# KARMACASH STRATEGY DOCUMENT

## Table of Contents

1. [Introduction](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-1-introduction)
   * [1.1 Purpose of This Document](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#11-purpose-of-this-document)
   * [1.2 Background](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#12-background)
   * [1.3 Document Goals](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#13-document-goals)
2. [Project Vision, Goals & Target Audience](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-2-project-vision-goals--target-audience)
3. [Core Features & User Experience](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-3-core-features--user-experience)
4. [Technical Architecture](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-4-technology-stack)
5. [High-Level Architecture & Data Model](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-5-high-level-architecture--data-model)
6. [Development Plan & Milestones](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-6-development-plan--milestones)
7. [AI Collaboration Workflow](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-7-ai-collaboration-workflow)
8. [Healthy Work Practices Guidelines](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-8-healthy-work-practices-guidelines)
9. [Commercial Considerations & Cost Analysis](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-9-commercial-considerations--cost-analysis)
10. [Competitive Analysis & Differentiation](https://docs.google.com/document/d/1csB1Fesuiz2sHFB4e4-wA4oFI92B3V47EJpESlLmbHU/edit#section-10-competitive-analysis--differentiation)

## SECTION 1: INTRODUCTION

### 1.1 Purpose of This Document

This Rebuilding Strategy Document serves as the comprehensive blueprint for rebuilding the KarmaCash personal budgeting web application from scratch. It establishes a clear vision, defines technical requirements, outlines the development approach, and serves as the central reference point ("The Bible") for all project stakeholders.

### 1.2 Background

The original personal budgeting web application was built using AI tools (Claude, Cursor AI) and deployed to Firebase, using Google Sheets as a database. However, over time, the local/modified codebase became unmanageable, and crucially, access to the source code of the working deployed version on Firebase is currently unavailable. Rather than attempting to debug the modified code or retrieve the deployed version, the decision has been made to rebuild the application from scratch with a cleaner foundation.

### 1.3 Document Goals

This document aims to:

* Define a clear project vision and scope for the rebuilt KarmaCash application
* Establish detailed requirements and user stories
* Outline the technical architecture and data model
* Provide a structured development plan with defined milestones
* Document key calculation logic to ensure financial accuracy
* Establish a consistent AI collaboration workflow between Claude and Cursor
* Set guidelines for healthy work practices
* Define commercial considerations and future scalability options

## SECTION 2: PROJECT VISION, GOALS & TARGET AUDIENCE

### 2.1 Vision

To create a clean, intuitive, and reliable personal budgeting web application that helps users gain complete control over their finances through accurate tracking, insightful visualization, and effective budget planning. The application will evolve from a personal tool to a potential commercial product that can serve a wider audience.

### 2.2 Primary Goals

* **Rebuild with Clean Architecture**: Develop a well-structured, maintainable codebase using modern web technologies and best practices.
* **Enhanced User Experience**: Create an intuitive interface that makes budget management simple and accessible, with particular attention to efficient transaction entry and meaningful visualization.
* **Financial Accuracy**: Ensure all financial calculations (especially rollover, available funds, and category spending) are 100% accurate and transparent to users.
* **Data Integrity & Persistence**: Implement a robust database structure using Firebase Firestore, ensuring user data is securely stored and readily available across devices.
* **Commercial Potential**: Design the application with scalability in mind, establishing a foundation that can eventually support a freemium model with premium features.

### 2.3 Target Audience

#### Primary Audience (MVP)

* Individual users and couples looking for a simple yet powerful budgeting tool
* Users comfortable with technology who want more control than simple banking apps offer
* French-speaking users (specifically fr-CA locale)

#### Secondary Audience (Post-MVP)

* Small families or households wanting to collaborate on a shared budget
* Users seeking advanced features like savings goals and future projections
* Financially-minded individuals looking to optimize their spending and saving habits

### 2.4 MVP vs. Post-MVP Feature Comparison

| **Feature Area** | **MVP (Included)** | **Post-MVP (Future Development)** |
| --- | --- | --- |
| **Authentication** | Email/password login, session persistence | Social login options, account recovery |
| **Transactions** | Add/edit/delete, basic recurring, categorization | Future projections, receipt scanning |
| **Budgeting** | Monthly zero-based budget, category allocation | Goal-based savings, custom periods |
| **UI/UX** | Weekly/monthly views, basic charts | Enhanced visualizations, custom reports |
| **Sharing** | None | Household sharing, permissions system |
| **Notifications** | None | Spending alerts, recurring reminders |
| **Integration** | None | Export options (no bank connections) |

## SECTION 3: CORE FEATURES & USER EXPERIENCE (MVP FOCUS)

### 3.1 Core Feature List (User Stories - MVP)

#### Authentication & User Management (MVP)

* As a user, I want to securely sign up with my email and password, so that I can create a personal KarmaCash account.
* As a user, I want to securely log in with my email and password, so that I can access my financial data.
* As a user, I want to be able to log out, so that I can end my session securely.
* As a user, I want the application to remember my login session (e.g., via Firebase Auth persistence), so that I don't have to log in every time I open it.

#### Transaction Management (MVP)

* As a user, I want to add an expense transaction with date, description, amount, and category using a dedicated form, so that I can track my spending.
* As a user, I want to add an income transaction with date, description, amount, and category/source using a dedicated form, so that I can track my earnings.
* As a user, I want the transaction form to provide a date picker for selecting the transaction date.
* As a user, I want to view my transactions listed chronologically and grouped by day within the selected period (week or month), so I can review daily activity.
* As a user, I want these daily transaction groups to be expandable/collapsible, so I can manage the display detail.
* As a user, I want to delete a specific transaction (e.g., via a button or swipe gesture), so that I can correct mistakes.
* As a user, I want transactions identified as 'Recurring' in the database to be visually marked in lists, so I can distinguish them.
* As a user, I want to define rules for recurring transactions (e.g., 'Rent, $1500, Expense, Monthly on 1st'), so the system can automatically generate expected future transactions.
* As a user, I want to view generated recurring transactions (both past and future based on rules), so I can see my commitments.
* As a user, I want to edit the details of a recurring transaction rule, so I can update it if things change.
* As a user, I want to delete a recurring transaction rule, which should also offer to remove associated future generated transactions, so I can stop tracking it.

#### Category Management (MVP)

* As a user, I want KarmaCash to start with a default set of categories (e.g., Épicerie, Transport, Resto, Santé, Loisirs, Autres), so I have a basic structure.
* As a user, I want to add my own custom expense and income categories, so that tracking aligns perfectly with my habits.
* As a user, I want to edit the names of my custom categories, so I can refine my organization. (Assumption: Default categories might be non-editable)
* As a user, I want to delete custom categories (potentially re-assigning existing transactions), so I can clean up my category list.
* As a user, I want the system to categorize transactions based on their assigned category, handling the 'Santé' reimbursements (revenue offsetting expenses within the category) and assigning uncategorized expenses to 'Autres'.

#### Budgeting (MVP)

* As a user, I want to view a dedicated Budget section that operates on a monthly basis, independent of the global week/month view, so that I can manage my monthly plan.
* As a user, in the Budget section for a selected month, I want to see my 'Fonds Disponibles' correctly calculated based on that month's income, identified recurring expenses, and the accurate rollover (positive or negative) from the previous month's budget, so I know how much I can allocate.
* As a user, I want to set a target budget allocation amount for each category (default and custom) for the current month, so I can plan my spending.
* As a user, I want to see the actual amount spent in each category (calculated from transactions) compared to my allocated budget for the month, with visual indicators of progress/status, so I can track my performance.
* As a user, I want to see the 'Reste à Allouer' amount update in real-time as I adjust category allocations, helping me achieve a zero-based budget.
* As a user, I want my budget allocations for each month to be saved persistently in Firestore, linked to my user account, so they are available across devices and sessions.
* As a user, I want the budget rollover (unspent/overspent amount) from the previous month to be correctly calculated (handling negative rollovers) and automatically applied to the next month's 'Fonds Disponibles', ensuring budget continuity.

#### Visualization & Navigation (MVP)

* As a user, I want to switch between a Weekly and Monthly view for displaying transactions and analysis in the Transactions, Graphiques, and Catégories sections, so I can see different time perspectives.
* As a user, I want Previous/Next buttons to navigate through weeks or months in these display sections, updating the data shown.
* As a user, I want the selected date range and view mode (week/month) to persist as I navigate between the Transactions, Graphiques, and Catégories sections, so my context isn't lost.
* As a user, I want a 'Graphiques' section displaying a pie chart of expense breakdown by category and a bar chart of income vs. expenses vs. balance for the selected period, so I can visually understand my finances.
* As a user, I want a 'Catégories' section that lists transactions grouped by category (including custom ones) with totals for the selected period, making it easy to analyze spending patterns.
* As a user, I want the monthly calendar view within the 'Transactions' section to display a summary total amount (sum of the day's transactions) on each day instead of just a dot, so I get a quick financial overview of the day.
* As a user, I want a toggle switch (e.g., top-right labeled 'Solde: Cumulatif / Période') to control how the main balance displayed in the header is calculated: either as a running total up to the end of the selected period ('Cumulatif') or reflecting only the net change within the selected period ('Période').

### 3.2 Core Feature List (User Stories - Post-MVP / Future)

#### Notifications

* As a user (specifically one of a couple/group sharing a budget), I want to receive a push notification on my registered devices when another member adds a transaction, so we have real-time awareness of shared finances.
* As a user, I want to manage my notification preferences within KarmaCash.

#### Future Transactions / Projections

* As a user, I want future-dated transactions (manually added or generated by recurrence rules) to be visually distinct (with subtle styling aligned with [B3.8 v2]), so I can differentiate planned vs. past events.
* As a user, I want an optional notification on the day a future-dated transaction occurs, prompting me to confirm or adjust it, so I can verify planned expenses/income.

#### Budget Section Enhancements

* As a user, I want to drill down into the 'Fonds Disponibles' calculation to see the contributing revenue, recurring expenses, and rollover amounts.
* As a user, I want budget forecasting features that align with the Zen/Tranquility theme's mindful approach to financial planning.
* As a user, I want detailed savings goals tracking with the calming visual indicators described in [B3.11].
* As a user, I want reporting and export capabilities that maintain the clear, uncluttered aesthetic of KarmaCash.

#### Goal-Based Savings Categories (YNAB-inspired)

* As a user, I want to create specific budget categories designated as 'Savings Goals' (e.g., "Vacation Fund," "Down Payment"), so I can track progress towards specific financial objectives.
* As a user, when creating a Savings Goal category, I want to set a target amount and optionally a target date, so KarmaCash can help me track my progress.
* As a user, I want KarmaCash to visualize my progress towards my defined Savings Goals when I allocate funds to those categories, using the gentle animation principles defined in [B3.11].

#### Shared Budgets (Shareroo-inspired)

* As a budget owner, I want to invite other registered users (e.g., family members, partner) to access and collaborate on my KarmaCash budget, so we can manage finances together.
* As an invited user, I want to accept an invitation to join a shared budget, so I can view and contribute to it.
* As a member of a shared budget, I want transactions added by any member to be visible to all members, so we have a unified financial picture.
* As a member of a shared budget, I want budget allocations and spending progress to be visible and potentially editable by all members (permissions might be needed later), fostering collaboration.

#### Financial Mindfulness Features

* As a user, I want optional reflective prompts when entering significant transactions, encouraging mindful spending decisions aligned with my financial goals.
* As a user, I want periodic calm financial "check-ins" that help me reflect on my spending patterns without judgment.
* As a user, I want to set personal financial intentions within KarmaCash that align with the Zen/Tranquility theme's focus on mindfulness.

### 3.3 Key User Flows (MVP)

This section documents the primary user journeys within KarmaCash, following the Zen/Tranquility principles of clarity, focus, and ease of use defined in [B3.4]. Each flow is optimized for a calm, mindful user experience.

#### User Onboarding (Sign Up & First Login)

1. User visits the KarmaCash PWA URL.
2. User clicks 'Sign Up' (styled according to [B3.8 v2]).
3. User enters email and password in a clean, focused form.
4. User clicks 'Submit' button (Primary Sage Green from palette [B3.9.2]).
5. Firebase Authentication creates the user account.
6. User is automatically logged in and redirected to the main application view (likely 'Transactions' section).
7. A corresponding user document is created in Firestore /users/{userId}.
8. Default categories are pre-populated in /users/{userId}/categories.
9. User receives a gentle welcome message introducing the core features of KarmaCash.

#### User Login

1. User visits the KarmaCash PWA URL.
2. User clicks 'Log In'.
3. User enters email and password in a clean form aligned with [B3.8 v2].
4. User clicks 'Submit'.
5. Firebase Authentication verifies credentials.
6. During verification, a subtle loading animation is shown (as per [B3.11]).
7. User is logged in and redirected to the main application view with a gentle transition.
8. App loads user-specific data based on their UID.

#### Adding a Transaction

1. User navigates to the 'Ajouter' section.
2. User is presented with a calm, focused form with generous spacing ([B3.8 v2]).
3. User selects the date using the date picker (styled with Zen aesthetic).
4. User selects the type ('Dépense' or 'Revenu').
5. User selects a category from the dropdown (includes default and custom categories).
6. User enters the amount.
7. User enters a description.
8. User optionally selects recurrence details (if adding a recurring rule).
9. User clicks 'Ajouter' (Add) button (Primary Sage Green color).
10. Frontend validates the input with gentle feedback animations ([B3.11]).
11. Frontend sends transaction data to Firestore (/users/{userId}/transactions).
12. Upon successful save, a subtle confirmation message appears with a gentle animation.
13. Relevant UI sections update with smooth transitions ([B3.11]).

#### Setting Monthly Budget Allocations

1. User navigates to the 'Budget' section.
2. The app determines the current month (e.g., '2025-04').
3. The app fetches or creates the budget document for that month from Firestore (/users/{userId}/budgets/2025-04).
4. The UI displays categories with current spending and allocation controls, using the Zen/Tranquility styling defined in [B3.8 v2].
5. The UI displays calculated 'Fonds Disponibles' and 'Reste à Allouer' in a prominent, calming display ([B3.12]).
6. User adjusts the allocated amount for one or more categories using interactive controls.
7. The 'Reste à Allouer' display updates in real-time with subtle animation ([B3.11]).
8. User clicks 'Sauvegarder' (Save) styled with Primary Sage Green.
9. Frontend sends the updated allocations map to the specific monthly budget document in Firestore.
10. A gentle confirmation message appears.

#### Viewing Transactions (Weekly/Monthly)

1. User navigates to the 'Transactions' section.
2. App defaults to the current week/month view based on persisted state or default.
3. User uses the 'Semaine'/'Mois' toggle to switch views (styled as per [B3.8 v2]).
4. User uses the Previous/Next controls (using Heroicons [B3.9.5]) to navigate time periods.
5. The UI fetches transaction data for the selected period from Firestore.
6. Content transitions smoothly between views ([B3.11]).
7. Data is displayed either as expandable daily cards (weekly view) or a calendar grid with daily totals (monthly view), both following the Zen aesthetic.

#### Adding a Custom Category

1. User navigates to the 'Ajouter' section and finds the 'Gérer les catégories' (Manage Categories) option.
2. User clicks 'Ajouter catégorie' (Add Category).
3. User enters the category name and selects type (income/expense) in a clean, focused form.
4. User clicks 'Save'.
5. A new category document is created in Firestore (/users/{userId}/categories).
6. The UI confirms the addition with a subtle animation ([B3.11]).
7. The new category becomes available in transaction form dropdowns and budget allocation.

#### Defining a Recurring Transaction Rule

1. User navigates to a dedicated 'Manage Recurring' area (or via the 'Ajouter' form).
2. User clicks 'Add Recurring Rule' button styled according to [B3.8 v2].
3. User enters details in a clean, well-spaced form: description, amount, category, type, frequency (e.g., 'Monthly', 'Weekly', 'Bi-Weekly', 'Annually'), interval (if applicable), start date, and relevant day.
4. User clicks 'Save'.
5. A new rule document is created in Firestore (/users/{userId}/recurringRules).
6. A gentle confirmation appears with a subtle animation ([B3.11]).
7. (Backend/Scheduled Function) This rule is now active for generating future transaction instances.

#### Deleting a Transaction

1. User views transactions in a list (e.g., daily view, category view).
2. User swipes left on a transaction item (with smooth animation per [B3.11]).
3. A 'Supprimer' (Delete) button appears, styled with the Negative accent color (Soft Terra Cotta).
4. User taps 'Supprimer'.
5. A calm confirmation prompt appears with gentle animation.
6. Upon confirmation, transaction is removed from Firestore.
7. The UI updates with the item smoothly disappearing ([B3.11]).

### 3.4 UI/UX Guidelines - Zen/Tranquility Theme

These guidelines define the core UI/UX principles for KarmaCash, emphasizing the Zen/Tranquility theme established in [B3.9] and [B3.10], while ensuring a calm, focused, and mindful user experience.

#### Calm & Minimalist Aesthetic

* Create a serene, uncluttered interface that reduces financial anxiety
* Use muted, natural colors from the Japandi-inspired palette ([B3.9.2])
* Incorporate generous whitespace and clean typography
* Avoid visual noise or excessive elements that could create stress when dealing with financial information

#### Clarity & Focus

* Prioritize information hierarchy to guide users' attention to what matters most
* Use subtle visual cues rather than harsh contrasting elements
* Each screen should have a clear purpose with minimal distractions
* Support mindful financial decision-making through focused interface design

#### Responsive Design

* Ensure the application works seamlessly across devices (mobile to desktop)
* Maintain the calm aesthetic across all viewport sizes
* Use responsive techniques (flexbox, grid, media queries)
* Pay special attention to comfortable spacing on all screen sizes as defined in [B3.8 v2]

#### Component-Based Architecture

* Build the UI using reusable React components that follow the styling principles in [B3.8 v2]
* Maintain consistency in component behavior, styling, and animation across the application
* Create a component library that reflects the Zen/Tranquility principles

#### Consistent Navigation

* Implement intuitive, predictable navigation with gentle transitions between sections
* Keep tab-based main navigation ('Transactions', 'Graphiques', etc.) and date controls consistently placed and styled according to [B3.7]
* Ensure users always understand where they are within the application

#### Data Visualization Clarity

* Charts and graphs should convey financial information clearly without creating anxiety
* Use the softer, harmonious color scheme from [B3.9.2]
* Ensure adequate spacing between elements
* Implement the subtle animations defined in [B3.11]

#### Gentle Interactive Feedback

* Provide calm, reassuring feedback for user actions
* Use the micro-interactions defined in [B3.11] for form validation, button states, and loading indicators
* Toast notifications should appear and disappear with subtle animations
* Use appropriate semantic colors from [B3.8 v2]

#### Language & Localization

* Maintain French UI text as defined in [B3.6], with 'fr-CA' locale for date/currency formatting
* Ensure typography follows the guidelines in [B3.8 v2] for optimal readability of French text
* Currency symbol remains '$'

#### Accessibility & Inclusion

* Create an interface accessible to all users
* Ensure sufficient color contrast while maintaining the muted palette
* Make interactive elements clearly identifiable through subtle yet effective visual cues
* Support keyboard navigation and screen readers

#### Touch-Friendly Interactions

* Design for comfortable touch interaction with appropriately sized targets (minimum 44×44px as per [B3.8 v2])
* Implement refined touch gestures like swipe-to-delete for transactions with the subtle animations defined in [B3.11]

#### Progressive Disclosure

* Reveal complexity gradually, showing essential information first
* Allow users to access details as needed
* Support the Zen principle of maintaining calm while dealing with potentially complex financial data

#### Supportive Empty States

* Create thoughtful empty states that guide users with gentle encouragement
* Use supportive language and subtle visual cues to help users get started
* Avoid stark emptiness that could create uncertainty

#### Soothing Error Handling

* Present errors calmly using the Negative accent color (Soft Terra Cotta #C17C74) from [B3.8 v2]
* Avoid alarming experiences even when communicating problems
* Provide clear, actionable guidance for resolution

### 3.5 MVP Definition (KarmaCash)

The MVP for KarmaCash will be a functional personal budgeting web application built with React and Firebase (Auth, Firestore, Functions, Hosting), designed with a Zen/Tranquility aesthetic ([B3.4], [B3.8 v2]). It will allow users to:

* Securely sign up / log in ([B3.1])
* Manage custom income/expense categories ([B3.1])
* Add / delete transactions efficiently, including swipe-to-delete ([B3.1], [B3.3])
* Define and manage rules for recurring transactions ([B3.1])
* View transactions chronologically (grouped by day) within selectable weekly/monthly views ([B3.1])
* Utilize an enhanced calendar view for monthly transaction overviews ([B3.1])
* View clear graphical summaries (e.g., income vs. expense, category breakdown) styled calmly according to [B3.8 v2] ([B3.1])
* Manage a monthly zero-based budget with:
  + Accurate 'Fonds Disponibles' calculation ([B3.1], [B5.3])
  + Category allocation controls ([B3.1])
  + Clear spending tracking visuals ([B3.1])
  + Correct positive/negative rollover between months ([B3.1], [B5.3])
* Benefit from persisted date range and view mode selection across relevant sections ([B3.1])
* Use the cumulative/period balance toggle for flexible balance viewing ([B3.1])

Push notifications and future transaction projections ([B3.2]) are explicitly excluded from the MVP. The primary focus is delivering core budgeting functionality within a calm, clear, and supportive user experience.

### 3.6 French Terminology Glossary

This glossary provides a comprehensive reference of French UI text elements used throughout the application, their English equivalents, and descriptions of their purpose or context.

