device. On the UI side, a drop-down or sidebar area could display the saved views. Storage requires a new API endpoint (e.g. `SaveView`), which takes a combination of the filter settings and the name and stores views in a table (UserID, Name, FilterJSON).
  The filter engine in the frontend must be able to set the filters from such a JSON.
  Performance optimization: If the data sets are very numerous, the backend should support pagination and, if necessary, server-side filtering (the filter criteria are given in the API call).
- **Folder:**
- *frontend/src/components/KanbanBoard/* – Components for Kanban (Board, Column, Card)
- *frontend/src/hooks/useDragDrop.ts* – if necessary, hook for drag&drop logic
- *frontend/src/components/FilterBar.tsx* -- UI for filter inputs (dropdowns, text search, datepicker, etc.)
- *api/SaveView/index.ts* -- Function for saving a view (POST; writes to views table)
- *api/ListViews/index.ts* – Function for retrieving saved views per user (GET; usable when loading the page)

- *sql/tables.sql* – Extension of the DB script with table *Views(User, Name, FilterConfig)*
- **Recommended tools:** Drag&Drop library (React Beautiful DnD, DnD Kit or even board component of e.g. Syncfusion as inspiration), date-fns (for date filters), state or Redux (if global state management is needed for filters/view), Lodash (helpful utility functions for deep merge of filter objects etc.), maybe Fuse.js for client-side fuzzy search (or SQL full-text index for server-side search).
- **Snapshot point:** After phase 5, the dashboard offers significantly more **usability**: Users can flexibly display their data. A concrete showcase: By default, Marketing Manager Max sees the Kanban board of all activities in his unit but can switch to the table view with two clicks and enter a search term to filter for "Cisco campaign", for example. It saves this search as a "Cisco Campaigns" view. The financial controller, on the other hand, sees a filtered list of all activities that need to be approved by default. This phase increases the product value enormously and should be closely coordinated with user feedback (UX testing with key users if necessary).

## PHASE 6: COLLABORATION FEATURES (COMMENTS & ATTACHMENTS)

- **Goal:** To improve **collaboration and traceability** through the possibility to have a **discussion (comment thread)** per activity and to add **file attachments** (screenshots, invoices) – all versioned and auditable.
- **Description:** In this phase, a detail view is implemented for each activity and a **comment thread** underneath , similar to a ticketing system. Users can add comments; each comment contains the author, timestamp and text. Comments are **immutable** (editing only shortly after creation, otherwise new follow-up comments instead of edit to maintain audit trail). The Azure SQL table **Comments** was already created in phase 3 – now we implement the `functions AddComment` (POST, requests ActivityID and text) and `ListComments` (GET, returns all comments to ActivityID, paging if necessary). An Azure **Blob Storage** Container is set up for attachments. Each uploaded file (UI: Drag&Drop or File selection at comment) is loaded from the frontend directly into blob storage or via a function (e.g. `UploadAttachment` – Option: Get SAS tokens from Function and then upload them directly from the browser). Important: Saving must be **Audit-proof** be. Therefore, we activate a **Immutability Policy** (e.g. legal hold or time-based retention) [7] [8] so that files cannot be deleted or overwritten until the retention period expires. The metadata of the attachment (blob URL, associated activity/comment, uploader, timestamp, possibly hash) is stored in the **Engagers**table. In the UI, attachments are displayed either as thumbnails (for images) or as file links. When clicked, they can be opened in a viewer or new tab. In addition, in this phase we can **Email Notifications** for comments: e.g. when someone adds a comment to an activity, everyone gets *Observer* or responsible persons. The email is sent via the Graph API `sendMail` (Delegated in the user context or via an app account) [9] [10]. To do this, Azure AD must have permission **Mail.Send** for the app and either the user consented, or an app secret is used. Content of the mail: link to the activity, excerpt of the comment, etc. These features strengthen collaboration and ensure that any changes **Documented and traceable** – a must for internal audits.
- **Folder:**
- *frontend/src/pages/ActivityDetail.tsx* – Detail view of an activity with comment list and file upload component
- *frontend/src/components/CommentList.jsx/CommentItem.jsx* – Display of comments
- *frontend/src/components/FileUploader.jsx* – Component to upload attachments (with progress indicator)
- *api/AddComment/index.ts* – Function for the new comment (writes to comments table, optionally send e-mail to thrust)
- *api/ListComments/index.ts* – Function for loading the comments of an activity
- *api/UploadAttachment/index.ts* - (Optional) Function that accepts uploads or issues SAS tokens
- *api/NotifyNewComment/index.ts* – (Optional) separate process or integrated into AddComment: outsource logic for sending emails
- *azure/blob-storage/* – (documentation/script to set up the blob container with immutability)
- **Recommended tools:** Azure SDK for JavaScript/TS (for blob storage access, e.g. `@azure/storageblob` Package), UUID library (generate file names uniquely), Sharp (if images are to be processed server-side), Outlook mail templates (for formatted notification mails possibly as Adaptive Card via email).
- **Snapshot point:** After phase 6, the system is **audit-ready** and collaborative: A marketing employee can create an activity, attach an invoice as a screenshot and create a Add comment "@Finance please check". The Finance user then receives an e-mail notification and can view all details (including unchanged original invoice). All