#### Navigation & Main Interface

| **French** | **English** | **Description** |
| --- | --- | --- |
| Transactions | Transactions | Main navigation tab for viewing and managing transactions |
| Graphiques | Graphs | Main navigation tab for charts and visual data representation |
| Catégories | Categories | Main navigation tab for viewing transactions grouped by category |
| Ajouter | Add | Main navigation tab for adding new transactions |
| Budget | Budget | Main navigation tab for budget allocation and tracking |
| Semaine | Week | Toggle button for weekly view |
| Mois | Month | Toggle button for monthly view |
| Aujourd'hui | Today | Button to navigate to current day/week/month |
| Solde | Balance | The current account balance display |
| Cumulatif | Cumulative | Toggle option for displaying running total balance |
| Période | Period | Toggle option for displaying balance within selected period only |

#### Date Navigation

| **French** | **English** | **Description** |
| --- | --- | --- |
| dim. | Sun. | Abbreviation for Sunday (Dimanche) |
| lun. | Mon. | Abbreviation for Monday (Lundi) |
| mar. | Tue. | Abbreviation for Tuesday (Mardi) |
| mer. | Wed. | Abbreviation for Wednesday (Mercredi) |
| jeu. | Thu. | Abbreviation for Thursday (Jeudi) |
| ven. | Fri. | Abbreviation for Friday (Vendredi) |
| sam. | Sat. | Abbreviation for Saturday (Samedi) |
| avril | April | Month name |

#### Transaction Management

| **French** | **English** | **Description** |
| --- | --- | --- |
| Ajouter une transaction | Add a transaction | Header for transaction creation form |
| Date | Date | Field label for transaction date |
| Type | Type | Field label for transaction type |
| Catégorie | Category | Field label for transaction category |
| Montant | Amount | Field label for transaction amount |
| Dépense | Expense | Transaction type option |
| Revenu | Income | Transaction type option |
| Récurrence (optionnel) | Recurrence (optional) | Field label for recurring transaction settings |
| Non récurrent | Non-recurring | Option for one-time transactions |
| Récurrent | Recurring | Label for recurring transactions |
| Supprimer | Delete | Button to delete a transaction |
| Supprimer série | Delete series | Button to delete a recurring transaction series |

#### Budget Management

| **French** | **English** | **Description** |
| --- | --- | --- |
| Allocation et suivi du budget | Budget allocation and tracking | Header for budget management section |
| Fonds disponibles | Available funds | Display of available funds for budgeting |
| Reste à allouer | Remaining to allocate | Amount left to allocate in zero-based budget |
| Économies ce mois-ci | Savings this month | Current month savings tracking |
| Économies totales | Total savings | Accumulated savings across all months |
| Économies du mois | Month savings | Savings calculation for current month |
| Économies accumulées sur tous les mois | Accumulated savings across all months | Description of total savings calculation |
| Alloué | Allocated | Label for amount allocated to category |
| Dépensé | Spent | Label for amount spent in category |
| Réinitialiser | Reset | Button to reset budget allocations |
| Sauvegarder | Save | Button to save budget allocations |

#### Categories

| **French** | **English** | **Description** |
| --- | --- | --- |
| Dépenses par catégorie | Expenses by category | Header for category expense breakdown |
| Répartition des dépenses | Expense distribution | Title for pie chart showing expense breakdown |
| Évolution du solde | Balance evolution | Title for bar chart showing income/expense/balance |
| Total des dépenses | Total expenses | Sum of all expenses |
| Épicerie | Grocery | Default expense category |
| Transport | Transportation | Default expense category |
| Resto | Restaurant | Default expense category |
| Santé | Health | Default expense category |
| Loisirs | Leisure | Default expense category |
| Autres | Others | Default category for miscellaneous expenses |
| Travail | Work | Example category |
| Cadeau | Gift | Example category |
| Assurance | Insurance | Example category |
| Gardienne | Babysitter | Example category |

#### Charts & Visualization

| **French** | **English** | **Description** |
| --- | --- | --- |
| Revenus | Income | Label for revenue in charts |
| Dépenses | Expenses | Label for expenses in charts |
| Solde | Balance | Label for net balance in charts |

#### Budget Calculations

| **French** | **English** | **Description** |
| --- | --- | --- |
| non dépensé moins dépassement | unspent minus overages | Description of how monthly savings are calculated |

#### Date Formats & Abbreviations

| **French** | **English** | **Description** |
| --- | --- | --- |
| avr. | Apr. | Abbreviation for April (Avril) |
| mars | March | Month name |

#### UI Actions & States

| **French** | **English** | **Description** |
| --- | --- | --- |
| Gérer les catégories | Manage categories | Option to administer categories |
| Ajouter catégorie | Add category | Button to create a new category |

### 3.7 UI Component Specifications (KarmaCash)

This section identifies and describes the distinct UI components that make up the KarmaCash interface, based on the user stories and design requirements, aligning with the Zen/Tranquility theme ([B3.4], [B3.8 v2]).

#### 3.7.1 Navigation Components

**Main Navigation Bar**

* **Purpose**: Provides primary navigation between main application sections
* **Description**: Horizontal navigation bar containing primary options: Transactions, Graphiques, Catégories, Ajouter, Budget. Visually styled according to [B3.8 v2]
* **Behavior**: Active item is clearly highlighted, potentially using the Primary Sage Green color or other subtle visual cues defined in [B3.8 v2]
* **Location**: Fixed at the top of the application below the header (Or potentially other locations like bottom navigation on mobile, as per responsive considerations in [B3.8 v2])

**Date Navigation Controls**

* **Purpose**: Allows navigation between different time periods
* **Description**: Contains previous/next interactive elements (using icons like Heroicons from [B3.8 v2]), date range display, and view mode toggles (e.g., 'Semaine', 'Mois', 'Aujourd'hui'). Employs generous spacing as per [B3.4]
* **Behavior**:
  + Left/right arrows navigate to previous/next period
  + Center displays current period (week range or month/year)
  + View toggles switch between Semaine/Mois/Aujourd'hui modes, styled per [B3.8 v2]
* **Location**: Consistently placed below the main navigation bar or relevant section header

**Period Toggle**

* **Purpose**: Switches between period-specific or cumulative balance display
* **Description**: Interactive toggle switch element, labeled "Solde" with "Cumulatif/Période" options. Visually styled as per [B3.8 v2] button/toggle specifications, ensuring adequate touch target size
* **Behavior**: Controls how the main balance is calculated and displayed in the App Header
* **Location**: Top-right corner of the header

#### 3.7.2 Header Components

**App Header**

* **Purpose**: Displays the application name/logo and current balance
* **Description**: Styled header area presenting the "KarmaCash" name/logo on the left and the primary balance amount on the right. Background uses theme colors (e.g., Primary Sage Green, muted neutral, or subtle gradient) as defined in [B3.8 v2]
* **Behavior**: Balance updates based on transactions and selected mode (Cumulatif/Période). Positive/Negative balances are indicated using the Muted Teal / Soft Terra Cotta accent colors from [B3.8 v2]
* **Location**: Fixed at the top of the application

#### 3.7.3 Transaction Components

**Transaction Day Card**

* **Purpose**: Groups transactions by day or other relevant time periods
* **Description**: Card container holding transactions for a specific day. Follows card styling (backgrounds, rounded corners, subtle shadows, generous padding) from [B3.8 v2]. Header displays the date, potentially differentiated by subtle background shift or color-coded elements, not necessarily a full colored bar ([B3.12]). Content area is expandable/collapsible
* **Behavior**: Can be expanded/collapsed (with gentle animation per [B3.11]) to show/hide transactions
* **Location**: In the Transactions section, stacked chronologically

**Transaction Item**

* **Purpose**: Displays individual transaction details
* **Description**: Row or card element displaying transaction details (e.g., category name, description, date, amount). Amounts use accent colors (Muted Teal / Soft Terra Cotta) from [B3.8 v2] to indicate income/expense. Optional recurring indicator styled subtly (e.g., small icon or text label using theme colors). Uses increased spacing between elements [B3.12]
* **Behavior**: Interaction (e.g., swipe, tap) reveals delete options with gentle animation/transition ([B3.11])
* **Location**: Within Day Cards or other transaction lists

**Transaction Form**

* **Purpose**: Allows creation/editing of transactions
* **Description**: Input form with fields for date, type, category, amount, etc. Uses form element styling (inputs, dropdowns, buttons, labels) and generous spacing defined in [B3.8 v2]
* **Behavior**: Validates input, submits data. Provides feedback using gentle micro-interactions (e.g., soft validation messages) as per [B3.11]
* **Location**: In the 'Ajouter' section or modal dialog

**Calendar View**

* **Purpose**: Displays monthly calendar view with transaction summaries
* **Description**: Grid-based calendar layout, styled for an airy, modern appearance with ample whitespace ([B3.12]). Days display dates and potentially summary amounts or subtle indicators for transaction activity, styled per [B3.8 v2]. Visual weight of grid lines is minimized or removed ([B3.12])
* **Behavior**: Indicates days with transactions, highlights current/selected day using theme accents [B3.8 v2]. May allow selection of a day to view/add transactions
* **Location**: Alternative view in the Transactions section (Month view)

#### 3.7.4 Budget Components

**Budget Allocation Bar**

* **Purpose**: Visualizes budget allocation and spending progress for a category
* **Description**: Progress bar styled with increased height and rounded corners per [B3.8 v2] / [B3.12]. Uses theme accent colors (e.g., Primary Sage Green for progress, Soft Terra Cotta for over-budget indication) for status. May include subtle patterns or transparency effects ([B3.12])
* **Behavior**: Visual indicator fills based on spending percentage, with smooth transitions ([B3.11]). Clearly indicates spent vs. allocated amounts
* **Location**: In the Budget section for each category

**Remaining to Allocate Indicator**

* **Purpose**: Shows amount remaining to be allocated in the monthly budget
* **Description**: Prominent display area (e.g., styled card) showing the amount left to allocate. Negative values are indicated using the Negative Accent Color (Soft Terra Cotta) defined in [B3.8 v2]
* **Behavior**: Updates dynamically with subtle animation ([B3.11]) as category allocations change
* **Location**: Typically at the bottom or top of the allocation list in the Budget section

**Monthly Savings Display**

* **Purpose**: Shows savings progress for the current month and potentially accumulated savings
* **Description**: Displays savings amount(s), potentially using progress bars styled per [B3.8 v2] and [B3.12], with clear labels and descriptions. May use visual metaphors for growth ([B3.12])
* **Behavior**: Updates based on budget calculations and transaction data, potentially with gentle animations ([B3.11])
* **Location**: In the Budget section

**Available Funds Display**

* **Purpose**: Shows funds available for budget allocation in the current month
* **Description**: Prominent display area, potentially a card-like element with subtle elevation [B3.12], clearly presenting the calculated available funds amount. Styled according to [B3.8 v2]
* **Behavior**: Calculated based on income, recurring expenses, and rollover
* **Location**: Typically at the top of the Budget section

#### 3.7.5 Visualization Components

**Expense Distribution Pie Chart**

* **Purpose**: Visualizes spending breakdown by category
* **Description**: Donut chart using a softer, harmonious color scheme derived from the palette in [B3.8 v2]. Presented with clear labels, potentially percentages, and increased spacing around elements. May use semi-transparency or explore alternative chart types that align with the Zen theme ([B3.12])
* **Behavior**: Size of segments reflects proportion of total spending. May include subtle hover/interaction states ([B3.11])
* **Location**: In the 'Graphiques' section

**Balance Evolution Bar Chart**

* **Purpose**: Compares income, expenses, and balance over the selected period
* **Description**: Bar chart using theme colors from [B3.8 v2]. Bars may have rounded caps or subtle gradient fills ([B3.12]). Axes and grid lines have reduced visual weight. Uses clear, readable typography for labels ([B3.8 v2])
* **Behavior**: Bar heights reflect relative amounts. Employs progressive animations on load/update ([B3.11])
* **Location**: In the 'Graphiques' section

#### 3.7.6 Category Components

**Category List Item**

* **Purpose**: Displays spending totals or budget status for a specific category
* **Description**: Row or card-like item displaying category name and relevant financial data (e.g., spent amount, remaining budget). May include a color indicator using category-specific theme colors ([B3.8 v2]). Utilizes increased row padding/spacing
* **Behavior**: Can be expandable (with subtle animation [B3.11]) to show associated transactions
* **Location**: In the 'Catégories' and 'Budget' sections

**Category Selector**

* **Purpose**: Allows selection of a transaction category
* **Description**: Dropdown menu styled according to form element specifications in [B3.8 v2] (colors, spacing, border radius). May include visual cues like color swatches alongside names ([B3.12])
* **Behavior**: Shows available categories for selection; opens/closes with subtle transitions ([B3.11])
* **Location**: In the transaction form

#### 3.7.7 Action Components

**Delete Button**

* **Purpose**: Allows deletion of items (transactions, categories, etc.)
* **Description**: Button styled using the Negative Accent Color (Soft Terra Cotta, #C17C74) defined in [B3.8 v2], likely with a subtle visual treatment (e.g., outline style or muted fill) rather than a strong solid color ([B3.12]). Labelled "Supprimer"
* **Behavior**: Triggers confirmation before destructive action. Includes gentle hover/active states ([B3.11])
* **Location**: Appears with deletable items (e.g., on swipe/hover, within item details)

**Delete Series Button**

* **Purpose**: Allows deletion of recurring transaction series/rules
* **Description**: Button styled similarly to the single Delete Button (using Soft Terra Cotta accent). May include additional text ("série") or an icon (from Heroicons [B3.8 v2]) to clearly differentiate it from single deletion ([B3.12]). Labelled "Supprimer série"
* **Behavior**: Triggers confirmation before deleting the rule
* **Location**: Appears with recurring transaction items/rules

**Save Button**

* **Purpose**: Commits changes (e.g., budget allocations, form entries)
* **Description**: Primary action button styled using the Primary Accent Color (Sage Green, #919A7F) or potentially a subtle gradient, as defined in [B3.8 v2]. Labelled "Sauvegarder", "Ajouter", etc. depending on context
* **Behavior**: Triggers data saving. Includes standard button interactions ([B3.11]) and potential success/loading states
* **Location**: Forms, Budget section, etc.

**Reset Button**

* **Purpose**: Reverts changes or clears form fields
* **Description**: Secondary action button, styled using neutral colors or an outline variant based on styles in [B3.8 v2]. Labelled "Réinitialiser". Ensures clear visual distinction from the primary save button
* **Behavior**: Clears or reverts relevant data. Includes standard button interactions ([B3.11])
* **Location**: Typically adjacent to a Save button

### 3.8 UI Style Guide v2 - Zen/Tranquility Theme

*Note: This document replaces the previous UI Style Guide (now archived as v1-Legacy [B3.8]).*

#### 1. Color Palette

Based on the Japandi-inspired color palette defined in [B3.9.2]

##### 1.1 Primary Colors

* **Primary Sage**: #919A7F - Sage green for primary actions, active elements
* **Secondary Taupe**: #A58D7F - For secondary elements, subtle accents

##### 1.3 Neutral Colors

* **Background**: #F3F0E8 - Soft off-white for main background
* **Surface**: #FFFFFF - Pure white for cards, modals, inputs
* **Text Primary**: #2F2F2F - Deep charcoal for main text
* **Text Secondary**: #88837A - Medium gray for secondary information, labels

##### 1.4 Accent/Semantic Colors

* **Positive**: #568E8D - Muted teal for positive balances, successful actions
* **Negative**: #C17C74 - Soft terra cotta for negative balances, warnings
* **Information**: #7A8D99 - Slate blue-gray for informational elements
* **Highlight**: #D9D0C7 - Soft neutral highlight for subtle emphasis

##### 1.5 Gradient Usage

* **Gradients**: Use extremely subtle gradients for section backgrounds or emphasis
* **Example**: Soft blue gradient inspired by Calm app's breathing exercise [B3.9.2]
* **Opacity**: Keep gradients subtle with low opacity (5-20%)
* **Limited usage**: Primarily for key action buttons or section backgrounds

#### 2. Typography

Based on the typography recommendations in [B3.9.3]

##### 2.1 Font Family

**Primary Option 1**: 'Work Sans' + 'IBM Plex Serif'

* Headings: 'Work Sans' (300, 400, 500)
* Body: 'IBM Plex Serif' (400)

**Primary Option 2**: 'Public Sans' + 'Bitter'

* Headings: 'Public Sans' (300, 500)
* Body: 'Public Sans' (400)

**Primary Option 3**: 'Nunito' + 'Nunito Sans'

* Headings: 'Nunito' (300, 600)
* Body: 'Nunito Sans' (400)

**Fallback Font Stack**: system-ui, -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, 'Helvetica Neue', Arial, sans-serif

##### 2.2 Font Size Scale

* **Base Size**: 16px (1rem)
* **Scale Ratio**: 1.618 (golden ratio) as recommended in [B3.9.3]
* **Size Variants**:
  + xs: 0.75rem (12px) - Chart labels, smaller text
  + sm: 0.875rem (14px) - Form labels, secondary text
  + base: 1rem (16px) - Body text, input/select text
  + lg: 1.125rem (18px) - Important text, list items
  + xl: 1.25rem (20px) - Section headers
  + 2xl: 1.5rem (24px) - Page titles, app title
  + 3xl: 2rem (32px) - Major headlines
  + 4xl: 2.5rem (40px) - Hero text, special elements

##### 2.3 Font Weights

* **Light**: 300 - For page titles and larger text for elegance
* **Regular**: 400 - For body text
* **Medium**: 500 - For section headers and emphasis
* **Semibold**: 600 - For emphasis/numbers, particularly financial figures
* Avoid heavy weights (700+) as they can create visual tension

##### 2.4 Line Heights

* **Tight**: 1.25 - For headings
* **Base**: 1.5 - For body text
* **Relaxed**: 1.7 - For larger text blocks and improved readability
* **Loose**: 1.8-2.0 - For maximum readability in longer passages

##### 2.5 Text Styles

* **Headings**: Light (300) for elegance or Medium (500) for emphasis, letter-spacing: -0.02em
* **Body Text**: Regular weight, slightly loose letter-spacing: 0.01em
* **Financial Figures**: Use tabular numbers for alignment when displaying currency
* **Emphasis**: Use weight (Semibold 600) rather than italics for emphasizing financial data

#### 3. Spacing System

##### 3.1 Base Unit

* **Base**: 4px
* **Scale**: Multiples of 4px
* **Note**: Use more generous spacing than typical applications to create sense of calm

##### 3.2 Spacing Scale

* 0: 0px
* 1: 4px (0.25rem)
* 2: 8px (0.5rem)
* 3: 12px (0.75rem)
* 4: 16px (1rem)
* 5: 20px (1.25rem)
* 6: 24px (1.5rem)
* 8: 32px (2rem)
* 10: 40px (2.5rem)
* 12: 48px (3rem)
* 16: 64px (4rem)
* 20: 80px (5rem)
* 24: 96px (6rem)

##### 3.3 Whitespace Guidelines

* **Generous Spacing**: Use more whitespace than typical applications to create a sense of calm
* **Consistent Rhythm**: Maintain consistent spacing between similar elements
* **Vertical Rhythm**: Use the spacing scale for margins between blocks
* **Content Breathing**: Allow content extra room, particularly financial information

#### 4. Iconography

As specified in [B3.9.5], maintaining consistency with the selected library

##### 4.1 Icon System

* **Primary Icon Set**: Heroicons (<https://heroicons.com/>)
* **Style**: Outline variant for most UI controls; solid variant for selected/active states
* **Size Scale**:
  + Small: 16px (1rem)
  + Medium: 20px (1.25rem)
  + Large: 24px (1.5rem)
  + XLarge: 32px (2rem)

##### 4.2 Icon Treatment

* **Stroke Width**: 1.5px for outline icons
* **Corner Radius**: 2px minimum
* **Color**: Should match text color unless highlighting a specific action
* **Touch Targets**: Minimum 44px x 44px for interactive icons
* **Usage Guidelines**:
  + Limit icon use to where they genuinely aid understanding
  + Consider subtle animations for state changes (outline to solid)
  + Maintain consistent sizing within context

#### 5. Shadows & Elevation

Follows the shadow treatment guidelines in [B3.9.4]

##### 5.1 Shadow Levels

* **Level 0**: No shadow (flat) - Most elements
* **Level 1**: 0 1px 2px rgba(0,0,0,0.05), 0 1px 1px rgba(0,0,0,0.03) - Subtle elevation
* **Level 2**: 0 2px 4px rgba(0,0,0,0.06), 0 1px 2px rgba(0,0,0,0.04) - Cards, dropdowns
* **Level 3**: 0 4px 8px rgba(0,0,0,0.07), 0 2px 4px rgba(0,0,0,0.05) - Floating elements
* **Level 4**: 0 8px 16px rgba(0,0,0,0.08), 0 4px 8px rgba(0,0,0,0.04) - Modals, popovers

##### 5.2 Shadow Usage

* Use extremely subtle shadows (2-4px blur, 10-20% opacity)
* Limit to 1-2 levels of elevation in the interface
* Consider no shadows for most elements, reserving them only for modals/overlays
* Consider using very subtle background color changes as an alternative to shadows

#### 6. Border Radius

Aligned with [B3.9.4] Border & Corners recommendations

##### 6.1 Radius Scale

* **Small**: 4-6px - For buttons and interactive elements
* **Medium**: 8-12px - For containers and cards
* **Large**: 12-16px - For modals, larger containers
* **Full**: 9999px - For pills, tags, and circular elements

##### 6.2 Usage Guidelines

* Prefer moderate rounding (8-12px radius) for containers and cards
* Use slightly tighter rounding (4-6px) for buttons and interactive elements
* Be consistent with radius choice across similar components
* Buttons should have consistent corner radius throughout the app

#### 7. UI Component Styles

##### 7.1 Buttons

**Primary Button**

* Background: Primary Sage (#919A7F) or subtle Primary Gradient
* Text: White or very light gray (#F8F9FA)
* Padding: 12px 24px (3 units × 6 units)
* Border Radius: 6px
* Font Weight: Medium (500) or Semibold (600)
* Transition: 0.2s ease-out all

**Secondary Button**

* Background: Transparent
* Border: 1.5px solid Primary Sage (#919A7F)
* Text: Primary Sage (#919A7F)
* Padding: 12px 24px (3 units × 6 units)
* Border Radius: 6px
* Font Weight: Medium (500)
* Transition: 0.2s ease-out all

**Tertiary/Text Button**

* Background: Transparent
* Text: Primary Sage (#919A7F) or Text Primary (#2F2F2F)
* Padding: 12px 16px (3 units × 4 units)
* Border Radius: 4px
* Font Weight: Regular (400) or Medium (500)
* Transition: 0.15s ease-out all

**Disabled State**

* Opacity: 0.6
* Cursor: not-allowed

##### 7.2 Form Elements

**Input Fields**

* Background: Surface White (#FFFFFF)
* Border: 1px solid Text Secondary (#88837A) at 40% opacity
* Border Radius: 6px
* Padding: 12px 16px (3 units × 4 units)
* Focus State: 1.5px border of Primary Sage (#919A7F), subtle shadow (Level 1)
* Error State: 1.5px border of Negative accent (Soft Terra Cotta #C17C74)

**Checkboxes & Radio Buttons**

* Size: 18px × 18px
* Border: 1.5px solid Text Secondary (#88837A)
* Border Radius: 4px for checkboxes, full (circular) for radio buttons
* Checked State: Primary Sage (#919A7F) background, white checkmark
* Focus State: Light sage outline (3px) outside border

**Dropdown**

* Similar to Input Fields, with right-aligned dropdown icon (Heroicon)
* Dropdown Items: 12px padding, hover state with Background color (#F3F0E8)

##### 7.3 Cards & Containers

**Standard Card**

* Background: White or Soft White
* Border Radius: 8px
* Padding: 24px (6 units)
* Shadow: Level 1 or 2
* Border: Optional 1px Light Gray

**Information Container**

* Background: Light variant of Primary Sage (#919A7F) at 10% opacity
* Border Radius: 8px
* Padding: 16px 20px (4 units × 5 units)
* Border: Optional 1px of Primary Sage (#919A7F) at 20% opacity

**Transaction Container**

* Background: White
* Border-left: 3px solid semantic color (based on transaction type)
* Border Radius: 6px
* Padding: 16px (4 units)
* Hover: Very subtle background change (#F8F9FA to #F1F3F5)

##### 7.4 Navigation

**Top Navigation**

* Background: Surface White (#FFFFFF)
* Shadow: Level 1
* Height: 64px (16 units)
* Text: Text Primary (#2F2F2F), Primary Sage (#919A7F) for active
* Item Spacing: 32px (8 units) between items

**Side Navigation**

* Background: Background (#F3F0E8)
* Width: 240-280px
* Item Padding: 12px 16px (3 units × 4 units)
* Active Item: Light background (10% Primary Sage), left border of Primary Sage (#919A7F, 3px)

##### 7.5 Modals & Dialogs

**Standard Modal**

* Background: White
* Border Radius: 12px
* Padding: 24px (6 units)
* Shadow: Level 4
* Header: Bottom border of Light Gray, 24px bottom margin
* Footer: Top border of Light Gray, 24px top margin

**Alert Dialog**

* Similar to Standard Modal, with color-coded borders or icons based on alert type

#### 8. States & Variations

##### 8.1 Interactive States

**Hover State**

* Subtle background lightening (5-10%)
* For buttons: slight brightness increase or 10% darkening
* Transition: 0.15s ease-out

**Active/Pressed State**

* Slightly darker than hover (additional 5%)
* Subtle "push" effect (1px downward shift or box-shadow adjustment)
* Transition: 0.05s ease-out

**Focus State**

* Outline: Primary Sage (#919A7F) outline (2-3px) with 2px offset
* Transition: 0.1s ease-out
* Note: Should be visible enough for accessibility while maintaining the Zen aesthetic

##### 8.2 Component Variations

**Success Variation**

* Use Success color
* Apply to buttons, alerts, indicators that signify completion

**Warning Variation**

* Use Warning color
* Apply to cautionary alerts, pending status indicators

**Danger Variation**

* Use Danger color
* Apply to destructive actions, error states, critical alerts

##### 8.3 Disabled States

* Opacity: 0.6
* Remove hover interactions
* Cursor: not-allowed

#### 9. Responsive Adaptations

##### 9.1 Breakpoints

* Small: 640px and up
* Medium: 768px and up
* Large: 1024px and up
* X-Large: 1280px and up

##### 9.2 Component Adjustments

* Cards: Reduce padding to 16px on small screens
* Typography: Reduce heading sizes by one step on small screens
* Buttons: Consider full-width buttons on very small screens
* Navigation: Convert to bottom tabs or hamburger menu on small screens

#### 10. Implementation Notes

##### 10.1 CSS Variables

Implementation should use CSS Custom Properties (variables) for all color, spacing, and typography values to ensure consistency and enable theming.

##### 10.2 Dark Mode Considerations

While the initial implementation focuses on light mode, all colors should have dark mode equivalents defined that maintain the Zen/Tranquility feeling.

##### 10.3 Legacy Compatibility

This style guide replaces v1-Legacy but components should be designed for graceful degradation when interacting with legacy components during transition.

### 3.9 Mood Board / Inspiration: Zen/Tranquility Theme

This section captures the visual and conceptual inspiration driving the 'Zen/Tranquility' aesthetic for KarmaCash (formerly Budget App). It aims to guide design decisions towards creating a calm, clear, and supportive user experience that reduces financial anxiety and promotes mindfulness.

#### 3.9.1 Core Concepts

The desired theme is defined by the following principles:

* **Calmness & Serenity**: Creating a non-stressful environment for managing finances
* **Clarity & Focus**: Prioritizing information, reducing clutter, ensuring easy comprehension
* **Minimalism**: Using only necessary elements, simplifying forms and navigation
* **Generous Whitespace**: Employing ample spacing to improve readability and reduce visual noise
* **Softness**: Utilizing gentle colors, rounded corners, and subtle interactions
* **Approachability**: Using clear language and subtle visual cues to make finance management less intimidating

#### 3.9.2 Color Palette Inspiration

The goal is a muted, natural, and soothing palette. Earthy tones, soft greens, calming blues, and warm neutrals are preferred.

**Primary Palette: Japandi Theme**

* Based on the Japandi color palette (Image 4)
* Core colors:
  + Primary: Sage Green (#919A7F) - For primary actions, active elements
  + Secondary: Warm Taupe (#A58D7F) - For secondary elements, subtle accents
  + Background: Soft Off-White (#F3F0E8) - Main background
  + Surface: Pure White (#FFFFFF) - Cards, modals, inputs
  + Text Primary: Deep Charcoal (#2F2F2F) - Main text
  + Text Secondary: Medium Gray (#88837A) - Secondary information, labels

**Functional Color Assignments**

* Positive Financial Indicators: Muted Teal (#568E8D) - For positive balances, successful actions
* Negative Financial Indicators: Soft Terra Cotta (#C17C74) - For negative balances, warnings
* Note: These colors are softer than traditional green/red but maintain clear distinction
* Information/Neutral: Slate Blue-Gray (#7A8D99) - For informational elements

**Accessibility Notes**

* All text colors maintain a minimum contrast ratio of 4.5:1 against their backgrounds
* Primary interactive elements use the darker shades for adequate contrast
* Consider using slightly deeper versions of the palette colors for text on light backgrounds

**Gradient Use**

* Subtle gradients can be used for section backgrounds or emphasis
* Example: The Calm app's breathing exercise (Image 5) demonstrates an effective use of a soft blue gradient that could inspire meditation-like states during financial review

#### 3.9.3 Typography Recommendations

**Primary Font Pairings**:

* **Option 1**: Work Sans + IBM Plex Serif
  + Headings: Work Sans (300, 400, 500) - Clean, slightly rounded sans-serif
  + Body: IBM Plex Serif (400) - For longer text passages, adds warmth
* **Option 2**: Public Sans + Bitter
  + Headings: Public Sans (300, 500) - Government-inspired, clear, trustworthy
  + Body: Public Sans (400) - Maintains clarity with slightly loose letter-spacing
* **Option 3**: Nunito + Nunito Sans
  + Headings: Nunito (300, 600) - Friendly, rounded terminals
  + Body: Nunito Sans (400) - More neutral but related feel

**Font Weight Usage**:

* Page Titles: Light (300) for elegance or Medium (500) for emphasis
* Section Headers: Medium (500)
* Regular Text: Regular (400)
* Emphasis/Numbers: Semi-bold (600) - Particularly for financial figures
* Avoid heavy weights (700+) as they can create visual tension

**Size Hierarchy**:

* Follow a harmonious scale based on the golden ratio (1:1.618)
* Base size: 16px (body text)
* Sizes should create clear hierarchy without extreme contrasts

#### 3.9.4 UI Element Treatment

**Borders & Corners**:

* Prefer moderate rounding (8-12px radius) for containers and cards
* Use slightly tighter rounding (4-6px) for buttons and interactive elements
* Buttons should have consistent corner radius throughout the app
* Minimize use of borders - separate content with whitespace instead

**Shadows & Depth**:

* Use extremely subtle shadows (2-4px blur, 10-20% opacity)
* Limit to 1-2 levels of elevation in the interface
* Consider no shadows for most elements, reserving them only for modals/overlays
* Example: The Betterment app (Image 2) demonstrates effective card separation with minimal shadows

**Container Treatment**:

* Card backgrounds: Pure white (#FFFFFF)
* Page backgrounds: Soft off-white from the palette
* Consider very subtle texture or grain for background surfaces to add warmth
* Differentiate sections with subtle color shifts rather than hard dividers

#### 3.9.5 Iconography

**Style**: Minimalist, clean line icons that are easily recognizable and contribute to the uncluttered aesthetic. Consistency is key.

**Selected Library**: Heroicons (<https://heroicons.com/>) is the preferred library due to its clean design, comprehensive set, and compatibility with modern web development (React/SVG).

**Usage Guidelines**:

* Prefer outline style for most UI contexts
* Use solid style for active states or emphasis
* Maintain consistent sizing within context (navigation, buttons, etc.)
* Limit icon use to where they genuinely aid understanding
* Consider subtle animations for state changes (e.g., outline to solid on activation)

#### 3.9.6 Animation & Interaction

**Timing & Easing**:

* Use slower transitions (300-500ms) for major state changes
* Use slightly faster transitions (150-250ms) for hover/focus states
* Prefer ease-out or ease-in-out easing functions for natural movement
* All animations should feel gentle, never jarring or surprising

**Key Interactions to Animate**:

* Tab/view transitions: Subtle fade and slide
* Button state changes: Gentle color/shadow transitions
* Data updates: Smooth number transitions when values change
* Form feedback: Soft reveal of validation messages

**Breathing/Pulsing Elements**:

* Consider subtle "breathing" animations (similar to Calm's breathing circle in Image 5) for loading states or as a visual metaphor for financial "health"
* Potential for circular progress indicators that evoke the enso symbol

**Micro-interaction Guidelines**:

* Animations should communicate meaning, not just decorate
* Use subtle scaling (95-105%) for press/touch feedback
* Consider gentle fades for appearing/disappearing elements
* Avoid animations that block user interaction or create delay

#### 3.9.7 Layout & Interaction Inspiration

**App Examples**:

*Headspace App (Image 1)*:

* Demonstrates effective use of illustration, friendly typography, and muted color blocking
* The light slate blue background creates a calm atmosphere without being clinical
* Character illustrations add personality without cluttering the interface
* Clear content hierarchy with ample whitespace

*Betterment App (Image 2)*:

* Shows clean list presentation with excellent typography hierarchy
* Financial information is presented clearly without creating anxiety
* Subtle iconography complements rather than dominates
* Card metaphor used lightly without strong borders
* Great example of density control - sufficient information without overwhelming

*Wealthfront Onboarding (Image 3)*:

* Exemplifies extreme minimalism and focus on a single task
* Generous whitespace frames the content
* Simple visualization with muted colors
* Clear differentiation between interactive and informational elements
* Typography creates hierarchy without requiring multiple weights

*Calm App (Image 5)*:

* Demonstrates effective use of gradients to create atmosphere
* Circular central element creates focus (potential parallel with enso logo)
* Minimalist design still feels warm and inviting
* Text is limited to only what's essential

**Layout Principles**:

* Maximize whitespace around elements and sections
* Use clear visual hierarchy (size, weight, color) to guide the eye
* Prefer simpler navigation patterns
* Employ card-based layouts or distinct visual grouping with spacing rather than heavy lines/borders
* Consider circular motifs that reinforce the enso brand element

#### 3.9.8 Visual Reference Links

* Heroicons: <https://heroicons.com/>
* Coolors Palette Generator: <https://coolors.co/>
* Google Fonts: <https://fonts.google.com/>
* Japandi Design Principles: <https://www.pinterest.com/search/pins/?q=japandi%20design>

#### 3.9.9 Application to KarmaCash

The Zen/Tranquility theme aligns perfectly with the KarmaCash name and enso logo, creating a cohesive brand identity:

* The circular enso symbol represents completeness and harmony - ideal concepts for financial wellbeing
* "Karma" suggests the positive outcomes of mindful financial management
* The calming interface reduces the anxiety often associated with financial apps
* Clean information hierarchy ensures users can understand their finances without stress
* Soft visual design creates a welcoming environment for regular financial check-ins

In implementation, consider a lightweight prototype to validate this aesthetic direction, particularly focusing on how users emotionally respond to the calming interface when dealing with potentially stressful financial information.

### 3.10 Design Rationale: Zen/Tranquility Theme

#### 3.10.1 Theme Selection

The decision to adopt a "Zen/Tranquility" visual theme for KarmaCash (formerly Budget App) stems from a deliberate effort to transform the typically stressful experience of financial management into one of mindfulness, clarity, and positive control. This approach represents a significant departure from conventional financial application design, which often emphasizes data density, corporate aesthetics, and utilitarian interfaces.

#### 3.10.2 Core Motivations & Benefits

**Reducing Financial Anxiety**: Financial management is consistently ranked among the top sources of personal stress. Studies show that 72% of Americans report feeling stressed about money at least some of the time (American Psychological Association, 2022). A calming visual environment—achieved through muted natural colors, generous whitespace, and minimalist design ([B3.9])—can measurably reduce cognitive load and emotional tension. The app thus becomes a supportive sanctuary rather than another source of pressure.

**Promoting Clarity and Focus**: The human brain processes visual information 60,000 times faster than text (3M Corporation research). By minimizing visual clutter and establishing clear information hierarchy ([B3.4]), users can more quickly comprehend their financial status and make informed decisions. This clarity is particularly crucial when users are making consequential financial choices or feeling overwhelmed.

**Supporting Mindful Financial Decisions**: The Zen aesthetic encourages users to slow down and be present—principles directly applicable to healthier financial decision-making. Research in behavioral economics suggests that impulsive financial decisions often lead to regret, while deliberate choices align better with long-term goals. Our design supports this "slow finance" approach.

**Brand Alignment with KarmaCash Identity**: The name "KarmaCash" itself evokes the Eastern philosophical concept that actions have consequences—perfectly aligned with financial planning where today's decisions affect tomorrow's outcomes. The enso circle logo (an incomplete circle drawn in a single brushstroke) symbolizes both perfection and imperfection, mirroring the reality of personal finance as an ongoing journey rather than a fixed destination.

#### 3.10.3 Psychological Foundations

**Color Psychology**: The selected palette ([B3.9.2]) leverages established research on color effects:

* Soft greens promote restfulness and reduce anxiety (University of British Columbia studies)
* Earth tones create feelings of stability and groundedness
* Muted blues improve focus and productivity
* The deliberate avoidance of harsh reds (except in subtle, softer forms for warnings) prevents triggering stress responses

**Spatial Psychology**: The generous use of whitespace isn't merely aesthetic—studies show that visual crowding increases cortisol (stress hormone) levels. The app's breathing room allows users to process financial information without feeling overwhelmed.

**Cognitive Load Theory**: By limiting the visual elements and creating clear visual hierarchy, we reduce extraneous cognitive load, allowing users to dedicate more mental resources to the intrinsic cognitive load of financial decision-making.

#### 3.10.4 User Context Applications

The Zen/Tranquility theme provides particular benefit in high-stress financial scenarios:

* **Monthly Budget Planning**: When users are allocating limited resources across competing needs, the calming interface reduces decision fatigue.
* **Negative Balance Situations**: When displaying concerning financial information (overspending, negative balances), the soothing design elements help maintain perspective and encourage solution-focused thinking rather than panic.
* **Financial Goal Setting**: The clean, focused UI supports thoughtful reflection on priorities when establishing savings goals or spending limits.
* **Daily Transaction Tracking**: The non-judgmental, clean aesthetic makes regular financial check-ins feel like a mindfulness practice rather than a chore.

#### 3.10.5 Competitive Differentiation

Most financial applications fall into two design categories:

* **Corporate/Banking Aesthetic**: Characterized by aggressive blue color schemes, dense information display, and emphasis on numbers over human experience.
* **Gamified/Trendy Approach**: Using bright colors, animations, and reward systems that can sometimes trivialize financial management.

KarmaCash's Zen/Tranquility theme establishes a third path—one that respects the seriousness of financial management while acknowledging the human emotional experience. This approach differentiates the app in an increasingly crowded marketplace and appeals to users seeking a more mindful relationship with their finances.

#### 3.10.6 Sensory Design Principles

The theme extends beyond visual elements to create a holistic sensory experience:

* **Visual Calm**: Achieved through color palette, typography, and spatial design.
* **Interactive Gentleness**: Subtle animations, soft transitions, and responsive feedback create a tactile sense of cushioned interaction.
* **Temporal Rhythm**: Deliberately slower animations for major transitions encourage momentary pauses for reflection.
* **Emotional Resonance**: Design elements that evoke positive emotional states—confidence, security, and accomplishment—through appropriate use of color and visual metaphor.

#### 3.10.7 Implementation Philosophy

The Zen/Tranquility theme isn't merely a visual overlay—it should inform every design decision from information architecture to interaction patterns. Key implementation principles include:

* **Essential Simplicity**: Continuously question whether each element serves a vital purpose.
* **Progressive Disclosure**: Reveal complexity only when needed, maintaining the calm surface experience.
* **Consistency of Feeling**: Ensure all components—from major screens to minor dialogs—maintain the tranquil quality.
* **Balancing Serenity with Action**: While promoting calm, the design must still effectively guide users to take necessary financial actions.

The theme should be validated through user testing, particularly observing physiological and self-reported stress measures when users interact with potentially anxiety-inducing financial scenarios in the app.

### 3.11 Micro-interactions & Animation Principles

This document builds upon the animation & interaction guidelines established in [B3.9.6]

#### 1. Core Animation Principles

##### 1.1 Purpose & Philosophy

Animations in KarmaCash should:

* Enhance, not distract - Support the user's tasks without calling attention to themselves
* Provide meaning - Clarify relationships between elements and guide attention
* Create calm - Reinforce the Zen/Tranquility theme with gentle, smooth movements
* Reduce cognitive load - Help users understand what's happening through visual feedback

##### 1.2 General Guidelines

* Subtlety is key - All animations should be understated and refined
* Consistency matters - Similar actions should have similar animations
* Performance first - Animations should never cause lag or janky scrolling
* Accessibility aware - All animations must respect user preferences (prefers-reduced-motion)

#### 2. Timing & Easing Functions

##### 2.1 Duration Standards

Following the timing recommendations in [B3.9.6]

* Ultra-Fast: 100ms - For immediate feedback (button clicks, toggle switches)
* Fast: 150-250ms - For hover/focus states, aligned with [B3.9.6]
* Medium: 250-350ms - For standard transitions (panel slides, modals)
* Slow: 300-500ms - For major state changes, as recommended in [B3.9.6]
* Very Slow: 600-800ms - Reserved for special moments (celebrations, onboarding)
* Note: Durations over 1000ms should almost never be used as they will feel sluggish

##### 2.2 Easing Functions

* Standard Ease: cubic-bezier(0.4, 0.0, 0.2, 1) - For most transitions
* Ease Out: cubic-bezier(0.0, 0.0, 0.2, 1) - For elements entering the screen
* Ease In: cubic-bezier(0.4, 0.0, 1, 1) - For elements exiting the screen
* Sharp: cubic-bezier(0.4, 0.0, 0.6, 1) - For emphasis or attention

##### 2.3 Animation Sequence Principles

* Staggered entries: When multiple items appear, stagger them by 20-40ms
* Hierarchical timing: More important elements should animate slightly before less important ones
* Logical flow: Animation should follow the natural reading direction (typically left-to-right, top-to-bottom)

#### 3. Specific Micro-interactions

##### 3.1 Button Interactions

Incorporating the subtle scaling (95-105%) mentioned in [B3.9.6]

* Hover: Subtle background lightening (150-250ms ease-out)
* Press/Active: Slight scale down to 95-98% size, slightly darker (100ms ease-out)
* Focus: Gentle pulse of outline (single pulse, 300ms)
* Loading: Subtle opacity pulse or minimal spinner (consistent speed regardless of operation length)
* Success Completion: Gentle fade from loading to checkmark icon (250ms)

##### 3.2 Form Element Interactions

* Field Focus: Subtle border animation emphasizing the highlighted state (200ms)
* Validation Success: Gentle fade-in of success icon (250ms ease-out)
* Validation Error: Soft appearance of error message with 2px slide from top (250ms ease-out)
* Checkbox/Toggle: Smooth transition between states (150ms), with slight bounce for toggle switches

##### 3.3 Navigation & Transitions

* Page Transitions: Subtle fade combined with slight movement (300-400ms)
* Tab Switching: Content crossfade with active indicator sliding (250ms)
* Menu Opening: Gentle slide-in with fade (250ms ease-out)
* Menu Closing: Quicker fade-out (200ms ease-in)

##### 3.4 Data & Feedback Interactions

* Data Updates: Values that change should transition smoothly (300ms), not jump instantly
* Charts & Graphs: Progressive reveal of data points (staggered, total animation under 800ms)
* Progress Indicators: Smooth fills for progress bars (vary speed based on operation)
* Notifications: Gentle slide in from top or side, with subtle bounce at end (enter: 300ms, exit: 200ms)

##### 3.5 Financial-Specific Interactions

* Balance Changes: Numbers should count up/down for significant changes (400ms)
* Transaction Addition: New transactions slide in and settle (300ms ease-out)
* Goal Achievement: Subtle celebration animation - gentle pulse of color or icon (600ms)
* Budget Status Changes: Color transitions should be smooth when categories change status (300ms)

#### 4. When NOT to Animate

##### 4.1 Inappropriate Animation Contexts

* Critical Error Messages: Should appear immediately without delay
* Frequent Updates: Elements that update very frequently (e.g., real-time data)
* Dense Information Displays: Animation could make it harder to scan and compare
* After User Preference Indication: Respect "prefers-reduced-motion" settings
* Financial Decision Points: Avoid animations that might distract from critical financial decisions

##### 4.2 Performance Considerations

* Heavy Rendering Contexts: Avoid animations that run alongside complex calculations
* Multiple Simultaneous Animations: Limit concurrent animations to prevent performance issues
* Mobile/Low-Power Contexts: Be more conservative with animations on mobile devices

#### 5. Technical Implementation Guidelines

##### 5.1 Preferred Technologies

* CSS Transitions: For simple state changes
* CSS Animations: For repeating or complex single-element animations
* React Spring/Framer Motion: For more complex physics-based interactions
* Avoid jQuery animations or setTimeout-based animations for performance reasons

##### 5.2 Performance Optimization

* Transform & Opacity: Prefer these properties over animating position or dimensions
* Will-change: Use sparingly for complex animations
* RequestAnimationFrame: For custom JavaScript animations
* Avoid Animating: layout-triggering properties (width, height, top, left)

##### 5.3 Accessibility Implementation

// Example implementation of respecting reduced motion preferences

const prefersReducedMotion = window.matchMedia('(prefers-reduced-motion: reduce)').matches;

// CSS approach

:root {

--transition-standard: 200ms cubic-bezier(0.4, 0.0, 0.2, 1);

}

@media (prefers-reduced-motion: reduce) {

:root {

--transition-standard: 0ms;

}

}

// React approach with styled-components example

const fadeIn = css`

@media (prefers-reduced-motion: no-preference) {

transition: opacity 300ms ease-out;

}

`;

#### 6. Example Scenarios & Implementation

##### 6.1 New Transaction Entry

1. Form appears with subtle fade-in (250ms)
2. As user completes fields, gentle validation indicators appear (200ms)
3. On submit, button shows loading state (subtle pulse)
4. On success, brief success animation (checkmark icon, 250ms)
5. Form dismisses with fade-out (200ms)
6. New transaction appears in list with highlight that fades (400ms)

##### 6.2 Budget Category Progress

1. Progress bar fills smoothly when updated (300ms)
2. When approaching threshold (80%), subtle pulse animation draws attention
3. When exceeding budget, color transition is smooth (300ms)
4. Category expands/collapses with smooth height animation (250ms)

##### 6.3 Dashboard Loading Sequence

1. Page structure appears immediately
2. Placeholder content shown with subtle pulse animation
3. Data elements fade in as they become available
4. Charts animate in with progressive reveal (600ms total)
5. Final "ready" state indicated with subtle completed animation

#### 7. Testing & Refinement

##### 7.1 Animation Testing Checklist

* Verify animations work at different screen sizes
* Test on both high and low-performance devices
* Ensure animations respect reduced-motion settings
* Confirm animations work in all supported browsers
* Measure performance impact (no frame drops below 60fps)

##### 7.2 User Experience Validation

* Animations should feel "right" - not too fast or too slow
* Get feedback on whether animations help or hinder task completion
* Verify animations align with the Zen/Tranquility theme's calm feeling
* Check that animations don't cause motion sickness or discomfort

#### 8. Integration with Style Guide

This Micro-interactions & Animation Principles document complements the UI Style Guide [B3.8 v2] and should be referenced alongside it when implementing components. All animations should reinforce the Zen/Tranquility theme established in the Design Rationale [B3.10] and the animation concepts introduced in [B3.9.6].