Actions are stored in the platform in a traceable way. In the event of an audit, every step from application to disbursement can be traced through the comments and attachments – changes are not possible silently, as new comments act as an audit trail.
This phase thus also meets many **SOX compliance** requirements: An **audit trail** of all Changes/communication are automatically recorded [1], and **role-based access control** and storage immutability protect data from unauthorized access or deletion [14].

**PHASE 7: NOTIFICATIONS & WORKFLOW (TEAMS INTEGRATION)**

- **Goal:** To increase the proactivity of the system through **automated notifications** and initial workflow support (e.g. approval processes), including integration into **Microsoft Teams**.
- **Description:** Phase 7 builds on and expands on the email notifications from Phase 6. Firstly, further **email notifications** are implemented, e.g. for status changes (if an activity is set to "Pending Approval", the finance manager should receive an email). Secondly, it is examined which of these events can be meaningfully mapped in **teams**. Microsoft Teams offers **the ability to display formatted information and action items in chats/channels** with Adaptive Cards [15]. Specifically, in the case of a release request, the system could use an adaptive Card to a defined Teams user or group, which will provide the activity details and Approve/Reject *buttons*. When clicked, an **embedded webhook/bot** or via Microsoft **Graph Approvals API** played the decision back into the system. The new **Approvals API** of Graph even enables the creation of complete approval processes in the Teams ApprovalsApp context [11]. For the MVP, it's sufficient to send a chat or channel message with Adaptive Card via Graph API. To do this, the application must be registered as a **Teams app** with bot permission (or more simply via a Power Automate flow, if allowed). Furthermore, the entire web app can be provided as a **Teams tab** : For this purpose, a Teams app package (XML Manifest) is created that points a *personalTab* to the URL of the static web app. With Azure AD Single sign-on (a special OAuth implicit flow for teams), the user can then download the app within Use Teams without logging in again. This integration increases adoption as users find notifications and editing options directly in Teams. In this phase, a **notification concept should also** be created: which actions trigger emails, which go into Teams, can the user adjust the frequency or unsubscribe? Possibly administration in the user profile (opt-in/opt-out for certain notifications).
- **Folder:**
- *api/NotifyStatusChange/index.ts* – Function that is triggered on certain status changes (via event or directly in the update function), and sends either mail or Teams message.
- *teams/AppManifest/* – Files to define the Teams app (manifest.json, icons, etc.)
- *teams/CardTemplates/* -- JSON templates for Adaptive Cards (can be in code as a string or separately)
- *api/TeamsWebhook/index.ts* - (Optional) an HTTP webhook that an Adaptive Card calls on button click to process Approve/Reject.
- **Recommended tools:** Microsoft Graph SDK (for Sending Chat Message with Adaptive Card Payload), Adaptive Cards Designer (for pre-layout the Card JSON), ngrok (testing Teams webhooks locally), Office 365 Connectors (alternative: setting up a connector hook in Teams for simple card posts), Power Automate (as a temporary solution if custom dev is too time-consuming: e.g. Flow listens for DB event via Power Automate trigger, Approval Card).
- **Snapshot point:** The system now **proactively** informs users. Final test could be: Marketing creates activity and marks it "for approval" -> Finance leader gets a Adaptive Card and by e-mail the request. He clicks "Approve" in Teams > changes in the web app The status of the activity is set to *Approved* and Marketing is notified in turn. This cycle shows the full added value of the platform and integrates with existing communication flows.