The principles defined here expand on the breathing/pulsing elements concept mentioned in [B3.9.6], providing concrete timing, easing, and implementation guidelines to ensure consistent application of the Zen/Tranquility theme across all interactive elements.

### 3.12 Impact on UI Components

#### 1. Overview of Impact

This document reviews how the new Zen/Tranquility theme will impact the existing UI Component Specifications [B3.7]. The assessment identifies stylistic changes needed for components while maintaining their functional requirements.

#### 2. General Component Impact

##### 2.1 Visual Changes Across All Components

The following changes should be applied across all components to align with the Zen/Tranquility theme:

* **Increased Whitespace**: All components require more padding and margins as specified in [B3.9.1]
* **Softer Color Palette**: Replace existing high-contrast colors with the new Japandi-inspired palette from [B3.9.2]
* **Rounded Corners**: Increase border radius values (8-12px for containers, 4-6px for interactive elements)
* **Reduced Visual Weight**: Decrease border thickness, use lighter shadows (2-4px blur, 10-20% opacity)
* **Typography Updates**: Apply new font pairings from [B3.9.3] with the suggested weight usage
* **Animation Integration**: Implement the slower, gentler transitions (300-500ms for major changes)
* **Iconography**: Replace existing icons with Heroicons in the outlined style as specified in [B3.9.5]

##### 2.2 Anticipated Effort Levels

* **Low Impact**: Components requiring minimal changes (color, spacing)
* **Medium Impact**: Components needing structural adjustments (layout, hierarchy)
* **High Impact**: Components requiring significant redesign or new concepts

#### 3. Specific Component Assessments

##### 3.1 Navigation Components

###### 3.1.1 Main Navigation Bar

* **Impact Level**: Medium
* **Key Changes**:
  + Replace blue underline for active tab with softer Primary Sage Green (#919A7F)
  + Increase vertical padding from standard to 16-20px
  + Replace text styling with new typography standards
  + Implement subtle hover/active states (150-250ms transitions)
* **Visual Recommendations**:
  + Consider a slightly thicker underline (2px) with rounded ends for active tab

###### 3.1.2 Date Navigation Controls

* **Impact Level**: Medium
* **Key Changes**:
  + Update arrow icons to Heroicons
  + Increase spacing between controls
  + Replace blue highlight for selected view mode with Sage Green
  + Apply more generous padding to clickable areas
* **Visual Recommendations**:
  + Consider subtle background for the current period display

###### 3.1.3 Period Toggle

* **Impact Level**: Low
* **Key Changes**:
  + Update styling to match new color palette
  + Apply more rounded corners to toggle switch (full radius)
  + Increase touch target size with added padding
* **Visual Recommendations**:
  + Consider a more subtle visual treatment for the toggle

##### 3.2 Header Components

###### 3.2.1 App Header

* **Impact Level**: Medium
* **Key Changes**:
  + Replace strong blue (#0D6EFD) with muted Sage Green (#919A7F) or a subtle gradient
  + Update typography to match new style guide
  + Replace success green for balance with muted Teal (#568E8D)
* **Visual Recommendations**:
  + Consider reducing visual weight with a lighter background or subtle gradient
  + Explore adding subtle texture or grain to the header background

##### 3.3 Transaction Components

###### 3.3.1 Transaction Day Card

* **Impact Level**: Medium
* **Key Changes**:
  + Replace blue card header with Sage Green or other palette color
  + Increase card padding from 1rem to 1.5rem (24px)
  + Apply more generous border radius (8-12px)
  + Reduce shadow to very subtle elevation
* **Visual Recommendations**:
  + Consider using color-coded left borders instead of full colored headers

###### 3.3.2 Transaction Item

* **Impact Level**: Medium
* **Key Changes**:
  + Increase spacing between items
  + Replace red/green indicators with muted Terra Cotta (#C17C74) and Teal (#568E8D)
  + Update recurring indicator to match new palette
  + Implement subtle hover state with very light background change
* **Visual Recommendations**:
  + Consider using subtle iconography to indicate transaction types

###### 3.3.3 Transaction Form

* **Impact Level**: High
* **Key Changes**:
  + Increase field spacing significantly (from 1rem to 1.5rem)
  + Apply new form styling with softer focus states
  + Update button styling to match Zen aesthetic
  + Implement gentle micro-interactions for form feedback
* **Visual Recommendations**:
  + Consider grouping related fields with subtle background colors
  + Explore using floating labels or alternative input styling

###### 3.3.4 Calendar View

* **Impact Level**: High
* **Key Changes**:
  + Apply more generous spacing to calendar grid
  + Reduce visual weight of grid lines or eliminate them
  + Update day indicators to use new palette colors
  + Implement subtle hover effects for interactive days
* **Visual Recommendations**:
  + Consider a more modern, airy calendar design with ample whitespace

##### 3.4 Budget Components

###### 3.4.1 Budget Allocation Bar

* **Impact Level**: Medium
* **Key Changes**:
  + Increase height and border radius of progress bars
  + Update colors to match new semantic palette
  + Add subtle transitions when values change
  + Implement softer visual treatment for over-budget indicators
* **Visual Recommendations**:
  + Consider adding very subtle patterns or transparency effects

###### 3.4.2 Remaining to Allocate Indicator

* **Impact Level**: Medium
* **Key Changes**:
  + Update styling to use new color palette
  + Soften the visual treatment for negative values
  + Add subtle animation for value changes
* **Visual Recommendations**:
  + Consider making this component more prominent with card-like treatment

###### 3.4.3 Monthly Savings Display

* **Impact Level**: Medium
* **Key Changes**:
  + Apply new progress bar styling
  + Update typography for clearer hierarchy
  + Implement gentle animations for value changes
* **Visual Recommendations**:
  + Consider visual metaphors that reinforce the "growth" concept

###### 3.4.4 Available Funds Display

* **Impact Level**: Medium
* **Key Changes**:
  + Update styling to use new color palette
  + Increase padding and whitespace
  + Implement subtle highlighting effect
* **Visual Recommendations**:
  + Consider making this a central focal point with card elevation

##### 3.5 Visualization Components

###### 3.5.1 Expense Distribution Pie Chart

* **Impact Level**: High
* **Key Changes**:
  + Update all chart colors to new palette
  + Implement softer, more harmonious color scheme for segments
  + Add subtle animations for segment hover
  + Increase spacing around chart elements
* **Visual Recommendations**:
  + Consider using semi-transparency or subtle patterns in segments
  + Explore alternative chart types that feel more zen-like

###### 3.5.2 Balance Evolution Bar Chart

* **Impact Level**: High
* **Key Changes**:
  + Update bar colors to match new palette
  + Reduce visual weight of axes and grid lines
  + Implement progressive animations
  + Apply new typography to labels
* **Visual Recommendations**:
  + Consider using rounded bar caps
  + Explore subtle gradient fills for bars

##### 3.6 Category Components

###### 3.6.1 Category List Item

* **Impact Level**: Medium
* **Key Changes**:
  + Increase row padding and spacing
  + Update color indicators to new palette
  + Implement subtle hover/expand animations
* **Visual Recommendations**:
  + Consider using card-like treatment for expandable sections

###### 3.6.2 Category Selector

* **Impact Level**: Medium
* **Key Changes**:
  + Update dropdown styling to match form elements
  + Implement new color coding system
  + Add subtle transitions for open/close
* **Visual Recommendations**:
  + Consider a more visual category selection interface with color swatches

##### 3.7 Action Components

###### 3.7.1 Delete Button

* **Impact Level**: Medium
* **Key Changes**:
  + Replace standard red with muted Terra Cotta (#C17C74)
  + Increase padding and border radius
  + Implement gentler hover/active states
* **Visual Recommendations**:
  + Consider a more subtle approach to destructive actions

###### 3.7.2 Delete Series Button

* **Impact Level**: Medium
* **Key Changes**:
  + Similar updates as Delete Button
  + Add clearer differentiation from single delete
* **Visual Recommendations**:
  + Consider using iconography to reinforce the "series" concept

###### 3.7.3 Save Button

* **Impact Level**: Medium
* **Key Changes**:
  + Replace blue with Sage Green primary color
  + Update hover/active states to be more subtle
  + Implement gentle success animation
* **Visual Recommendations**:
  + Consider subtle gradient or texture for primary actions

###### 3.7.4 Reset Button

* **Impact Level**: Low
* **Key Changes**:
  + Update to new neutral color from palette
  + Ensure sufficient contrast with Save button
* **Visual Recommendations**:
  + Consider using outline style for secondary actions

#### 4. Implementation Recommendations

##### 4.1 Phased Approach

We recommend a phased implementation:

1. **Foundation Phase**: Update design tokens (colors, typography, spacing)
2. **Core Components Phase**: Update frequently used components (navigation, cards, buttons)
3. **Feature Components Phase**: Update specialized components (charts, forms, tables)
4. **Polish Phase**: Implement micro-interactions and animations

##### 4.2 Prototype Recommendation

Before full implementation, we recommend creating a lightweight prototype focusing on:

* **Color System**: Validate the new palette works across all use cases
* **Typography System**: Ensure readability and hierarchy
* **Key Components**: Test most frequently used components with new styling
* **Responsive Behavior**: Validate the design across breakpoints

##### 4.3 Design-to-Code Considerations

* Implement a design token system for colors, spacing, and typography
* Consider a component library approach for consistent implementation
* Document animation standards for developer reference
* Create a comprehensive demo page showing all component states

## SECTION 4: TECHNOLOGY STACK

### 4.1 Frontend

* **React** (Using functional components with Hooks)
  + Modern, component-based UI library
  + Efficient rendering through virtual DOM
  + Strong ecosystem and community support
  + Excellent for building interactive single-page applications

### 4.2 Backend/API

* **Firebase Functions** (Node.js)
  + Primarily for scheduled tasks like recurring transaction generation
  + Serverless architecture eliminates need for traditional backend management
  + Direct Firestore access from frontend for most CRUD operations
  + Event-triggered functions for specific scenarios as needed

### 4.3 Database

* **Firebase Firestore**
  + NoSQL, document-oriented database
  + Real-time data synchronization capabilities
  + Offline support for improved user experience
  + Scalable with Firebase's pay-as-you-go model
  + Built-in security rules to protect user data

### 4.4 Authentication

* **Firebase Authentication**
  + Email/password authentication initially
  + Session persistence for seamless user experience
  + Potential to add additional auth providers later (Google, Facebook, etc.)
  + Secure token-based authentication integrated with Firestore security rules

### 4.5 Deployment

* **Firebase Hosting**
  + Fast, secure hosting for web applications
  + Global CDN for improved performance
  + Easy deployment through Firebase CLI
  + SSL certificates provided automatically
  + Seamless integration with other Firebase services

### 4.6 Version Control

* **Git & GitHub**
  + Track all code changes
  + Support for branching and collaboration
  + Potential for CI/CD integration in the future
  + Documentation storage (including this strategy document)

### 4.7 Development Tools

* **IDE**: VS Code with Cursor AI extension
* **AI Assistance**: Claude (Strategy, Planning, Prompts), Cursor AI (Code Generation/Modification)
* **Package Manager**: npm (for dependency management)
* **Build Tool**: Vite (Chosen for fast development server and modern tooling)
* **Browser DevTools** (for debugging and performance monitoring)
* **Linting/Formatting**: ESLint and Prettier (for code quality and consistency)

## SECTION 5: HIGH-LEVEL ARCHITECTURE & DATA MODEL

### 5.1 Architecture Overview

Based on our chosen technology stack (React, Firebase), the high-level architecture will look like this:

* **Frontend (React SPA)**: A Single Page Application built with React running entirely in the user's browser after initial loading.
  + Handles all UI rendering and user interactions
  + Communicates directly and securely with Firebase services (Authentication and Firestore)
  + Manages its own state (potentially using React Context or a dedicated state management library like Zustand or Redux Toolkit if needed later, but starting simple first)
* **Firebase Authentication**: Handles user sign-up, login, logout, and session management. The frontend interacts directly with Firebase Auth SDK.
* **Firebase Firestore (Database)**: Our primary data store. The React frontend will read and write data directly to Firestore, structured into collections and documents. Access will be controlled by Firestore Security Rules.
* **Firebase Hosting**: Hosts the static files (HTML, CSS, compiled JavaScript) for our React application.
* **Firebase Cloud Functions (Node.js)**: While the frontend will handle most direct data interaction, Functions might be used for:
  + Scheduled Tasks: E.g., automatically generating upcoming recurring transaction instances based on defined rules.
  + Complex/Sensitive Calculations: E.g., potentially calculating end-of-month budget summaries, rollovers, or total savings updates if these prove too complex or inefficient to do reliably on the client-side across all users (less likely needed for MVP, but good to keep in mind).
  + (Post-MVP) Backend logic for notifications or shared budget invitations/management.
* **Version Control (Git/GitHub)**: Essential for tracking changes, collaboration (even with AI), and potentially automated deployments via GitHub Actions.

### 5.2 Firestore Data Model (MVP Draft)

Here's a proposed structure for our Firestore collections and documents to support the MVP features. We'll use user-specific subcollections extensively to simplify security rules and querying.

#### Users Collection: Stores basic user profile info and settings.

* **Path**: /users/{userId}
* **{userId}**: This will be the UID provided by Firebase Authentication.
* **Document Fields**:
  + email: String (User's email)
  + displayName: String (Optional, user's name)
  + createdAt: Timestamp (When the user account was created)
  + settings: Map (Stores user preferences)
    - currency: String (e.g., "CAD")
    - balanceDisplayMode: String ("cumulative" | "period" - for the header balance toggle)

#### Categories Subcollection: User-defined income/expense categories.

* **Path**: /users/{userId}/categories/{categoryId}
* **{categoryId}**: Auto-generated unique ID by Firestore.
* **Document Fields**:
  + userId: String (The user this category belongs to - good for rules/denormalization)
  + name: String (e.g., "Épicerie", "Salary", "Transport")
  + type: String ("expense" | "income")
  + isDefault: Boolean (Indicates if it was part of the initial set)
  + createdAt: Timestamp

#### Transactions Subcollection: Individual transaction records.

* **Path**: /users/{userId}/transactions/{transactionId}
* **{transactionId}**: Auto-generated unique ID.
* **Document Fields**:
  + userId: String
  + categoryId: String (References a document ID in the user's categories subcollection)
  + date: Timestamp (Crucial for querying/sorting by date)
  + type: String ("expense" | "income")
  + description: String (User-provided details, e.g., "Supermarket C&T", "Monthly Paycheck")
  + amount: Number (Store amounts as numbers for calculations)
  + isRecurringInstance: Boolean (True if this transaction was generated by a rule)
  + recurringRuleId: String (Optional: References the ID of the rule if isRecurringInstance is true)
  + createdAt: Timestamp
  + updatedAt: Timestamp

*Note: Documents where isRecurringInstance is true are generated by a backend process based on recurringRules definitions, typically covering a rolling 1-year future window.*

#### Recurring Rules Subcollection: Definitions for recurring transactions.

* **Path**: /users/{userId}/recurringRules/{ruleId}
* **{ruleId}**: Auto-generated unique ID.
* **Document Fields**:
  + userId: String
  + categoryId: String
  + type: String ("expense" | "income")
  + description: String
  + amount: Number
  + frequency: String ("monthly" | "weekly" | "bi-weekly") // Define supported types
  + interval: Number (e.g., 1 for every month/week, 2 for every other)
  + dayOfMonth: Number (1-31, relevant for 'monthly')
  + dayOfWeek: Number (0-6, Sun-Sat, relevant for 'weekly', 'bi-weekly')
  + startDate: Timestamp (When the recurrence begins)
  + endDate: Timestamp | Null (Optional: When the recurrence ends)
  + isActive: Boolean (Allows users to pause a recurrence)
  + createdAt: Timestamp
  + updatedAt: Timestamp

*Note: Transaction instances based on these rules are intended to be generated approximately 1 year into the future by a scheduled Cloud Function.*

#### Budgets Subcollection: Stores the user's budget plan and summary for each month.

* **Path**: /users/{userId}/budgets/{yyyy-mm}
* **{yyyy-mm}**: A string ID like "2025-04" makes fetching specific months easy.
* **Document Fields**:
  + userId: String
  + month: String ("YYYY-MM" format)
  + year: Number
  + calculated: Map (Stores calculated summary figures for the month)
    - revenue: Number (Total income for the month)
    - recurringExpenses: Number (Total recurring expenses for the month)
    - rolloverFromPrevious: Number (Carry-over, can be negative)
    - availableToAllocate: Number (revenue - recurringExpenses + rolloverFromPrevious)
    - totalAllocated: Number (Sum of allocations below)
    - remainingToAllocate: Number (availableToAllocate - totalAllocated)
    - totalSpent: Number (Total non-recurring expenses + relevant recurring for the month)
    - monthlySavings: Number (Likely revenue - totalSpent - recurringExpenses)
  + allocations: Map (Stores the user's budget allocation targets for the month)
    - Key: categoryId (String)
    - Value: allocatedAmount (Number)
    - Example: { "catId\_epicerie": 500, "catId\_transport": 150 }
  + createdAt: Timestamp
  + updatedAt: Timestamp

#### Relationships & Indexing:

* Data is primarily linked via the userId. Subcollections inherently tie data to a user.
* Transactions and Recurring Rules link to Categories via categoryId.
* Recurring instances in Transactions link back to their rule via recurringRuleId.
* Budget allocations use categoryId as keys.
* Firestore automatically creates basic indexes. As we develop queries (e.g., fetching transactions by date range and category), we might need to define specific composite indexes in Firestore to ensure performance.

### 5.3 Key Calculation Logic (MVP)

**Purpose**: To define the exact formulas and logic for critical financial calculations within the application, ensuring consistency and accuracy, and addressing known issues from the previous implementation. All calculations should operate on numerical data fetched from Firestore.

**Note**: Budget calculations for the MVP (Available Funds, etc.) are based on actual transactions recorded within the month and rollover from the previous month. Future/predicted transactions generated by recurring rules are primarily for user awareness and forecasting display (e.g., in transaction lists/calendar) and do not directly influence the core 'Available Funds' calculation for the current month's budget allocation process.

**Note**: Stored calculated values within the monthly budget document (e.g., calculated.revenue, calculated.totalSpent) will primarily be updated when the user loads the Budget section for that month or saves allocation changes. Real-time comparison values like 'Actual Spending per Category' will be calculated on-the-fly in the frontend by querying the latest transactions.

#### 5.3.1 Monthly Revenue Calculation:

* **Input**: User ID (userId), Month (yyyy-mm).
* **Process**: Query the /users/{userId}/transactions subcollection for all documents where type is "income" and the date falls within the specified month. Sum the amount field of these transactions.
* **Output**: Total Revenue for the month (Number).
* **Storage**: This value will be stored in the corresponding /users/{userId}/budgets/{yyyy-mm} document under calculated.revenue.

#### 5.3.2 Monthly Recurring Expenses Calculation:

* **Input**: User ID (userId), Month (yyyy-mm).
* **Process**: Query the /users/{userId}/transactions subcollection for all documents where type is "expense", isRecurringInstance is true, and the date falls within the specified month. Sum the amount field of these transactions.
* **Output**: Total Recurring Expenses for the month (Number).
* **Storage**: This value will be stored in the corresponding /users/{userId}/budgets/{yyyy-mm} document under calculated.recurringExpenses.