*(Optionally, further phases could be defined as needed, e.g. performance optimization, reporting dashboards, handover to operations team, etc. However, the main requirements have been met by phase 7 fulfilled.)*

## Compliance requirements (SOX, etc.)

The application must meet the requirements of a **US public company** in the area of financial controls, especially with respect to **Sarbanes-Oxley (SOX).** This results in the following focal points:

- **Audit trail & documentation:** All changes to financially relevant data (budgets, approvals) as well as all communication steps must remain traceable. Our concept ensures this through **version history and immutable comments**. Each activity has a complete history from attachment, changes (status changes, amount adjustments) to releases, which can be recorded in the background, for example, in a history table. In addition, the comment history acts as an audit trail. This is in line with the requirement to *track all changes to financial data chronologically* [1]. During implementation, care must be taken to ensure that **deletion** or **subsequent editing** of important data records is either prohibited or is logged in an audit-proof manner. For example, you could not even provide for the deletion of an activity – instead use the "canceled" status so that records are preserved.

- **Roles and Internal Controls:** Strict role-based **access control** is a must. Only authorized persons are allowed to grant approvals (e.g. only Finance is allowed to set the status to Approved), only certain roles are allowed to see budget fields, etc. Azure AD as a central IAM serves two purposes: 1) secure login (MFA, company password) and 2) central maintenance of roles outside the app (no local user administration required). The app verifies the user's role before any security-related operation. In addition, it is ensured that no **bypass of the UI** is possible: All security-critical checks are carried out **on the server side** in the Azure Functions. Even if a malicious user were to try to do something illegal via API call, the backend check rejects it. This implements an important SOX control: **Segregation of Duties** – e.g. the one requesting an expense cannot release it themselves, this enforces the role model. In addition, sensitive data is encrypted both in the DB and secured in transit (TLS). **RBAC and encryption protect sensitive financial data from unauthorized access or modification** [14].

- **Data retention:** Financially-relevant data must usually be **7-10 years** (depending on local laws, but SOX requires that records remain available for at least 5 years after the end of the year). The use of Azure SQL and Blob Storage with backup/retention configuration ensures that no data is lo In particular, Azure Blob's WORM storage feature guarantees that uploaded document files *remain* immutable for the predefined period of time [7]. We plan to set the retention period in consultation with compliance (e.g. 7 years for attachments, 10 years for DB backups).

- **Documentation & transparency:** For review (internal or by auditors), the functioning of the platform should be documented. Therefore, we will create **technical documentation** (data model, authorization concept, configuration parameters) and make it available to the Compliance Team. Auditors can also request **read-only access** to the dashboard or a data export to check samples. Our software supports this by providing a **reporting module** or at least complete Excel exports of activities and histories.

In summary, the system addresses the SOX-relevant criteria through **access controls, logging and data integrity**. A quote on this: *"A key requirement of SOX compliance is maintaining an audit trail of changes to financial data... [Software] can automatically capture and maintain an audit trail of all changes... ensuring transparency and accountability."* [1] – this is exactly what our platform does.

## Data modeling and API protection recommendations

**Data modeling:** The relational schema should be designed to ensure **consistency** and **performance**. It is important to avoid **redundancy** (3NF if it makes sense) – e.g. we do not store user or unit names multiple times in each activity, but reference IDs and keep master data separate (if not exclusively via AD). Some recommendations:

- The **Activities** table contains core fields (name, description, financial parameters, status, timestamp, foreign key to creator and possibly current responsibility). For status, a **status lookup** (own table or enum in code) is recommended to use consistent values.
- **Comments** should have ActivityID as FK and maybe ParentCommentID (for threaded discussions, if necessary). Comments are **rather not editable**, so that no historicization per comment is necessary – every new comment is final. If you want to allow edit, then either only within X minutes or save change as a new comment event.
- **Attachments** contain at least `AttachmentID` , `ActivityID` (or `CommentID` if you attach per comment), `BlobUrl` or Blob Name, `FileName` , `Uploader` ,

`UploadTimestamp` . The **blob name** can be a combination of ActivityID and a GUID to avoid name collision. Since blob storage is versioned/immutable, we don't need our own versioning in the schema – there are no updates, only new attachments.