#### 5.3.3 Monthly Budget Rollover Calculation:

* **Input**: User ID (userId), Current Month (current\_yyyy-mm).
* **Process**:
  1. Determine the Previous Month (previous\_yyyy-mm). If the current month is January, the rollover is 0.
  2. Fetch the budget document for the Previous Month from Firestore: /users/{userId}/budgets/{previous\_yyyy-mm}.
  3. If the previous month's budget document exists:
     + Retrieve calculated.availableToAllocate from the previous month's document.
     + Retrieve calculated.totalAllocated from the previous month's document.
     + Rollover Amount = calculated.availableToAllocate - calculated.totalAllocated
  4. If the previous month's budget document does not exist, the Rollover Amount is 0.
* **Output**: Rollover Amount for the Current Month (Number). Crucially, this value can be positive (unspent funds) or negative (overspending).
* **Storage**: This value will be stored in the Current Month's budget document (/users/{userId}/budgets/{current\_yyyy-mm}) under calculated.rolloverFromPrevious.

**Note**: This calculation is performed dynamically whenever the budget data for the current\_yyyy-mm is loaded or recalculated. It uses the latest available data from the previous\_yyyy-mm budget document. Changes made to a previous month's budget or transactions can therefore affect the starting rollover of subsequent months upon recalculation.

#### 5.3.4 Monthly Available Funds Calculation:

* **Input**: User ID (userId), Current Month (yyyy-mm). Requires calculated.revenue, calculated.recurringExpenses, and calculated.rolloverFromPrevious to be calculated first for the current month's budget document.
* **Process**: Fetch the current month's budget document (/users/{userId}/budgets/{yyyy-mm}).
  + Available Funds = calculated.revenue - calculated.recurringExpenses + calculated.rolloverFromPrevious
* **Output**: Available Funds for allocation in the Current Month (Number).
* **Storage**: This value will be stored in the corresponding /users/{userId}/budgets/{yyyy-mm} document under calculated.availableToAllocate.

#### 5.3.5 Monthly Total Allocated Calculation:

* **Input**: User ID (userId), Current Month (yyyy-mm). Requires the allocations map from the current month's budget document.
* **Process**: Fetch the current month's budget document (/users/{userId}/budgets/{yyyy-mm}). Sum all the numerical values within the allocations map.
* **Output**: Total amount allocated across all categories for the month (Number).
* **Storage**: This value will be stored in the corresponding /users/{userId}/budgets/{yyyy-mm} document under calculated.totalAllocated.

#### 5.3.6 Monthly Remaining to Allocate Calculation:

* **Input**: User ID (userId), Current Month (yyyy-mm). Requires calculated.availableToAllocate and calculated.totalAllocated for the current month.
* **Process**: Fetch the current month's budget document (/users/{userId}/budgets/{yyyy-mm}).
  + Remaining to Allocate = calculated.availableToAllocate - calculated.totalAllocated
* **Output**: Amount remaining to be allocated to achieve zero-based budget (Number, can be negative).
* **Storage**: This value will be stored in the corresponding /users/{userId}/budgets/{yyyy-mm} document under calculated.remainingToAllocate. This value should update dynamically in the UI as the user adjusts allocations.

#### 5.3.7 Monthly Actual Spending per Category Calculation:

* **Input**: User ID (userId), Current Month (yyyy-mm), Category ID (categoryId).
* **Process**:
  1. Query /users/{userId}/transactions for the specified categoryId and month.
  2. Initialize totalExpenses = 0, totalIncomeInCategory = 0.
  3. Iterate through transactions:
     + If type is "expense" AND isRecurringInstance is false, add amount to totalExpenses.
     + If type is "income", add amount to totalIncomeInCategory.
  4. Net Spending for Category = totalExpenses - totalIncomeInCategory
* **Output**: Total actual net spending for that specific category in that month (Number). This can be negative if income/refunds exceed expenses in that category for the month.
* **Storage**: This value is typically calculated on-the-fly in the frontend whenever the Budget section or relevant analysis section is displayed, rather than being stored persistently in the budget document, to ensure real-time accuracy as new transactions are added.

**Note**: This ensures that for all categories, income/refunds/reimbursements correctly offset expenses within their assigned category, providing a more accurate view of net spending per category.

#### 5.3.8 Monthly Total Spent Calculation:

* **Input**: User ID (userId), Current Month (yyyy-mm).
* **Process**:
  1. Calculate the sum of actual spending for all budget categories for the month using the logic in 5.3.7 (sum of non-recurring expenses, net of any income/reimbursements across all categories).
  2. Add the calculated.recurringExpenses (from 5.3.2) for the month.
* **Output**: Total spending for the month (recurring + non-recurring discretionary).
* **Storage**: This value could be stored in the corresponding /users/{userId}/budgets/{yyyy-mm} document under calculated.totalSpent, potentially updated periodically or on demand.

#### 5.3.9 Monthly Savings Calculation:

* **Input**: User ID (userId), Current Month (yyyy-mm). Requires calculated.revenue and calculated.totalSpent (or its components) for the month.
* **Process**: Fetch the current month's budget document (/users/{userId}/budgets/{yyyy-mm}).
  + Monthly Savings = calculated.revenue - (calculated.recurringExpenses + Sum of actual non-recurring spending per category [net of any income/reimbursements in all categories])
  + Alternatively, if calculated.totalSpent (from 5.3.8) is reliably stored: Monthly Savings = calculated.revenue - calculated.totalSpent
* **Output**: Net savings for the month (Number, can be negative).
* **Storage**: This value could be stored in the corresponding /users/{userId}/budgets/{yyyy-mm} document under calculated.monthlySavings.

#### 5.3.10 Total Savings Calculation (Cumulative):

* **Input**: User ID (userId), Current Month (current\_yyyy-mm).
* **Process**:
  1. Determine the Previous Month (previous\_yyyy-mm).
  2. Fetch the budget document for the Previous Month (/users/{userId}/budgets/{previous\_yyyy-mm}).
  3. Fetch the Current Month's calculated monthlySavings (from 5.3.9).
  4. If previous month's budget exists, retrieve its calculated.totalSavings. If not, assume previous total savings is 0.
  5. Total Savings = Previous Month's calculated.totalSavings + Current Month's monthlySavings
* **Output**: Accumulated total savings up to the end of the Current Month (Number, should allow negative values).
* **Storage**: This value could be stored in the corresponding /users/{userId}/budgets/{yyyy-mm} document under calculated.totalSavings.

#### 5.3.11 Header Balance Calculation (Period vs. Cumulative):

* **Input**: User ID (userId), Selected Period (startDate, endDate), Balance Display Mode (cumulative | period from user settings).
* **Process**:
  + If Mode is 'period': Query transactions within the startDate and endDate. Calculate Sum(Income Amounts) - Sum(Expense Amounts) only for transactions within that period.
  + If Mode is 'cumulative': Query all transactions for the user up to the endDate of the selected period. Calculate Sum(All Income Amounts) - Sum(All Expense Amounts). This requires fetching potentially more data.
* **Output**: Balance amount to display in the header (Number).
* **Storage**: This is calculated on-the-fly in the frontend based on user interaction and settings.

### 5.4 Date & Time Handling Rules

#### 5.4.1 Storage:

All date/time values must be stored in Firestore using the Timestamp data type (which represents UTC). Do not store dates as strings.

#### 5.4.2 User Input:

Dates selected by the user (e.g., via date picker) represent a specific day in their local timezone. For storage, convert the start of that day (00:00:00) in the user's local timezone to a UTC Timestamp.

#### 5.4.3 Queries:

Date range queries (e.g., for a month or week) must use UTC Timestamp objects representing the start of the first day and the end (23:59:59.999) of the last day of the period in the user's local timezone, converted to UTC.

#### 5.4.4 Display:

Always convert fetched Firestore Timestamp values back to the user's local timezone before displaying them. Use a consistent date formatting library (Recommendation: date-fns) with the fr-CA locale for all user-facing dates and times.

#### 5.4.5 Library:

Use date-fns for all date manipulation (parsing, formatting, comparisons, adding/subtracting time) in the React frontend to ensure consistency and avoid native Date object pitfalls.

#### 5.4.6 Context Emphasis:

AI prompts related to date handling must explicitly reference these rules (Section 5.4) to ensure generated code adheres to the standard (e.g., "Generate code to query transactions between startDate and endDate using Firestore Timestamps, following rules in Section 5.4").

### 5.5 Sample Firestore Documents

This section provides example JSON structures for key Firestore collections and documents based on the data model defined in Section 5.2. These examples demonstrate the schema design and include realistic sample data for development and testing purposes.

#### 5.5.1 Users Collection

Path: /users/{userId}

{

"email": "jean.dupont@example.com",

"displayName": "Jean Dupont",

"createdAt": "2025-02-15T14:30:00Z",

"settings": {

"currency": "CAD",

"balanceDisplayMode": "cumulative"

}

}

#### 5.5.2 Categories Subcollection

Path: /users/{userId}/categories/{categoryId}

Example 1: Default Category

{

"userId": "user123456",

"name": "Épicerie",

"type": "expense",

"isDefault": true,

"createdAt": "2025-02-15T14:35:00Z"

}



Example 2: Custom Category

{

"userId": "user123456",

"name": "Gardienne",

"type": "expense",

"isDefault": false,

"createdAt": "2025-03-01T10:12:00Z"

}



Example 3: Income Category

{

"userId": "user123456",

"name": "Salaire",

"type": "income",

"isDefault": true,

"createdAt": "2025-02-15T14:35:00Z"

}



#### 5.5.3 Transactions Subcollection

Path: /users/{userId}/transactions/{transactionId}

Example 1: Regular Expense

{

"userId": "user123456",

"categoryId": "cat\_epicerie",

"date": "2025-04-10T00:00:00Z",

"type": "expense",

"description": "Supermarché C&T",

"amount": 87.45,

"isRecurringInstance": false,

"recurringRuleId": null,

"createdAt": "2025-04-10T14:22:15Z",

"updatedAt": "2025-04-10T14:22:15Z"

}



Example 2: Recurring Expense Instance

{

"userId": "user123456",

"categoryId": "cat\_assurance",

"date": "2025-04-09T00:00:00Z",

"type": "expense",

"description": "Assurance Auto",

"amount": 24.00,

"isRecurringInstance": true,

"recurringRuleId": "rule789012",

"createdAt": "2025-01-01T00:10:00Z",

"updatedAt": "2025-01-01T00:10:00Z"

}



Example 3: Income Transaction

{

"userId": "user123456",

"categoryId": "cat\_salaire",

"date": "2025-04-01T00:00:00Z",

"type": "income",

"description": "Salaire mensuel",

"amount": 3200.00,

"isRecurringInstance": true,

"recurringRuleId": "rule\_salaire",

"createdAt": "2025-01-01T00:15:00Z",

"updatedAt": "2025-01-01T00:15:00Z"

}



Example 4: Health Expense with Reimbursement

{

"userId": "user123456",

"categoryId": "cat\_sante",

"date": "2025-04-05T00:00:00Z",

"type": "expense",

"description": "Pharmacie",

"amount": 45.00,

"isRecurringInstance": false,

"recurringRuleId": null,

"createdAt": "2025-04-05T16:30:00Z",

"updatedAt": "2025-04-05T16:30:00Z"

}



Example 5: Health Reimbursement (Income within Health Category)

{

"userId": "user123456",

"categoryId": "cat\_sante",

"date": "2025-04-08T00:00:00Z",

"type": "income",

"description": "Remboursement assurance",

"amount": 36.00,

"isRecurringInstance": false,

"recurringRuleId": null,

"createdAt": "2025-04-08T10:15:00Z",

"updatedAt": "2025-04-08T10:15:00Z"

}

#### 5.5.4 Recurring Rules Subcollection

Path: /users/{userId}/recurringRules/{ruleId}

Example 1: Monthly Expense

{

"userId": "user123456",

"categoryId": "cat\_assurance",

"type": "expense",

"description": "Assurance Auto",

"amount": 24.00,

"frequency": "monthly",

"interval": 1,

"dayOfMonth": 9,

"dayOfWeek": null,

"startDate": "2025-01-09T00:00:00Z",

"endDate": null,

"isActive": true,

"createdAt": "2025-01-01T09:30:00Z",

"updatedAt": "2025-01-01T09:30:00Z"

}



Example 2: Weekly Expense

{

"userId": "user123456",

"categoryId": "cat\_resto",

"type": "expense",

"description": "Repas du vendredi",

"amount": 18.00,

"frequency": "weekly",

"interval": 1,

"dayOfMonth": null,

"dayOfWeek": 5,

"startDate": "2025-01-03T00:00:00Z",

"endDate": "2025-12-26T00:00:00Z",

"isActive": true,

"createdAt": "2025-01-01T10:00:00Z",

"updatedAt": "2025-01-01T10:00:00Z"

}



Example 3: Monthly Income

{

"userId": "user123456",

"categoryId": "cat\_salaire",

"type": "income",

"description": "Salaire mensuel",

"amount": 3200.00,

"frequency": "monthly",

"interval": 1,

"dayOfMonth": 1,

"dayOfWeek": null,

"startDate": "2025-01-01T00:00:00Z",

"endDate": null,

"isActive": true,

"createdAt": "2025-01-01T00:01:00Z",

"updatedAt": "2025-01-01T00:01:00Z"

}

#### 5.5.5 Budgets Subcollection

Path: /users/{userId}/budgets/{yyyy-mm}

Example: April 2025 Budget

{

"userId": "user123456",

"month": "2025-04",

"year": 2025,

"calculated": {

"revenue": 6400.00,

"recurringExpenses": 3252.00,

"rolloverFromPrevious": -2106.00,

"availableToAllocate": 1042.00,

"totalAllocated": 3280.00,

"remainingToAllocate": -2238.00,

"totalSpent": 5358.00,

"monthlySavings": -2210.00

},

"allocations": {

"cat\_epicerie": 730.00,

"cat\_transport": 43.00,

"cat\_resto": 115.00,

"cat\_sante": 148.00,

"cat\_loisirs": 0.00,

"cat\_autres": 1070.00

},

"createdAt": "2025-04-01T00:00:00Z",

"updatedAt": "2025-04-11T12:30:00Z"

}

#### 5.5.6 Example Queries

The following JavaScript examples demonstrate common queries for accessing the Firestore data model:

Get transactions for a specific month:

const startDate = new Date('2025-04-01T00:00:00Z');

const endDate = new Date('2025-05-01T00:00:00Z');

db.collection('users').doc('user123456')

.collection('transactions')

.where('date', '>=', startDate)

.where('date', '<', endDate)

.orderBy('date')

.get()

.then(snapshot => {

const transactions = snapshot.docs.map(doc => ({

id: doc.id,

...doc.data()

}));

console.log('April transactions:', transactions);

});



Get transactions for a specific category:

db.collection('users').doc('user123456')

.collection('transactions')

.where('categoryId', '==', 'cat\_sante')

.orderBy('date', 'desc')

.get()

.then(snapshot => {

const healthTransactions = snapshot.docs.map(doc => ({

id: doc.id,

...doc.data()

}));

console.log('Health transactions:', healthTransactions);

});



Get all active recurring rules:

db.collection('users').doc('user123456')

.collection('recurringRules')

.where('isActive', '==', true)

.get()

.then(snapshot => {

const activeRules = snapshot.docs.map(doc => ({

id: doc.id,

...doc.data()

}));

console.log('Active recurring rules:', activeRules);

});



Update a budget allocation:

const budgetRef = db.collection('users').doc('user123456')

.collection('budgets').doc('2025-04');

// Update a single category allocation

budgetRef.update({

'allocations.cat\_epicerie': 800.00,

'updatedAt': new Date()

}).then(() => {

console.log('Budget allocation updated');

// Recalculate totalAllocated and remainingToAllocate

return budgetRef.get();

}).then(doc => {

const budgetData = doc.data();

const allocations = budgetData.allocations;

// Sum all allocations

const totalAllocated = Object.values(allocations).reduce(

(sum, amount) => sum + amount, 0

);

// Calculate remaining to allocate

const remainingToAllocate =

budgetData.calculated.availableToAllocate - totalAllocated;

// Update calculated fields

return budgetRef.update({

'calculated.totalAllocated': totalAllocated,

'calculated.remainingToAllocate': remainingToAllocate,

'updatedAt': new Date()

});

});

#### 5.5.7 Firestore Security Rules

These security rules enforce data access patterns for the Budget App:

service cloud.firestore {

match /databases/{database}/documents {

// Match the users collection

match /users/{userId} {

// Allow read/write access only to authenticated user's own document

allow read, write: if request.auth != null && request.auth.uid == userId;

// Match nested collections

match /transactions/{transactionId} {

allow read, write: if request.auth != null && request.auth.uid == userId;

}

match /categories/{categoryId} {

allow read, write: if request.auth != null && request.auth.uid == userId;

}

match /budgets/{budgetId} {

allow read, write: if request.auth != null && request.auth.uid == userId;

}

match /recurringRules/{ruleId} {

allow read, write: if request.auth != null && request.auth.uid == userId;

}

}

}

}



These sample documents and queries serve as reference implementations of the data model defined in Section 5.2. They can be used for development, testing, and to better understand the structure and relationships of the data within the KarmaCash app.

#### 5.5.8 Recurring Transaction Generation Algorithm

The following pseudo-code describes the algorithm for generating recurring transaction instances based on recurring rules:

/\*\*

\* Generates recurring transaction instances from rules

\* This would run as a scheduled Cloud Function

\*/

function generateRecurringTransactions() {

// 1. Determine the date range to generate instances for

const today = new Date();

const oneYearFromNow = new Date();

oneYearFromNow.setFullYear(today.getFullYear() + 1);

// 2. Fetch all active recurring rules

const rulesSnapshot = await db.collectionGroup('recurringRules')

.where('isActive', '==', true)

.get();

// 3. For each rule, generate instances within the date range

for (const ruleDoc of rulesSnapshot.docs) {

const rule = ruleDoc.data();

const userId = rule.userId;

// Skip rules with endDate in the past

if (rule.endDate && rule.endDate.toDate() < today) {

continue;

}

// Delete future instances for this rule (to regenerate them)

const existingFuture = await db.collection('users').doc(userId)

.collection('transactions')

.where('recurringRuleId', '==', ruleDoc.id)

.where('date', '>=', today)

.get();

const batch = db.batch();

existingFuture.docs.forEach(doc => batch.delete(doc.ref));

await batch.commit();

// Generate new instances based on frequency

const instances = [];

let currentDate = new Date(Math.max(

rule.startDate.toDate(),

// For retroactive rules, only generate from 3 months ago max

new Date(today.getFullYear(), today.getMonth() - 3, 1)

));

while (currentDate <= oneYearFromNow) {

if (rule.endDate && currentDate > rule.endDate.toDate()) {

break;

}

// Skip dates that don't match the rule criteria

let shouldGenerate = false;

switch (rule.frequency) {

case 'monthly':

shouldGenerate = currentDate.getDate() === rule.dayOfMonth;

break;

case 'weekly':

shouldGenerate = currentDate.getDay() === rule.dayOfWeek;

break;

case 'bi-weekly':

// Calculate if this is a matching week in the bi-weekly cycle

const startWeek = Math.floor(rule.startDate.toDate().getTime() / (7 \* 24 \* 60 \* 60 \* 1000));

const currentWeek = Math.floor(currentDate.getTime() / (7 \* 24 \* 60 \* 60 \* 1000));

shouldGenerate =

(currentDate.getDay() === rule.dayOfWeek) &&

((currentWeek - startWeek) % 2 === 0);

break;

}

if (shouldGenerate) {

instances.push({

userId: rule.userId,

categoryId: rule.categoryId,

date: new Date(currentDate),

type: rule.type,

description: rule.description,

amount: rule.amount,

isRecurringInstance: true,

recurringRuleId: ruleDoc.id,

createdAt: new Date(),

updatedAt: new Date()

});

}

// Advance to the next day

currentDate.setDate(currentDate.getDate() + 1);

}

// Save the generated instances in batches (Firestore batch limit is 500)

for (let i = 0; i < instances.length; i += 400) {

const currentBatch = db.batch();

const chunk = instances.slice(i, i + 400);

chunk.forEach(instance => {

const newRef = db.collection('users').doc(userId)

.collection('transactions').doc();

currentBatch.set(newRef, instance);

});

await currentBatch.commit();

}

}

}



This algorithm ensures that recurring transactions are generated up to one year in the future, allowing users to see their upcoming financial commitments while preventing excessive data storage.