- **Users/Units/Roles:** Since we use Azure AD, we don't need a full-fledged user management table. But we can provide a **UserSettings** table to store user preferences (e.g. default view, notification settings). There you would create an entry via User's AAD Object ID. For **units**: if the affiliation results directly from the role, this is sufficient, alternatively a unit table (ID, name) and in UserSettings a UnitID column to define manually.
- **Audit log:** To be absolutely sure, an **ActivityHistory** table can be kept, which saves a snapshot every time an activity entry is changed (with Timestamp, User, field changes). This would make it possible to track exactly which fields were changed and when. This table can be large, so consider partitioning by year or similar if necessary.
- **Views (saved views):** if implemented, table `views` with `ViewID, User, Name, FilterJSON` . FilterJSON is a text box with a JSON that stores the criteria.

When modeling, you should also **pay attention to indexes: e.g. index to Activity (Status) for Kanban columns, to Activity (Unit) if filtered frequently by it, full-text index to fields for global search (or use Azure Cognitive Search as an alternative, but probably overkill).**

**API Protection:** The Web API is protected by several layers:

- **Authentication:** Each API route requires a valid JWT token (passed from the frontend after login). In cloud operation, Azure Static Web Apps already takes care of authentication at the infrastructure level – i.e. unauthenticated requests to protected routes do not get through to the function at all. In addition, our function validates the **role** (claim check) for safety-critical calls.

- **Rate Limiting / Throttling:** Since this is an internal tool for a limited number of users, abuse is unlikely. However, a simple **rate limit** per user/IP could be introduced for certain expensive endpoints, e.g. to prevent spam from automated requests (Azure Functions can be limited via integration with API Management or self-coded counter).

- **Input validation:** All data coming from the client is validated on the server side, e.g. with `AddActivity` we check that required fields are present and in the valid value range. Strings are limited to length, numeric values to plausible limits. This is also how we prevent **SQL injection** – although an ORM usually handles this already in a parameterized way. In addition, we should **either escape or clean up special characters** (e.g. B. HTML tags in comments) to avoid XSS when viewing later.

- **SQL protection:** Since Azure SQL is used, we should use either stored procedures or prepared statements (via ORM) so that user input never goes directly into query strings. Parameterization protects against injections. In addition, the Azure SQL user (or the managed identity) can only be given the minimum necessary rights (only to the used tables).

- **Transport encryption:** Communication between frontend and backend runs over HTTPS (enforced by Azure Static Web App, including free SSL certificate [16]). The DB connection is also TLS-encrypted.

- **Secret management:** App keys, DB connection strings, API keys (e.g. for Graph API Client Secret) are **not stored in the code**, but managed in Azure **Key Vault** or as a GitHub Actions Secret and only injected into the functions as an environment variable. This keeps the code repository clean of sensitive data.

- **Logging & Monitoring:** Every API access and important operations should be logged (Azure Application Insights can be integrated with Functions). In the event of security-relevant events (e.g. repeated failed attempts, unauthorized access attempts), alerts could be triggered. This helps to identify any patterns of attack or abuse.

Overall, this creates a robust API that is not publicly accessible to the outside world through Azure Static Web App anyway (access to certain AAD tenants could be restricted). Internally, it meets all the best practices for security:

**Authentication, Authorization, Validation, Encryption**. This ensures that the application and data are **protected against common threats** and that only proper activities are taking place.

## Maintainability & handover to the customer team

For a long-term project, code **maintainability** and a smooth **handoff** to Westcon's internal team is critical. The following measures are planned for this purpose:

- **Clean code base & conventions:** From the very beginning, we pay attention to a clearly structured code (separation of components, hooks, utilities in the frontend; separation of business logic, database access, route handling in the backend). We use linting (ESLint) and formatting (prettier) with defined rules to ensure a consistent style. In addition, we write **meaningful comments** in more complex places and maintain a **ARCHITECTURE.md** document in the repo that explains the most important decision points.

- **Modularity & Reusability:** By using Tailwind + ShadCN, we are building our own component library, which is well documented. This allows future developers to easily create new UI elements in the existing style. Similarly in the backend: By encapsulating the business logic in separate service functions (instead of writing everything in the function handler), we make testing and extending easier.

- **Automated testing:** We set up a **testing framework** in later phases. At least unit tests for critical logic (e.g. calculation of budget differences, checks) and ideally integration tests for API (e.g. with an in-memory SQLite for Prisma to simulate the DB). The frontend can also be fundamentally tested with Jest/React Testing Library (e.g. login flow, rendering of views). These tests are not only intended to catch bugs but also serve as living documentation of the expected behavior.

- **CI/CD pipeline:** Azure Static Web Apps already has a CI/CD; we will adapt it, e.g. to run tests before each deployment and to prevent deployment if it fails. Security checks (e.g. with GitHub Dependabot for npm package updates) are also enabled. A staging environment (e.g. via SWA Preview Environments or a separate Azure SWA instance) is used to have new features approved by the customer team before going into production.

- **Documentation & Training:** In addition to the in-code docs, a **user manual** for end users will be created so that new employees can quickly learn how to use the MDF dashboard. More importantly, an **admin/developer manual** for the Westcon IT team. In it, we describe the setup of Azure resources, CI/CD, as well as typical maintenance tasks (e.g. role management in AD, rotation of secrets, scaling if necessary). After the end of the project, we plan a **handover workshop** in which we explain the code to the customer team, go through the deployment pipeline together and clarify open questions.

- **Logging & error handling:** For the operational phase, we equip the application with sufficient **telemetry**. Azure Application Insights captures exceptions in Functions; in the frontend, we could include a tool like Sentry to track UI errors. This allows the maintenance team to proactively respond to problems. Important events (e.g. graph API errors when sending emails) may also be reported via monitor alert.

- **Continuous improvement:** We recommend introducing a **rule process** in which the system is regularly reviewed and improved (e.g. semi-annual security audit of the application, performance review of the SQL – especially because data volumes could grow over the years). It should also be possible to record future user requirements, for which we keep and prioritize a **backlog**. The architecture is deliberately *extensible* – e.g. it would be relatively easy to flange a module for **reporting** (KPI dashboards on MDF usage) or make a mobile-friendly adaptation without changing core components.

These measures will ensure that Westcon benefits from the platform in the long term. Even if the original developers are no longer available, the Westcon team can understand and develop the code. The use of common technologies (React, Node, Azure) ensures that professionals with these skills are available. In addition, we not only protect data integrity through extensive compliance and security mechanisms, but also facilitate future **audit processes**, as all relevant information is available centrally and neatly structured.

**Conclusion:** The presented concept provides a comprehensive solution for the management of MDF activities. It meets business requirements (dashboard, collaboration, workflow) and non-functional requirements (security, compliance, performance, maintainability) in equal measures. Thanks to the modular phase plan, Westcon can see interim results early on and provide feedback as we work towards the final product step by step. With the Westcon Marketing Platform – MDF Activities Dashboard, the handling of marketing development funds becomes more transparent, efficient and audit-proof – a win for all departments involved and a further step towards the digital transformation of business processes.

**Sources:**


- Microsoft Learn - Azure Static Web Apps Overview [2] [3]
- Microsoft Learn - Assign custom roles via Azure AD Graph (Azure Static Web Apps) [5] [6]
- Johannes Schmidt – RBAC with Azure Static Web Apps & App Roles (Medium, 2023) [17] [18]
- Microsoft Learn – MSAL React Doc (Authentication in React) [4]
- Microsoft Learn - Microsoft Graph API (example for sending emails) [9] [10]
- Microsoft Learn – Teams Approvals API Overview
- Microsoft Learn – Azure Blob Storage Immutable (WORM) Overview [7] [8]
- McKonly & Asbury – *Leveraging a Software Tool for SOX Compliance* (Insights, 2024) [1] [14]


---


[1] [14] Software Tools SOX Compliance | Risk Assessment Monitoring
https://macpas.com/leveraging-a-software-tool-for-sox-compliance/

[2] [3] [12] [16] What is Azure Static Web Apps? | Microsoft Learn
https://learn.microsoft.com/en-us/azure/static-web-apps/overview