## SECTION 6: DEVELOPMENT PLAN & MILESTONES

### 6.1 Phased Approach

The development will follow a structured, milestone-based approach to ensure steady progress and create functional components that build upon each other:

1. **Milestone 0: Setup & Foundation**
   * Initial project setup, repository creation, and environment configuration
   * This provides the technical foundation for all subsequent development
2. **Milestone 1: Authentication & Basic Layout**
   * Implement Firebase authentication (signup, login, logout) and create the basic application layout with navigation structure
   * This establishes the user identity and interface framework
3. **Milestone 2: Core Data - Categories & Transactions (Read/Write)**
   * Build functionality for managing categories and basic transaction operations
   * This establishes the core data entities and their CRUD operations
4. **Milestone 3: Recurring Transactions & Enhanced Transaction View**
   * Implement recurring transaction rules, transaction generation, and enhanced transaction display views
   * This adds the time-saving recurring functionality and improved transaction visualization
5. **Milestone 4: Core Budgeting Logic & UI**
   * Develop the monthly budget section with allocation, calculation logic, and spending tracking
   * This delivers the primary budgeting value proposition of the application
6. **Milestone 5: Analysis Sections & Navigation Polish**
   * Create visualization components (graphs, category analysis) and refine navigation between sections
   * This provides insightful data visualization and improves the overall user experience
7. **Milestone 6: Final Testing, Refinement & Deployment**
   * Comprehensive testing, UI polish, and production deployment
   * This ensures a high-quality, reliable application ready for regular use

### 6.2 Task Breakdown

#### Milestone 0: Setup & Foundation

* Create GitHub repository with appropriate .gitignore and README
* Initialize React project using Vite
* Set up ESLint and Prettier for code quality
* Create Firebase project (enable Auth, Firestore, Functions, Hosting)
* Add Firebase SDK configuration to React app
* Implement professional logging system with Winston and Loggly
* Set up Winston logger with custom levels (including AI\_CONTEXT)
* Configure Loggly transport for centralized log storage
* Create logger utilities for component/operation tracking
* Implement log extraction tools for AI collaboration
* Configure basic folder structure (components, hooks, utils, etc.)
* Set up deployment pipeline for Firebase Hosting
* Complete basic "Hello World" test deployment

#### Milestone 1: Authentication & Basic Layout

* Create Login component with email/password authentication
* Create Signup component for new user registration
* Implement logout functionality
* Add session persistence using Firebase Auth
* Create protected routes for authenticated content
* Build main application layout (header, navigation tabs, content area)
* Implement basic state management for user context
* Add loading and error states for authentication processes
* Create placeholder views for main sections

#### Milestone 2: Core Data - Categories & Transactions (Read/Write)

* Implement default categories creation for new users
* Create Category management UI (add, edit, delete)
* Build Transaction entry form (date, amount, description, category)
* Implement transaction listing functionality with basic filtering
* Add validation for transaction and category forms
* Create database seed script for testing data
* Implement transaction deletion (including swipe-to-delete on mobile)
* Add toast notifications for CRUD operations
* Set up Firestore security rules for categories and transactions

#### Milestone 3: Recurring Transactions & Enhanced Transaction View

* Create recurring rule management UI
* Implement rule creation and editing flow
* Design and build weekly view for transactions (grouped by day)
* Design and build monthly calendar view with daily totals
* Implement period navigation (week/month, prev/next) with persistence
* Develop Cloud Function for generating recurring transactions
* Add visual indicators for recurring transactions
* Implement expanded/collapsed toggle for transaction groups
* Add confirmation dialog for rule deletion

#### Milestone 4: Core Budgeting Logic & UI

* Create monthly budget view with category allocations
* Implement budget calculation logic (Available Funds, Rollover)
* Build UI for displaying current spending vs. allocation
* Add real-time updates for "Remaining to Allocate"
* Implement save functionality for budget allocations
* Add month navigation for budget section
* Create progress indicators for category spending
* Implement test cases for all budget calculations
* Add visual feedback for budget status (under/over)

#### Milestone 5: Analysis Sections & Navigation Polish

* Implement 'Graphiques' section with pie chart (expenses by category)
* Add bar chart for income vs. expenses vs. balance
* Create 'Catégories' section with categorized transaction lists
* Add balance toggle (Cumulatif/Période) in header
* Implement view mode persistence across sections
* Enhance navigation UX (active tab indicators, transitions)
* Add responsive design refinements for various screen sizes
* Implement expandable category groups in analysis section
* Add sorting/filtering options for transaction lists

#### Milestone 6: Final Testing, Refinement & Deployment

* Perform end-to-end testing of all user flows
* Test application across different browsers and devices
* Add final UI polish (animations, transitions, consistency)
* Optimize performance (code splitting, lazy loading)
* Review and refine Firestore security rules
* Conduct thorough error handling review
* Prepare production environment
* Deploy to production
* Create maintenance/backup plan

### 6.3 Timeline Estimate (Rough & Flexible)

These estimates assume working within the defined 2-hour daily window (20:00-22:00) and are intentionally conservative to account for unforeseen challenges:

* **Milestone 0 (Setup)**: ~1-2 sessions (2-4 hours)
* **Milestone 1 (Auth/Layout)**: ~3-5 sessions (6-10 hours)
* **Milestone 2 (Categories/Basic Transactions)**: ~4-6 sessions (8-12 hours)
* **Milestone 3 (Recurring/Enhanced View)**: ~6-8 sessions (12-16 hours) - Includes backend function
* **Milestone 4 (Core Budgeting)**: ~7-10 sessions (14-20 hours) - Calculation complexity
* **Milestone 5 (Analysis/Nav Polish)**: ~5-7 sessions (10-14 hours)
* **Milestone 6 (Final Testing/Deploy)**: ~3-5 sessions (6-10 hours)

**Total Estimated MVP Range**: Approximately 29-43 sessions (58-86 hours). This translates to roughly 1.5-2.5 months if working consistently 2 hours every single day, but likely longer with real-world breaks/interruptions.

### 6.4 Testing Strategy

#### General Approach

Testing will be primarily manual UI testing performed by the user, supported by console logging and direct inspection of Firestore data via the Firebase console. AI-generated code snippets should be reviewed for logical correctness against the Strategy Document before integration. As features become stable, we can consider adding automated tests (e.g., unit tests for calculation functions) in later phases if desired.

A test data seed script will be developed (likely in Milestone 2/3) to populate Firestore with a consistent set of sample transactions, categories, rules, and budgets for reliable testing of calculations, visualizations, and edge cases.

#### Milestone 0 Testing Strategy (Setup & Foundation)

**Goal**: Verify project setup, Firebase connection, and basic deployment. **Methods**: Manual checks, Firebase console verification. **Test Cases**:

* [Setup] Verify React project (Vite) initializes correctly and runs locally (npm run dev).
* [Git] Confirm Git repository is created on GitHub and initial code is pushed.
* [Firebase] Confirm Firebase project is created and Firestore, Auth, Hosting, Functions services are enabled.
* [Firebase] Verify Firebase SDK configuration is correctly added to the React app and connects without console errors on local startup.
* [Deployment] Deploy the initial basic React app (e.g., showing "Hello World") to Firebase Hosting using firebase deploy.
* [Deployment] Access the live Firebase Hosting URL and confirm the basic app loads correctly.

#### Milestone 1 Testing Strategy (Authentication & Basic Layout)

**Goal**: Verify user authentication works correctly and the main app layout/navigation is functional. **Methods**: Manual UI testing, Firebase Authentication console verification. **Test Cases**:

* [Auth] Sign up with a new email/password. Verify user account is created in Firebase Auth console. Verify user is logged in and redirected to the main app. Verify user document is created in Firestore /users/{userId}.
* [Auth] Log out. Verify user is redirected to the login screen.
* [Auth] Log in with the created account. Verify successful login and redirection to the main app.
* [Auth] Attempt login with incorrect password. Verify error message is shown.
* [Auth] Attempt sign up with an existing email. Verify error message is shown.
* [Auth] Close browser/tab after logging in, reopen the app. Verify the user is still logged in (session persistence).
* [Routing] Attempt to access a main app URL (e.g., /transactions) while logged out. Verify redirection to the login screen.
* [Layout] Verify the header (placeholder balance), main content area, and section navigation tabs are displayed correctly after login.
* [Layout] Click each section tab ('Transactions', 'Graphiques', etc.). Verify the main content area visually switches or indicates the active section (even if content is just placeholder text initially).

#### Milestone 2 Testing Strategy (Core Data - Categories & Transactions)

**Goal**: Verify category management and basic transaction CRUD operations work with Firestore. **Methods**: Manual UI testing, Firestore console data verification. **Test Cases**:

* [Category] Verify default categories are displayed correctly (fetching from /users/{userId}/categories).
* [Category] Add a new custom category (e.g., "Gifts", type "expense"). Verify it appears in the list and a new document exists in Firestore. Verify it appears in the 'Add Transaction' form dropdown.
* [Category] Edit the name of the custom category. Verify the change is reflected in the list and in Firestore.
* [Category] Delete the custom category. Verify it is removed from the list and Firestore. (Consider how to handle transactions already assigned - reassign to 'Autres'?).
* [Transaction] Add a new expense transaction using the form. Verify input validation works (e.g., requires amount). Verify success message. Verify a new document appears in /users/{userId}/transactions with correct data (date as Timestamp, amount as Number, correct categoryId).
* [Transaction] Add a new income transaction. Verify document created correctly in Firestore.
* [Transaction] View the basic transaction list. Verify newly added transactions appear, sorted correctly by date.
* [Transaction] Delete a transaction from the list. Verify it is removed from the UI and the Firestore document is deleted.
* Develop a basic seed script to add initial default categories and a small number of test transactions.

#### Milestone 3 Testing Strategy (Recurring Transactions & Enhanced Transaction View)

**Goal**: Verify recurring rule management, transaction instance generation, and the enhanced Transactions section UI/UX. **Methods**: Manual UI testing, Firestore data verification, potentially running/checking the generation script/function manually. **Test Cases**:

* [Recurring Rule] Add a new recurring rule (e.g., Monthly Rent). Verify UI allows input of all parameters (frequency, date, amount etc.). Verify rule document saved correctly in /users/{userId}/recurringRules.
* [Recurring Rule] Edit an existing rule. Verify changes saved in Firestore.
* [Recurring Rule] Delete a rule. Verify document removed from Firestore. (Verify associated future instances are handled as intended - e.g., deleted).
* [Recurring Instance - Manual Check] Manually verify (or simulate script run) that based on a rule, expected transaction documents would be created with isRecurringInstance: true and the correct recurringRuleId. (Full testing requires the generation logic/function).
* [Transaction View - Weekly] Navigate to the Transactions section (Weekly view). Verify transactions are grouped by day in expandable cards using test data.
* [Transaction View - Monthly] Switch to Monthly view. Verify the calendar grid displays. Verify days with transactions show the calculated daily total amount (not just a dot). Verify clicking a day potentially filters/highlights transactions for that day.
* [Transaction View - Navigation] Verify Prev/Next buttons work correctly in both Weekly and Monthly views, loading the appropriate data.
* [Transaction View - Recurring Display] Verify transactions marked as isRecurringInstance: true are visually distinct (e.g., tag, icon).
* [Transaction View - Delete Recurring] For a recurring instance, verify the options "Supprimer" (Delete Instance) and "Supprimer série" (Delete Rule & Future Instances) appear. Test both options, verifying the correct actions occur in Firestore and UI.
* [Transaction View - Swipe Delete] Verify swipe gesture reveals the delete button(s) on transaction items in lists. Verify button triggers the correct delete action.

#### Milestone 4 Testing Strategy (Core Budgeting Logic & UI)

**Goal**: Verify the Budget section UI is functional and that all budget calculations (Available Funds, Rollover, Spending vs. Allocation) are accurate based on test data. **Methods**: Manual UI testing, Firestore data verification, comparison of UI values against manually calculated expected values using test data. **Test Cases**:

* [Budget UI] Navigate to the Budget section. Verify the UI loads for the current month, displaying category rows, allocation inputs/sliders, spending progress, Available Funds, Remaining to Allocate.
* [Budget Data] Verify a budget document for the current month exists or is created in /users/{userId}/budgets/{yyyy-mm}.
* [Calculation - Available Funds] Using known test data (income, recurring expenses for the month + rollover from previous), manually calculate the expected Available Funds. Verify the value displayed in the UI matches the calculation defined in Section 5.3.4.
* [Calculation - Rollover] Ensure test data includes a previous month with a known positive and a known negative availableToAllocate - totalAllocated. Load the current month's budget. Verify the rolloverFromPrevious displayed matches the previous month's calculated value exactly (positive or negative), per Section 5.3.3.
* [Budget Allocation] Adjust allocation amounts for several categories using the UI. Verify 'Remaining to Allocate' updates instantly and correctly.
* [Budget Save] Click 'Save'. Verify the allocations map in the current month's Firestore budget document is updated correctly.
* [Budget Spending Display] Using test transactions for the month, manually calculate spending per category (net of income in category, excluding recurring expenses). Verify the 'Dépensé' values displayed for each category match the calculation defined in Section 5.3.7. Verify visual progress indicators reflect spending vs. allocation.
* [Budget Navigation] Navigate to a previous month in the Budget section. Verify its historical data (allocations, calculated values) loads correctly. Modify an allocation, save, and verify the past document updates.
* [Budget Navigation] Navigate to a future month. Verify it loads (likely creating a new document) and calculates rollover based on the current month's state. Set future allocations and save.

#### Milestone 5 Testing Strategy (Analysis Sections & Navigation Polish)

**Goal**: Verify the Graphs and Categories sections display accurate data based on the selected period and that navigation state persists correctly. **Methods**: Manual UI testing, comparison of charts/lists against manually calculated expected values using test data. **Test Cases**:

* [Graphs] Navigate to the 'Graphiques' section. Select a period with known test data.
* [Graphs - Pie Chart] Manually calculate the percentage breakdown of non-recurring expenses by category for the period. Verify the pie chart visually matches these percentages and labels/legend are correct.
* [Graphs - Bar Chart] Manually calculate total income, total expenses (recurring + non-recurring), and net balance for the period. Verify the bar chart heights and labels match.
* [Graphs - Navigation] Change the period (Week/Month, Prev/Next). Verify both charts update correctly with data for the new period.
* [Categories] Navigate to the 'Catégories' section. Select a period.
* [Categories - List] Verify categories (including 'Autres', 'Récurrent', custom) with activity in the period are listed with correct total amounts.
* [Categories - Expand] Expand a category row. Verify the individual transactions within that category for the period are displayed correctly.
* [Categories - Navigation] Change the period. Verify the list and totals update correctly.
* [Persistence] Select 'Monthly' view and navigate to March in the 'Transactions' section. Navigate to 'Graphiques'. Verify 'Graphiques' also shows data for March Monthly view. Navigate back to 'Transactions'. Verify it is still on March Monthly view. Repeat for Weekly view and different periods.
* [Header Balance] Select a period. Toggle the 'Solde: Cumulatif / Période' switch. Verify the balance displayed in the header updates according to the logic in Section 5.3.11, comparing against manual calculation using test data.

#### Milestone 6 Testing Strategy (Final Testing, Refinement & Deployment)

**Goal**: Comprehensive end-to-end testing, UI polish verification, and deployment confirmation. **Methods**: Manual end-to-end testing across all features, cross-browser/device checks, deployment verification. **Test Cases**:

* [E2E] Perform full user flows: Sign up -> Add categories -> Add recurring rules -> Add various transactions (manual, recurring, income, expense, refunds) across multiple months -> Set budgets for multiple months -> Verify calculations (rollover, available funds, spending) -> Check all views (Transactions, Graphs, Categories, Budget) for consistency -> Delete transactions/rules/categories.
* [Responsiveness] Test the application on different screen sizes (mobile, tablet, desktop) using browser developer tools and/or real devices. Verify layout adjusts correctly and all features are usable.
* [Cross-Browser] Test core functionality on latest versions of Chrome, Firefox, and Safari (desktop and mobile where possible).
* [Edge Cases] Test scenarios like: first use with no data, budgeting for January (no rollover), months with zero income/expenses, large transaction amounts, editing past data significantly.
* [UI Polish] Verify all UI elements match the intended design, feedback messages are clear, loading states are present, and interactions feel smooth.
* [Deployment] Perform final firebase deploy. Access the production URL. Perform a smoke test (login, add transaction, view budget) to ensure the deployed version is working.

### 6.4 Logging Strategy for Development and Debugging

#### 6.4.1 Purpose

To establish a structured, file-based logging system that provides clear visibility into application behavior, aids in debugging, and facilitates communication with AI tools during development. This strategy acknowledges the constraints of browser-based environments while ensuring comprehensive log capture and storage.

#### 6.4.2 Logging Architecture

Due to browser security constraints, a browser-based React PWA cannot directly write to the local file system. Therefore, our logging strategy employs a dual-component approach:

**Frontend Logger (React PWA)**:

* A lightweight client-side logger that captures logs in the browser
* Formats logs into structured JSON objects
* Buffers and sends logs to a backend service
* Mirrors important logs to the console during development

**Backend Log Sink (Cloud Function)**:

* Receives log batches from the frontend
* Writes logs to rotating files using Winston
* Handles file management (rotation, retention)
* Provides an API for log extraction tools

This architecture ensures we maintain file-based logging for AI collaboration while working within browser security constraints.

**Log Flow Diagram**:

[Browser] → Client Logger → Log Buffer → HTTP POST →

[Cloud Function] → Winston Logger → Rotating Log Files

#### 6.4.3 Logging Levels

Both frontend and backend components will use these consistent log levels:

* **ERROR**: Critical issues that prevent functionality from working
* **WARN**: Potential issues that don't block functionality but could indicate problems
* **INFO**: General information about application flow and key events
* **DEBUG**: Detailed information useful during development
* **TRACE**: Highly detailed information for specific troubleshooting
* **AI\_CONTEXT**: Special level for logs specifically intended for AI review

#### 6.4.4 Frontend Implementation

The frontend logger provides a consistent interface that mirrors traditional logging libraries but handles the browser constraints transparently.

**Client-Side Logger (src/services/logger.js)**:

// src/services/logger.js

import { getCurrentUser } from './firebase/auth'; // Assumes this helper exists

// Define log levels

const LogLevel = {

ERROR: 0,

WARN: 1,

INFO: 2,

DEBUG: 3,

TRACE: 4,

AI\_CONTEXT: 5,

SILENT: Infinity // To disable logging if needed

};

// Map level names to numerical values for comparison

const levelNames = {

error: LogLevel.ERROR,

warn: LogLevel.WARN,

info: LogLevel.INFO,

debug: LogLevel.DEBUG,

trace: LogLevel.TRACE,

ai\_context: LogLevel.AI\_CONTEXT,

};

// Set the active log level based on environment

const configuredLevelName = (import.meta.env.VITE\_LOG\_LEVEL || (import.meta.env.DEV ? 'debug' : 'warn')).toLowerCase();

const ACTIVE\_LOG\_LEVEL = levelNames[configuredLevelName] ?? LogLevel.INFO;

// Simple session ID

const sessionId = Date.now().toString(36) + Math.random().toString(36).substring(2);

// Buffer for logs before sending to backend

let logBuffer = [];

let bufferTimeout = null;

const BUFFER\_MAX\_SIZE = 20; // Send after 20 entries

const BUFFER\_TIMEOUT\_MS = 10000; // Send after 10 seconds max

// Backend log sink endpoint (Cloud Function URL)

const LOG\_SINK\_URL = import.meta.env.VITE\_LOG\_SINK\_URL || '/api/logSink';



async function sendLogsToServer() {

if (logBuffer.length === 0) return;

const logsToSend = [...logBuffer];

logBuffer = []; // Clear buffer immediately

clearTimeout(bufferTimeout);

bufferTimeout = null;

try {

await fetch(LOG\_SINK\_URL, {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(logsToSend),

keepalive: true // Attempt delivery on unload

});

} catch (error) {

console.error('Failed to send logs to server:', error);

// Fallback: store in localStorage temporarily if available

try {

const storedLogs = JSON.parse(localStorage.getItem('pendingLogs') || '[]');

localStorage.setItem('pendingLogs', JSON.stringify([...storedLogs, ...logsToSend]));

} catch (e) {

// If localStorage fails, at least try to show in console

console.error('Failed to store logs locally:', e);

}

}

}

// Check for and send any logs stored in localStorage

function sendStoredLogs() {

try {

const storedLogs = JSON.parse(localStorage.getItem('pendingLogs') || '[]');

if (storedLogs.length > 0) {

localStorage.removeItem('pendingLogs');

logBuffer.push(...storedLogs);

sendLogsToServer();

}

} catch (e) {

console.error('Failed to recover stored logs:', e);

}

}

// Call on app init

sendStoredLogs();

// Central log function

function log(level, component, operation, message, metadata = {}) {

if (level < ACTIVE\_LOG\_LEVEL) return; // Skip logging if below active level

const user = getCurrentUser();

// Prepare log entry matching the structure expected by the backend

const logEntry = {

timestamp: new Date().toISOString(),

level: Object.keys(LogLevel).find(key => LogLevel[key] === level)?.toLowerCase() || 'unknown',

message,

metadata: {

component: component || 'UnknownComponent',

operation: operation || 'unknownOperation',

userId: user ? user.uid : 'anonymous',

sessionId,

url: window.location.href,

userAgent: navigator.userAgent,

...metadata

}

};

// Output to console for local dev debugging

const levelName = logEntry.level;

const consoleMethod = console[levelName] || console.log;

consoleMethod(`[${logEntry.timestamp}] ${levelName.toUpperCase()} (${logEntry.metadata.component}.${logEntry.metadata.operation}): ${message}`, logEntry.metadata);

// Add to buffer and schedule sending

logBuffer.push(logEntry);

if (logBuffer.length >= BUFFER\_MAX\_SIZE) {

sendLogsToServer();

} else if (!bufferTimeout) {

bufferTimeout = setTimeout(sendLogsToServer, BUFFER\_TIMEOUT\_MS);

}

}

// Add event listener to try sending logs before page close

window.addEventListener('beforeunload', sendLogsToServer);

// Export logger methods with consistent interface

const logger = {

error: (component, operation, message, metadata) => log(LogLevel.ERROR, component, operation, message, metadata),

warn: (component, operation, message, metadata) => log(LogLevel.WARN, component, operation, message, metadata),

info: (component, operation, message, metadata) => log(LogLevel.INFO, component, operation, message, metadata),

debug: (component, operation, message, metadata) => log(LogLevel.DEBUG, component, operation, message, metadata),

trace: (component, operation, message, metadata) => log(LogLevel.TRACE, component, operation, message, metadata),

aiContext: (component, operation, message, metadata) => log(LogLevel.AI\_CONTEXT, component, operation, message, metadata)

};

export default logger;



**Usage in Components**:

import logger from '../../services/logger';

function BudgetCalculator() {

const calculateRollover = (availableToAllocate, totalAllocated) => {

logger.aiContext('BudgetCalculator', 'calculateRollover', 'Calculating rollover amount', {

availableToAllocate,

totalAllocated

});

try {

const rollover = availableToAllocate - totalAllocated;

logger.info('BudgetCalculator', 'calculateRollover', 'Rollover calculated successfully', {

result: rollover,

calculation: `${availableToAllocate} - ${totalAllocated} = ${rollover}`

});

return rollover;

} catch (error) {

logger.error('BudgetCalculator', 'calculateRollover', 'Error calculating rollover', {

error: error.message,

stack: error.stack

});

throw error;

}

};

// Component code...

}

#### 6.4.5 Backend Implementation

The backend component is responsible for receiving logs from the frontend and writing them to disk using Winston.