[4] React single-page application using MSAL React to authenticate users against Microsoft Entra External ID - Code Samples | Microsoft Learn
https://learn.microsoft.com/en-us/samples/azure-samples/ms-identity-ciam-javascript-tutorial/ms-identity-ciam-javascripttutorial-1-sign-in-react/

[5] [6] [13] Tutorial: Assign Azure Static Web Apps roles with Microsoft Graph (preview) | Microsoft Learn
https://learn.microsoft.com/en-us/azure/static-web-apps/assign-roles-microsoft-graph

[7] [8] Overview of immutable storage for blob data - Azure Storage | Microsoft Learn
https://learn.microsoft.com/en-us/azure/storage/blobs/immutable-storage-overview

[9] [10] user: sendMail - Microsoft Graph v1.0 | Microsoft Learn
https://learn.microsoft.com/en-us/graph/api/user-sendmail?view=graph-rest-1.0

[11] Approvals app APIs - Microsoft Graph | Microsoft Learn https://learn.microsoft.com/en-us/graph/approvals-app-api

[15] Overview of adaptive cards for Teams - Power Automate
https://learn.microsoft.com/en-us/power-automate/overview-adaptive-cards

[17] [18] Role Based Access Control with App Roles in Azure Static Web Apps | by Johannes Schmidt | Medium
https://johschmidt42.medium.com/role-based-access-control-with-app-roles-in-azure-static-web-apps-f6525d1b7d92

## Enhanced Scope Additions – 2025 Alignment

The following features were confirmed and added to meet expanded compliance, collaboration, and flexibility requirements:

- ◇ Event-Based Budgeting: Users can define a total event budget and add multiple vendor funding lines against it. The system tracks used vs. remaining budget.
- ◇ Unit-Specific Forms: Each business unit (Marketing, Finance, etc.) can have customized data entry forms tailored to their needs and permissions.
- ◇ Admin-Controlled Dynamic Columns: Unit admins can add new data fields dynamically during the process. These columns can be rendered per-unit and stored in SQL or JSON structures.
- ◇ Personal & Shared Views: Users can create personal views with saved filters and sorts. Admins can publish shared views for all users, similar to Airtable.
- ◇ Secure Uploads & Comments: Users can attach screenshots (e.g., invoice confirmations) and leave structured comments. These can block progression if required data is missing.
- ◇ ECharts Integration: For advanced and interactive data visualization (as a modern open-source alternative to Power BI).
- ◇ Notification System: Phase 1 – email notifications via Microsoft Graph. Phase 2 – Teams Adaptive Cards with Approvals integration.
- ◇ Azure AD-based Permissions: User access is controlled by role and unit using Azure AD. Supports column-level visibility and per-entry restrictions.
- ◇ Compliance & Audit Logging: Full traceability for changes, uploads, and approvals in compliance with SOX/public company standards.

**Event-Based Budget Model**

| Concept | Implementation |
|---|---|
| **Event** | One Event entry with TotalBudget, metadata |
| **Funding Lines** | Child table: FundingLines with Vendor, Amount, Currency, Notes etc. |
| **Tracking** | Auto-calculate: RemainingBudget = Total - SUM(Lines) |
| **UI** | Event form with embedded dynamic sub-table (“Add Funding Line”) |
| **Validation** | Real-time check: prevent overspending if required |

**Flexible Schema Strategy (Central Table + Unit Extensions)**

| Feature | Approach |
|---|---|
| **Master Table** | Stores core activity/event fields (ActivityID, Owner, Budget, etc.) |
| **Dynamic Columns** | Stored as JSONB-like structure (CustomFields) or added to SQL if stable |
| **UI Per Unit** | Unit-specific forms show only relevant fields |
| **Column Visibility Control** | Based on role (only unit admins see “Add Field” button) |
| **Versioning** | Optional: log schema change per unit (audit trail) |
| **Field Library** | Admin selects field type (Text, Dropdown, Number, Attachment, etc.) |

Implement a **view management system** that supports:

- **Shared Views**: Created by admins or power users, visible to all
- **Personal Views**: Created by individuals, saved under their user profile
- Stored either in:
  - LocalStorage (for POC)
  - Azure SQL (for persistence, later)

You’ll be able to:

- Save filter/group/sort state
- Set default view
- Switch between views easily (e.g. dropdown)