**Log Sink Cloud Function (functions/logSink.js)**:

const functions = require('firebase-functions');

const winston = require('winston');

require('winston-daily-rotate-file');

const path = require('path');

const cors = require('cors')({ origin: true });

// Define custom levels matching frontend

const customLevels = {

levels: {

error: 0,

warn: 1,

info: 2,

debug: 3,

trace: 4,

ai\_context: 5

},

colors: {

error: 'red',

warn: 'yellow',

info: 'green',

debug: 'blue',

trace: 'grey',

ai\_context: 'magenta'

}

};

// Create formatter for JSON structured logs

const logFormatter = winston.format.combine(

winston.format.timestamp(),

winston.format.metadata({ fillExcept: ['message', 'level', 'timestamp'] }),

winston.format.json()

);

// Base directory for logs - in Cloud Functions, use /tmp for ephemeral storage

const LOG\_DIR = process.env.NODE\_ENV === 'production'

? path.join('/tmp', 'logs')

: path.join(\_\_dirname, 'logs');

// Create rotating file transports

const fileTransport = new winston.transports.DailyRotateFile({

filename: path.join(LOG\_DIR, 'application-%DATE%.log'),

datePattern: 'YYYY-MM-DD',

maxSize: '20m',

maxFiles: '14d',

format: logFormatter

});

const aiContextTransport = new winston.transports.DailyRotateFile({

filename: path.join(LOG\_DIR, 'ai-context-%DATE%.log'),

datePattern: 'YYYY-MM-DD',

maxSize: '20m',

maxFiles: '30d', // Keep AI context logs longer

format: logFormatter,

level: 'ai\_context'

});

// Create logger with custom format

const logger = winston.createLogger({

levels: customLevels.levels,

format: logFormatter,

defaultMeta: { service: 'budget-app' },

transports: [

// Console for Function logs

new winston.transports.Console({

format: winston.format.combine(

winston.format.colorize({ colors: customLevels.colors }),

winston.format.simple()

)

}),

fileTransport,

aiContextTransport

]

});

// Cloud Function to receive logs

exports.logSink = functions.https.onRequest((request, response) => {

cors(request, response, async () => {

try {

// Require POST method

if (request.method !== 'POST') {

response.status(405).send('Method Not Allowed');

return;

}

// Get log entries from request body

const logEntries = request.body;

if (!Array.isArray(logEntries)) {

response.status(400).send('Expected array of log entries');

return;

}

// Process each log entry

for (const entry of logEntries) {

const { level, message, metadata = {} } = entry;

// Only accept valid log levels

if (!level || !message || !customLevels.levels.hasOwnProperty(level)) {

continue;

}

// Write to Winston logger

logger.log(level, message, metadata);

}

response.status(200).send({ success: true });

} catch (error) {

console.error('Error processing logs:', error);

response.status(500).send('Error processing logs');

}

});

});



#### 6.4.6 Log Storage Strategy

**Primary Storage**:

* Rotating JSON log files in the Cloud Function environment:
  + application-%DATE%.log - Standard application logs
  + ai-context-%DATE%.log - Special logs for AI collaboration

**Retention Policies**:

* Standard logs: Retained for 14 days
* AI Context logs: Retained for 30 days

**Security Considerations**:

* Log files are not directly accessible via web requests
* Cloud Function is authenticated to prevent unauthorized log submission
* User IDs are included for audit purposes but sensitive data should be redacted

**Note on Cloud Function Ephemeral Storage**: Cloud Functions use ephemeral storage (/tmp directory) which persists only during function instance lifetime. For a small-scale application, this may be sufficient as:

* Function instances often stay "warm" for extended periods
* Our log extraction typically happens within a development session

For more persistent storage, consider:

* Using Firebase Cloud Functions v2 with optional mounted storage
* Adding a Google Cloud Storage backup transport to Winston

#### 6.4.7 Log Extraction for AI Collaboration

The log extraction process allows developers to pull relevant logs for AI assistance:

**1. Direct Function Access**: During development, access logs directly via Firebase CLI:

firebase functions:log --only logSink



**2. Custom Log Extraction Utility**: We'll implement a Node.js utility script to extract and format logs from the log files:

// functions/tools/extract-logs.js

const fs = require('fs');

const readline = require('readline');

const path = require('path');

const glob = require('glob');

async function extractLogs({

fromDate,

toDate = new Date(),

components = [],

operations = [],

levels = ['error', 'warn', 'info', 'ai\_context'],

includeAiContext = true,

maxResults = 100,

outputFormat = 'markdown'

}) {

// Find all relevant log files

const logDir = path.join(\_\_dirname, '..', 'logs');

// Glob pattern for application logs or ai-context logs

const logPatterns = [

path.join(logDir, 'application-\*.log')

];

if (includeAiContext) {

logPatterns.push(path.join(logDir, 'ai-context-\*.log'));

}

// Find all matching log files

const logFiles = [];

for (const pattern of logPatterns) {

const files = glob.sync(pattern);

logFiles.push(...files);

}

// Filter by date if specified

const filteredFiles = logFiles.filter(file => {

// Extract date from filename using regex

const match = file.match(/\d{4}-\d{2}-\d{2}/);

if (!match) return false;

const fileDate = new Date(match[0]);

return (!fromDate || fileDate >= fromDate) && fileDate <= toDate;

});

// Extract and filter logs

const matchingLogs = [];

for (const file of filteredFiles) {

const fileStream = fs.createReadStream(file);

const rl = readline.createInterface({

input: fileStream,

crlfDelay: Infinity

});

for await (const line of rl) {

try {

const log = JSON.parse(line);

// Apply filters

if (levels.length > 0 && !levels.includes(log.level)) continue;

if (components.length > 0 && !components.includes(log.metadata?.component)) continue;

if (operations.length > 0 && !operations.includes(log.metadata?.operation)) continue;

// Check timestamp if fromDate specified

if (fromDate && new Date(log.timestamp) < fromDate) continue;

matchingLogs.push(log);

if (matchingLogs.length >= maxResults) break;

} catch (e) {

console.error(`Error parsing log line: ${e.message}`);

}

}

if (matchingLogs.length >= maxResults) break;

}

// Format results based on outputFormat

if (outputFormat === 'json') {

return JSON.stringify(matchingLogs, null, 2);

} else {

// Default: Markdown

return formatLogsAsMarkdown(matchingLogs);

}

}

function formatLogsAsMarkdown(logs) {

// Group logs by component and operation

const groupedLogs = {};

logs.forEach(log => {

const component = log.metadata?.component || 'unknown';

const operation = log.metadata?.operation || 'unknown';

const key = `${component}:${operation}`;

if (!groupedLogs[key]) {

groupedLogs[key] = [];

}

groupedLogs[key].push(log);

});

// Create markdown output

let output = '## Log Summary\n\n';

// Errors first

const errors = logs.filter(log => log.level === 'error');

if (errors.length > 0) {

output += '### Errors\n\n';

errors.forEach(error => {

output += `- \*\*${error.metadata?.component || 'unknown'}.${error.metadata?.operation || 'unknown'}\*\*: ${error.message}\n`;

if (error.metadata?.error) {

output += ` - Error: ${error.metadata.error}\n`;

}

if (error.metadata?.stack) {

output += ` - Stack: ${error.metadata.stack.split('\n')[0]}\n`;

}

});

output += '\n';

}

// Group contents

Object.entries(groupedLogs).forEach(([key, groupLogs]) => {

const [component, operation] = key.split(':');

output += `### ${component}.${operation}\n\n`;

// Count logs by level

const levelCounts = {};

groupLogs.forEach(log => {

levelCounts[log.level] = (levelCounts[log.level] || 0) + 1;

});

output += '\*\*Log counts by level:\*\*\n\n';

Object.entries(levelCounts).forEach(([level, count]) => {

output += `- ${level}: ${count}\n`;

});

output += '\n';

// For AI context logs, include full details

const aiContextLogs = groupLogs.filter(log => log.level === 'ai\_context');

if (aiContextLogs.length > 0) {

output += '\*\*AI Context Logs:\*\*\n\n';

aiContextLogs.forEach(log => {

output += `- [${new Date(log.timestamp).toLocaleString()}] ${log.message}\n`;

// Add metadata as indented JSON if present

if (Object.keys(log.metadata).length > 0) {

const metadataStr = JSON.stringify(log.metadata, null, 2)

.split('\n')

.map(line => ` ${line}`)

.join('\n');

output += `\`\`\`json\n${metadataStr}\n\`\`\`\n\n`;

}

});

}

});

return output;

}

// Example usage (from CLI)

if (require.main === module) {

// Parse command line arguments

const args = process.argv.slice(2);

const options = {

fromDate: args.includes('--from') ? new Date(args[args.indexOf('--from') + 1]) : null,

components: args.includes('--components') ? args[args.indexOf('--components') + 1].split(',') : [],

levels: args.includes('--levels') ? args[args.indexOf('--levels') + 1].split(',') : ['error', 'warn', 'info', 'ai\_context'],

maxResults: args.includes('--max') ? parseInt(args[args.indexOf('--max') + 1]) : 100,

outputFormat: args.includes('--format') ? args[args.indexOf('--format') + 1] : 'markdown'

};

extractLogs(options).then(result => {

console.log(result);

}).catch(err => {

console.error('Error extracting logs:', err);

});

}

module.exports = { extractLogs };



**3. Dev Tools Integration**: For development convenience, we'll add a simple log viewer UI when the application is running in development mode:

// src/components/dev/LogViewer.jsx (only included in dev builds)

import React, { useState, useEffect } from 'react';

import ReactMarkdown from 'react-markdown';

function LogViewer() {

const [logs, setLogs] = useState('Loading logs...');

const [filters, setFilters] = useState({

components: [],

levels: ['error', 'warn', 'info', 'ai\_context'],

maxResults: 50

});

const fetchLogs = async () => {

try {

const response = await fetch('/api/dev/logs', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify(filters)

});

if (!response.ok) throw new Error('Failed to fetch logs');

const result = await response.text();

setLogs(result);

} catch (error) {

setLogs(`Error loading logs: ${error.message}`);

}

};

useEffect(() => {

fetchLogs();

}, []);

return (

<div className="dev-log-viewer">

<h2>Development Log Viewer</h2>

<div className="filters">

{/\* Simple filter controls \*/}

<button onClick={fetchLogs}>Refresh Logs</button>

</div>

<div className="log-content">

<ReactMarkdown>{logs}</ReactMarkdown>

</div>

</div>

);

}

// Only export in development mode

export default process.env.NODE\_ENV === 'development' ? LogViewer : () => null;

#### 6.4.8 Deployment Considerations

**Environment Configuration**:

* Set appropriate logging levels in different environments using environment variables (VITE\_LOG\_LEVEL)
* Configure the log sink URL in production build

**Firebase Setup**:

* Deploy the logSink Cloud Function during Milestone 0 setup
* Configure CORS for the function to accept requests from your application domains
* Consider adding authentication for the log sink function in production

**Log Access During Development**:

* Use Firebase CLI to access function logs
* Use the log extraction utility to prepare logs for AI sessions
* Consider adding a development-only UI route to view logs within the application

### 6.5 React Component Structure & Standards

This section outlines the recommended structure, organization, and coding conventions for React components within the KarmaCash project. Adhering to these standards will promote consistency, maintainability, and collaboration.

#### 6.5.1 Folder Organization

We will adopt a hybrid approach, organizing components primarily by feature while maintaining a separate directory for shared, reusable UI elements.

/src

|-- /app # Core app setup, routing, global providers (e.g., main App.jsx, router setup)

|-- /assets # Static assets (images, icons - if not inline SVGs)

|-- /components # Shared, reusable UI components (presentation-focused)

| |-- /ui # Generic UI primitives (Button.jsx, Input.jsx, Card.jsx, Modal.jsx, Spinner.jsx)

| |-- /layout # Layout structure components (Header.jsx, Footer.jsx, MainLayout.jsx)

| |-- /navigation # Shared navigation elements (MainNav.jsx, DateNav.jsx, PeriodToggle.jsx)

| |-- /visualization # Shared Chart components (PieChart.jsx, BarChart.jsx - wrappers around charting library)

|-- /features # Components, hooks, and logic specific to a feature area

| |-- /auth # LoginScreen.jsx, SignupScreen.jsx, useAuth.js

| |-- /transactions # TransactionList.jsx, TransactionItem.jsx, AddTransactionForm.jsx, CalendarView.jsx, useTransactions.js

| |-- /budget # BudgetScreen.jsx, BudgetAllocationItem.jsx, BudgetBar.jsx, AvailableFunds.jsx, useBudget.js

| |-- /categories # CategoryList.jsx, CategoryItem.jsx, useCategories.js

| |-- /graphs # GraphsScreen.jsx, ExpensePieChart.jsx (feature-specific implementation), BalanceBarChart.jsx

|-- /hooks # Global custom React hooks (e.g., useUserSettings.js, usePersistentState.js)

|-- /contexts # Global state contexts (AuthContext.js, SettingsContext.js)

|-- /services # Abstractions for external services (e.g., /firebase/firestore.js, /firebase/auth.js)

|-- /styles # Global CSS, base styles, theme variables (e.g., index.css, theme.css)

|-- /utils # Utility functions (e.g., dateUtils.js, formatCurrency.js, calculationUtils.js)

|-- /types # TypeScript type definitions (if using TS, otherwise optional for prop-types)

|-- main.jsx # Application entry point



**Rationale**:

* **/components**: Contains highly reusable, often stateless UI components. These should be generic and unaware of specific application features.
* **/features**: Encapsulates components, hooks, and potentially feature-specific services related to a major application area (Transactions, Budget, etc.). This promotes modularity and makes it easier to understand the scope of a feature. Feature components often compose shared UI components from /components.
* **Clear separation of concerns**: Hooks, Contexts, Services, Utils, Styles are kept in dedicated top-level folders.

### 6.5.2 Component Hierarchy (Conceptual)

This illustrates the typical nesting of components. Refer to [B3.7] for detailed descriptions of each component.

App / MainLayout (/components/layout)

├── Header (/components/layout)

│ ├── AppTitle

│ └── BalanceDisplay

│ └── PeriodToggle (/components/navigation)

├── MainNav (/components/navigation)

├── Current Feature View (Routed Content)

│ ├── TransactionsScreen (/features/transactions)

│ │ ├── DateNav (/components/navigation)

│ │ ├── TransactionList (/features/transactions)

│ │ │ └── TransactionDayCard

│ │ │ └── TransactionItem (/features/transactions)

│ │ │ └── DeleteButton (/components/ui/Button.jsx)

│ │ ├── CalendarView (/features/transactions)

│ │ │ └── CalendarDayCell

│ ├── GraphsScreen (/features/graphs)

│ │ ├── DateNav (/components/navigation)

│ │ ├── ExpensePieChart (/features/graphs) composing PieChart (/components/visualization)

│ │ └── BalanceBarChart (/features/graphs) composing BarChart (/components/visualization)

│ ├── CategoriesScreen (/features/categories)

│ │ ├── DateNav (/components/navigation)

│ │ ├── CategoryList (/features/categories)

│ │ │ └── CategoryItem (/features/categories)

│ ├── AddTransactionScreen (/features/transactions) /\* Or maybe a dedicated 'Add' feature \*/

│ │ └── AddTransactionForm (/features/transactions)

│ │ ├── /\* various Input, Select, DatePicker from /components/ui \*/

│ │ └── Button (/components/ui/Button.jsx)

│ ├── BudgetScreen (/features/budget)

│ │ ├── DateNav (/components/navigation) /\* Month Navigation Only \*/

│ │ ├── AvailableFunds (/features/budget)

│ │ ├── BudgetAllocationList

│ │ │ └── BudgetAllocationItem (/features/budget)

│ │ │ └── BudgetBar (/features/budget)

│ │ ├── RemainingToAllocate (/features/budget)

│ │ ├── MonthlySavingsDisplay (/features/budget)

│ │ └── Button (/components/ui/Button.jsx) /\* Save, Reset \*/

└── Footer (/components/layout)

### 6.5.3 State Management Approach

#### Global State (Cross-Feature):

* **React Context API**: Preferred for MVP due to simplicity and built-in nature.
* **AuthContext**: Manages user authentication state (user object, loading, errors). Provides currentUser. Located in /contexts.
* **SettingsContext**: Manages global user preferences like persisted date range (selectedPeriod, viewMode: 'week' | 'month'), balance calculation mode (balanceMode: 'cumulative' | 'period'). Located in /contexts.

#### Feature State (State shared within a feature):

* **Custom Hooks**: Encapsulate data fetching, state logic, and actions related to a feature (e.g., useTransactions(selectedPeriod) in /features/transactions, useBudget(monthYear) in /features/budget). These hooks handle interaction with /services (e.g., Firestore).
* **Prop Drilling**: Acceptable for shallow component trees within a feature.
* **Feature-Specific Context**: Consider if prop drilling becomes excessive within a complex feature (e.g., BudgetContext for managing draft allocations before saving).

#### Local Component State:

* **useState**: Default for simple UI state (input values, toggles, conditional rendering flags).
* **useReducer**: Use for more complex state logic within a component or when the next state depends on the previous one (e.g., managing form state with validation).

### 6.5.4 Example Code Structure (Key Components)

These examples illustrate basic structure, props, and adherence to the style guide. (Note: Assumes basic setup with CSS Modules or a CSS-in-JS solution for styling).

**1. TransactionItem Component (/features/transactions/TransactionItem.jsx)**

import React from 'react';

import PropTypes from 'prop-types';

import { formatCurrency, formatDate } from '../../utils'; // Example utils

import styles from './TransactionItem.module.css'; // Example CSS Modules

import Button from '../../components/ui/Button'; // Reusable Button

// Corresponds to [B3.7.3] Transaction Item

function TransactionItem({ transaction, onDelete, showSeriesDelete }) {

const { id, date, description, category, amount, type, isRecurring, recurringId } = transaction;

// Determine amount color based on type (using Style Guide colors [B3.8.1])

const amountStyle = type === 'income' ? styles.incomeAmount : styles.expenseAmount;

// Potentially map category to a color dot using [B3.8.1]

const handleDelete = () => {

if (window.confirm('Supprimer cette transaction ?')) {

onDelete(id);

}

};

const handleDeleteSeries = () => {

if (window.confirm('Supprimer toute la série récurrente ?')) {

// Need a different handler prop for series deletion

console.warn("Series delete handler not implemented yet for ID:", recurringId);

// onDeleteSeries(recurringId);

}

};

return (

<div className={styles.transactionItem}>

{/\* Optional: Swipe detection wrapper \*/}

<div className={styles.details}>

<span className={styles.category}>{category?.name || 'Uncategorized'}</span>

<span className={styles.description}>{description}</span>

<span className={styles.date}>{formatDate(date)} - {type}</span>

{isRecurring && <span className={styles.recurringIndicator}>Récurrent</span>}

{/\* Optional: Display recurring orange bar from [B3.8.1] here if needed \*/}

</div>

<div className={styles.amountContainer}>

<span className={`${styles.amount} ${amountStyle}`}>

{formatCurrency(amount)}

</span>

{/\* Actions revealed on swipe/click - simplified here \*/}

<div className={styles.actions}>

<Button variant="danger-outline" size="small" onClick={handleDelete}>

Supprimer

</Button>

{showSeriesDelete && isRecurring && (

<Button variant="danger" size="small" onClick={handleDeleteSeries}>

Supprimer série

</Button>

)}

</div>

</div>

{/\* Simple swipe action placeholder \*/}

{/\* More complex implementation would involve touch event handlers or a library \*/}

</div>

);

}

TransactionItem.propTypes = {

transaction: PropTypes.shape({

id: PropTypes.string.isRequired,

date: PropTypes.object.isRequired, // Or string/number depending on Firestore data

description: PropTypes.string,

category: PropTypes.shape({ name: PropTypes.string }), // Assuming category is object/fetched

amount: PropTypes.number.isRequired,

type: PropTypes.oneOf(['income', 'expense']).isRequired,

isRecurring: PropTypes.bool,

recurringId: PropTypes.string, // ID of the rule if it's a recurring instance

}).isRequired,

onDelete: PropTypes.func.isRequired,

showSeriesDelete: PropTypes.bool, // Prop to control if series delete button is visible

// onDeleteSeries: PropTypes.func, // Add if implementing series delete

};

export default TransactionItem;



**2. BudgetBar Component (/features/budget/BudgetBar.jsx)**

import React from 'react';

import PropTypes from 'prop-types';

import { formatCurrency } from '../../utils';

import styles from './BudgetBar.module.css'; // Example CSS Modules

// Corresponds partially to [B3.7.4] Budget Allocation Bar visual

function BudgetBar({ allocated, spent, categoryName }) {

// Handle null/undefined values

const spentValid = spent ?? 0;

const allocatedValid = allocated ?? 0;

const percentage = allocatedValid > 0 ? Math.min((spentValid / allocatedValid) \* 100, 100) : 0;

const isOverBudget = spentValid > allocatedValid && allocatedValid > 0;

const remaining = allocatedValid - spentValid;

let barStyle = styles.progressBar;

let statusText = `${formatCurrency(remaining)} restant`;

if (isOverBudget) {

barStyle = `${styles.progressBar} ${styles.overBudget}`;

statusText = `${formatCurrency(Math.abs(remaining))} dépassé`;

} else if (percentage === 100) {

barStyle = `${styles.progressBar} ${styles.fullySpent}`;

statusText = 'Budget atteint';

} else if (allocatedValid <= 0 && spentValid > 0) {

barStyle = `${styles.progressBar} ${styles.overBudget}`; // Spending without allocation

statusText = `${formatCurrency(spentValid)} dépensé (non alloué)`;

} else if (allocatedValid <= 0) {

statusText = 'Non budgété';

}

return (

<div className={styles.budgetBarContainer} aria-label={`Budget pour ${categoryName}`}>

<div className={styles.labels}>

<span>Dépensé: {formatCurrency(spentValid)}</span>

<span>Alloué: {formatCurrency(allocatedValid)}</span>

</div>

<div className={styles.progressBarBackground}>

<div

className={barStyle}

style={{ width: `${isOverBudget ? 100 : percentage}%` }} // Fill bar fully if overspent

role="progressbar"

aria-valuenow={spentValid}

aria-valuemin="0"

aria-valuemax={allocatedValid}

aria-valuetext={statusText}

></div>

</div>

<div className={styles.statusText}>{statusText}</div>

</div>

);

}

BudgetBar.propTypes = {

allocated: PropTypes.number, // Allow null/undefined if not set

spent: PropTypes.number, // Allow null/undefined if no spending

categoryName: PropTypes.string.isRequired, // For aria-label

};

export default BudgetBar;

## 6.6 Error Handling Strategy

This section outlines the strategy for handling, logging, and displaying errors within the KarmaCash application, ensuring consistency, clarity, and alignment with the Zen/Tranquility UI/UX guidelines ([B3.4]).

### 6.6.1 Goals

* **Graceful Degradation**: Prevent application crashes and provide a stable user experience even when errors occur.
* **User Clarity**: Inform users about issues clearly and calmly, without causing unnecessary alarm, and guide them towards resolution where possible.
* **Developer Insight**: Provide sufficient diagnostic information through logging ([B6.4.2]) to enable efficient debugging and resolution.
* **Consistency**: Apply standard patterns for error handling across the application.

### 6.6.2 Error Categories & Severity Levels

Errors will be broadly categorized based on their source and impact:

| **Category** | **Description** | **Severity** | **User Impact** | **Example** |
| --- | --- | --- | --- | --- |
| Input Validation | User input fails validation rules (e.g., missing required field, invalid format). | Low | User needs to correct input. Action blocked. | Invalid amount format in Transaction Form. |
| Data Integrity | Inconsistency found in stored data (unlikely with Firestore rules, but possible). | Medium | Functionality may be impaired. Requires investigation. | A calculation depends on malformed data. |
| Network/Connectivity | Client cannot reach Firebase services or the backend log sink. | Medium | Application is partially/fully unusable offline. | Fetching transactions fails due to no internet. |
| Firebase Service | Errors returned by Firebase services (Auth, Firestore, Functions). | High | Core functionality failure (e.g., login, data sync). | Firestore query denied due to security rules. |
| Calculation Error | An internal calculation fails unexpectedly (e.g., due to unforeseen data state). | Medium | Financial summaries may be incorrect. | Budget rollover calculation fails. |
| Application Logic | Unexpected errors within the client-side application code (bugs). | High | Component crash or unexpected behavior. | cannot read property 'x' of undefined. |
| Permissions Error | User attempts an action they are not authorized to perform. | Low/Med | Action blocked. Expected for security. | Unauthorized user tries to edit shared budget (future). |

**Severity Levels**:

* **Low**: Minor issues, typically user-correctable (e.g., validation errors).
* **Medium**: Affects specific functionality but doesn't crash the app; may require user retry or investigation (e.g., network timeout, data inconsistency).
* **High**: Critical errors that prevent core functionality or cause instability (e.g., service outages, unhandled exceptions).

### 6.6.3 User Error Presentation (Zen Theme)

Error messages shown to the user should follow [B3.4]'s principle of "Soothing Error Handling".

**Placement**:

* **Validation Errors**: Display inline, near the relevant form field, using subtle text styling (e.g., Text Primary color, possibly smaller font size) perhaps accompanied by a soft icon. Avoid jarring red text directly if possible; use Negative Accent Color ([B3.8 v2] Soft Terra Cotta #C17C74) for borders or icons sparingly.
* **General Errors (Medium/High)**: Use non-modal "Toast" notifications or a subtle inline message area. These should appear gently ([B3.11]), use theme colors (e.g., Negative Accent Color background at low opacity, or a neutral background with a colored icon), and be dismissible. Avoid full-screen blocking overlays unless absolutely critical.

**Tone & Language**:

* Use calm, clear, and simple language (in French, see [B6.6.4]).
* Avoid technical jargon.
* Explain what happened concisely and what the user can do (e.g., "Veuillez vérifier le format du montant." / "Impossible de synchroniser. Vérifiez votre connexion et réessayez.").

**Visual Style**:

* Adhere strictly to [B3.8 v2].
* Error states on inputs might involve a subtle border change (e.g., to Soft Terra Cotta) rather than filled backgrounds.
* Error messages should use standard text styles, potentially with an accompanying Heroicon (e.g., exclamation-triangle) styled with the Negative Accent Color.

### 6.6.4 Standard Error Message Templates (French)

This subsection lists standard French error messages mapped to common error scenarios.

**Validation**:

* Required Field: "Ce champ est requis."
* Invalid Format (Email): "Veuillez entrer une adresse courriel valide."
* Invalid Format (Amount): "Veuillez entrer un montant numérique valide."
* Number Out of Range: "La valeur doit être entre [MIN] et [MAX]."

**Network/Connectivity**:

* General Offline: "Connexion Internet requise. Veuillez vérifier votre réseau."
* Sync Failed: "La synchronisation a échoué. Réessayer?" (with retry button)

**Firebase/Server**:

* Authentication Failed: "Courriel ou mot de passe incorrect."
* General Save Failed: "Erreur lors de l'enregistrement. Veuillez réessayer."
* Data Load Failed: "Impossible de charger les données. Veuillez réessayer."

**Generic/Unexpected**:

* "Une erreur inattendue est survenue. Veuillez réessayer plus tard."
* (Optionally include a Correlation ID for support): "... Si le problème persiste, notez cet ID : [CorrelationID]."

### 6.6.5 Error Logging Procedures

All errors, regardless of severity or whether they are shown to the user, MUST be logged using the logger defined in [B6.4.2].

**Level**:

* Use logger.error() for High severity errors
* Use logger.warn() or logger.error() for Medium severity depending on context
* Low severity errors (like standard input validation) might be logged at logger.info() or logger.debug() if useful for flow analysis, or logger.warn() if they indicate a pattern of user confusion
* Uncaught exceptions MUST be logged as error or fatal

**Context**:

* Provide comprehensive metadata ([B6.4.2]):
  + component: Name of the component where the error was caught/occurred
  + operation: The specific action being performed (e.g., 'submitTransaction', 'fetchBudget', 'handleLogin')
  + error: The raw error object (error.message, error.stack, error.code)
  + payload: Relevant inputs or state at the time of the error (avoid logging sensitive PII)
  + userId and sessionId (handled automatically by the logger)

**Correlation ID**:

* For operations involving multiple steps (e.g., data fetch -> calculation -> save), include a unique correlationId in all related log entries (info, warn, error) to trace the flow.

### 6.6.6 Error Handling Patterns (React)

**Synchronous Code**:

* Use standard try...catch blocks.

try {

// Code that might throw an error

const result = someCalculation(data);

// ...

} catch (error) {

logger.error('MyComponent', 'someCalculation', 'Calculation failed', { error });

// Update UI state to show error message

setUserErrorMessage("Le calcul a échoué.");

}



**Asynchronous Code (Promises/async/await)**:

* Use try...catch with async/await or .catch() handler for Promises.

// Async/Await example

async function fetchData() {

const operation = 'fetchData';

try {

setIsLoading(true);

logger.debug('MyComponent', operation, 'Starting fetch');

const response = await firebaseService.getData();

setData(response);

logger.debug('MyComponent', operation, 'Fetch successful', { count: response.length });

} catch (error) {

logger.error('MyComponent', operation, 'Fetch failed', { error });

// Set user-facing error state

setUserErrorMessage("Impossible de charger les données.");

} finally {

setIsLoading(false);

}

}

// Promise.catch example

firebaseService.saveData(newData)

.then(() => {

logger.info('MyComponent', 'saveData', 'Save successful');

// Show success feedback

})

.catch(error => {

logger.error('MyComponent', 'saveData', 'Save failed', { error, payload: { /\* sensitive fields omitted \*/ } });

setUserErrorMessage("Erreur lors de l'enregistrement.");

});



**Error Boundaries**:

* Implement one or more React Error Boundaries at strategic levels (e.g., around major feature routes, or a global fallback) to catch rendering errors in child components. The Error Boundary component should:
  + Log the error and component stack using componentDidCatch
  + Display a fallback UI (aligned with the Zen theme – e.g., a simple message "Une erreur s'est produite dans cette section.", potentially with a reload button) instead of a crashed component tree

# SECTION 7: AI COLLABORATION WORKFLOW

## 7.1 Purpose and Overview

This section defines the structured approach for collaboration between the human developer and AI assistants (primarily Claude and Cursor AI) during the development of the KarmaCash app. It establishes:

* Specific roles and responsibilities for each participant
* Document templates and communication protocols
* Context preservation techniques across development sessions
* Strategies for working within AI token limitations
* Integration with the logging system for enhanced debugging

The workflow is designed to maintain comprehensive context throughout the development process using a streamlined, single-document handoff system while optimizing for the token limitations of AI assistants.

## 7.2 Collaboration Roles

### 7.2.1 Human Developer

* Project owner and decision maker
* Provides initial context and requirements via the Bible and session planning
* Reviews and implements AI-suggested code
* Tests functionality and reports issues
* Maintains code repository and documentation (including the Bible)
* Structures communication to optimize AI assistance
* Initiates the creation of the Session Handoff & Initialization Plan document near the end of each work session

### 7.2.2 Claude (Primary Context Keeper)

* Serves as the primary context keeper and strategic guide during a work session
* Helps interpret and apply Bible document requirements based on provided context/excerpts
* Generates detailed prompts for Cursor AI (if applicable)
* Provides architectural guidance and code review
* Assists with debugging and problem-solving using provided logs/context
* Collaborates with the Human Developer at the end of each session to generate the Session Handoff & Initialization Plan for the next session
* Relies on the provided Handoff/Plan document for context continuity at the start of a new session

### 7.2.3 Cursor AI (Code Generation Assistant)

* Generates code directly in the development environment
* Implements specific components and functions based on prompts (often generated with Claude's help)
* Refactors code based on feedback
* Suggests optimizations and improvements
* Provides immediate coding assistance within VS Code

## 7.3 Document Structure

The project maintains a carefully structured document approach to optimize context sharing:

### 7.3.1 The Bible (Complete Reference)

* Comprehensive strategy document divided into sections
* Maintained in Google Drive with section-specific documents
* Serves as the definitive source of truth for the project
* Contains detailed specifications, requirements, and implementation guidelines
* Referenced using [BX.Y] notation (e.g., [B5.3]). May be too large to share in full; relevant excerpts or specific sections will be provided or referenced as needed

### 7.3.2 Session & Component Documents

**Session Handoff & Initialization Plan**:

* Single primary document created collaboratively at the end of each work session
* Contains a recap of the completed session and the detailed plan for the next session
* Serves as the primary context input for starting the next session
* Follows the template defined in [B7.6.1]

**Component Context Document**:

* Created as needed for specific complex component or feature implementations
* Contains relevant Bible extracts, visual references, code templates, and implementation guidance
* Provided alongside the Session Handoff & Initialization Plan at relevant session starts
* Follows the template defined in [B7.6.2]

## 7.4 Document Organization

Project documentation is organized in Google Drive as follows:

Budget App Rebuild/

├── Bible/ # Complete strategy document files (Section 1, 2, etc.)

│ ├── Section 1 - Introduction.docx

│ ├── ...

│ └── Section 7 - AI Collaboration Workflow.docx # This document

├── Session Templates/ # Reusable document templates

│ ├── Component Context Template.docx

│ └── Session Handoff & Initialization Plan Template.docx # New combined template

├── Work Sessions/ # Organized by milestone and session

│ ├── Milestone 0/

│ │ ├── Session 0.1 - Project Setup/

│ │ │ ├── Session Handoff & Initialization Plan (End of S0.1, Plan for S0.2).docx

│ │ │ └── Component Context - Logging Setup.docx

│ │ └── Session 0.2 - Logging Setup/

│ │ └── Session Handoff & Initialization Plan (End of S0.2, Plan for S0.3).docx

│ ├── Milestone 1/

│ └── etc...

├── Log Summaries/ # Extracted logs by milestone/session (optional)

└── Visuals/ # Screenshots, mockups, diagrams organized by section/feature

└── Section3/

└── screenshot-transaction-list.png

## 7.5 Workflow Process

### 7.5.1 Session Preparation (Start of Session N+1)

**Review Previous Context & Plan**:

* Human Developer locates the Session Handoff & Initialization Plan generated at the end of the previous session.

**Prepare Component Context (If Needed)**:

* Prepare or update a Component Context Document [B7.6.2] if detailed context beyond the main Bible is required.

**Gather Resources**:

* Collect necessary resources listed in the "Required Resources" section (e.g., Bible files, visual aids, log summaries).

### 7.5.2 Session Execution (During Session N+1)

**Initialize AI Context**:

* Human Developer provides Claude with the Session Handoff & Initialization Plan, Component Context Document (if available), and necessary resources.

**Confirm Understanding**:

* AI confirms understanding of session goals, tasks, deliverables, and parameters.

**Collaborative Development**:

* Collaborate on tasks, test implementations, provide feedback, and iterate as needed.

**Issue Resolution**:

* Utilize logging system [B6.4.1] and provide logs or code snippets for debugging assistance.

### 7.5.3 Session Closure (End of Session N+1)

**Initiate Handoff Creation**:

* Human Developer prompts AI to create the Session Handoff & Initialization Plan for the next session.

**Collaboratively Generate Handoff/Plan Document**:

* Document session achievements, challenges, and define the plan for the next session.

**Save and Commit**:

* Save the finalized document in Google Drive and commit any code changes to Git.

## 7.6 Document Templates

### 7.6.1 Session Handoff & Initialization Plan Template

(This is the primary template used to bridge sessions)

# Session Handoff & Initialization Plan: [Date] - Session #[Number]

## 1. AI Instructions & Workflow Reference

- Your primary role is context keeper and collaborator for Session #[N+1] (the session starting \*now\*).

- Refer to this document for context from Session #[N] (previous) and the plan for Session #[N+1] (this session).

- \*\*Detailed workflow processes, templates, and notation guides are in the Bible, Section 7 [B7]. Refer to [B7] if workflow questions arise.\*\*

- (Note: Access to the full Bible may be limited. Rely on provided excerpts/context. If complex details from a specific Bible section [BX.Y] are required for a task, instructions to consult it will be given explicitly.)

- Your goal is to assist in achieving the 'Plan for Session #[N+1]' outlined below (Section 4).

- At the end of Session #[N+1], you \*\*must\*\* help generate the \*next\* 'Session Handoff & Initialization Plan' document (for Session N+2). This is typically initiated by the Human Developer.

- Use ```jsx code blocks for code examples. Use [BX.Y] notation for specific Bible references mentioned herein.

## 2. Project Context & Status

- Project: Budget App Rebuild

- Current Milestone: [Current Milestone, e.g., Milestone 0, Milestone 1]

- Overall Goal: [Briefly state, e.g., Complete MVP as defined in Bible]

## 3. Recap of Previous Session (Session #[N] - [Date of Prev Session])

\*(This section details what was completed in the immediately preceding session)\*

### 3.1. Achievements

- [Achievement 1 from Session N]

- [Achievement 2 from Session N]

- [...]

### 3.2. Implementation Details

- Files created/modified:

- [Filename]: [Brief description of change/purpose]

- [...]

- Key technical decisions:

- [Decision made]: [Rationale/Reasoning]

- [...]

### 3.3. Challenges & Solutions Encountered

- [Challenge Name/Area]:

- Problem: [Concise description of the issue]

- Solution: [How it was resolved or mitigation strategy]

- [...]

### 3.4. Visual Progress (Optional)

- [Link to screenshots, or brief description if visual change occurred]

### 3.5. Key Log Insights (If applicable)

- [Summary of relevant log findings or link to extracted log summary file]

## 4. Plan for Current Session (Session #[N+1] - Today, [Date])

\*(This section outlines the specific plan for the session starting now)\*

### 4.1. Session Goal/Focus

- [\*\*Clear, concise statement of the primary objective for THIS session.\*\* Example: "Populate initial drafts of French Glossary and UI Component Spec based on Section 3 and existing screenshots."]

### 4.2. Component/Area of Focus

- [Specific component(s), Bible section(s), or functional area being worked on. Example: "Bible Section 3 Enhancement Artifacts"]

### 4.3. Specific Tasks for This Session

1. [Action-oriented task 1. Example: "Review Bible Section 3 User Stories and Flows."]

2. [Action-oriented task 2. Example: "Extract all French UI text visible in `screenshot-A.png`, `screenshot-B.png`."]

3. [Action-oriented task 3. Example: "Populate extracted terms into `French Terminology Glossary.docx`."]

4. [...]

5. [Action-oriented task N. Example: "Collaboratively generate the Handoff & Plan document for Session N+2."] \*(Usually the last task)\*

### 4.4. Required Resources for This Session

- Bible Sections: [List using notation, e.g., [B3], [B7.4]. Note if full section access needed vs. excerpts.]

- Component Context Document(s): [Link/Filename if a specific component context doc is needed, otherwise N/A]

- Visual References: [e.g., `screenshot-A.png` (Link or indicate to be provided by Human)]

- Existing Files: [e.g., `Bible/Section 3 - Core Features & User Experience.docx`]

- Other: [e.g., Access to Google Drive folder structure, Specific calculation logic from [B5.3]]

### 4.5. Expected Deliverables/Outputs for This Session

\*(What artifacts should exist or be updated by the end of this session?)\*

- [Deliverable 1: e.g., `French Terminology Glossary.docx` (created/updated in `Bible/Glossaries/`)]

- [Deliverable 2: e.g., Implemented `TransactionItem` React component file]

- [...]

- [Deliverable N: e.g., Completed `Session Handoff & Initialization Plan` document for Session N+2]

### 4.6. Session Parameters

- Time Boundary: [e.g., 20:00-22:00 EDT, adhering to B8.2]

- Out-of-Scope for This Session: [Explicitly state what will NOT be tackled. Example: "Backend function implementation", "User authentication flow"]

### 7.6.2 Component Context Template

This template is used as needed for specific complex components:

# Component Context: [Component Name]

## Visual Reference

- [Link to screenshot, mockup, or diagram of the component]

- [Example: `../Visuals/Section3/TransactionItem.png`]

## Component Purpose

- [Description of what this component does and how it fits in the application, referencing relevant User Story IDs from B3 if possible.]

## User Flow Integration

- [Describe how this component fits into one or more Key User Flows from B3.3]

## Relevant Bible Extracts

- [\*\*Copy-paste the most relevant snippets\*\* from the Bible here, e.g., specific User Stories from B3.1, UI/UX Guidelines from B3.4, Data Model fields from B5.2, Calculation Logic from B5.3]

- Example:

- "[B3.1.6]: As a user, I want to delete a specific transaction (e.g., via a button or swipe gesture)..."

- "[B5.2 Transaction Fields]: `description: String`, `amount: Number`, `date: Timestamp`..."

## UI Specifications (If applicable beyond global B3.4)

- Specific interactions: [e.g., Describe swipe gesture behavior]

- State variations: [e.g., How does it look when selected, disabled, etc.?]

- Responsive notes: [Any specific adjustments needed for this component]

## Implementation Requirements

- Key Props: [List expected input props, e.g., `transaction: TransactionData`, `onDelete: function`]

- Key State: [Internal state the component might manage]

- Data fetching/mutation: [Does it interact directly with Firestore? Which functions/hooks?]

- Event Handling: [What user events does it need to handle? e.g., `onClick`, `onSwipeLeft`]

- Error handling: [Specific error states or messages related to this component]

- Logging: [Key events or state changes within this component to log via logger.aiContext/info etc.]

## Code Foundation / Snippets (Optional)

// Starting point, example structure, or related utility function

import React from 'react';

import PropTypes from 'prop-types';

function ComponentName({ prop1, prop2 }) {

// TODO: Implement based on requirements above

return (

<div>

{/\* Component structure \*/}

</div>

);

}

ComponentName.propTypes = {

prop1: PropTypes.string.isRequired,

prop2: PropTypes.func,

};

export default ComponentName;

## French Terminology (Specific to this component)

- UI Labels: [Exact French text for buttons, headers within this component]

- Tooltips: [Any tooltips specific to elements in this component]

- Error messages: [Error messages originating from this component]

## 7.7 Integration with Logging System

The AI collaboration workflow integrates with the logging system [B6.4.1] to enhance debugging and problem-solving:

### 7.7.1 AI-Specific Logging

* Use logger.aiContext(...) for logs specifically intended to provide context for AI review during debugging.
* Include structured metadata (component, operation, state variables).

### 7.7.2 Log Analysis Process

* **Extract Relevant Logs**: Use the log extraction utility (or manually filter log files) based on the component/operation identified as problematic (often noted in the previous Session Handoff & Initialization Plan).
* **Generate Log Summary**: Focus on ERROR, WARN, and AI\_CONTEXT logs around the time of the issue.
* **Incorporate into AI Prompts**: Include concise log excerpts or summaries when asking for debugging help, providing context alongside code snippets. Reference the Handoff/Plan doc if the issue relates to a previously identified challenge.

### 7.7.3 Debugging with AI Assistance

* **Identify Issue**: Use logs and application behavior to pinpoint the problem.
* **Create AI Debugging Prompt**: Provide relevant code snippet, log excerpts/summary, expected vs. actual behavior, and reference the relevant task/goal from the current session's Handoff/Plan document.
* **Implement and Verify Fix**: Use AI suggestions (Cursor/Claude) to implement fixes. Add logging to verify. Document the resolution in the next Handoff/Plan document under 'Challenges & Solutions'.

## 7.8 Token Optimization Techniques

### 7.8.1 Text Optimization

* Use concise formats (lists, tables). Be specific. Remove redundancy. Use consistent notation [BX.Y].

### 7.8.2 Context Prioritization in Handoff/Plan

* **Must-Include**: The full recap of the previous session (Section 3) and the detailed plan for the next session (Section 4) are essential. Specific tasks and deliverables must be clear.
* **Reference-Only (Bible)**: General principles, detailed background, full calculation logic, alternative approaches. Only include Bible excerpts directly in the Handoff/Plan or Component Context docs when absolutely necessary for the immediate next session's tasks. Explicitly state when a full section (e.g., [B5.3]) needs to be consulted, assuming the human can provide it if needed.

### 7.8.3 Visual Context Enhancement

* Use screenshots, diagrams (links in documents) effectively, especially in Component Context docs, to reduce lengthy text descriptions.

## 7.9 First Session Implementation

For the very first development session (e.g., Milestone 0, Session 1):

* **Prepare Initial Plan**: Since there's no previous session, the human developer creates an initial Session Handoff & Initialization Plan document where Section 3 ("Recap") is minimal/N/A, and Section 4 ("Plan") details the goals, tasks, resources, etc. for Session 1.
* **Establish Workflow Pattern**: Use the templates [B7.6]. The first session should conclude with collaboratively generating the Handoff/Plan document for Session 2, setting the pattern for future sessions. Review workflow effectiveness after the first couple of sessions.

## 7.10 Early Workflow Testing Procedures (Milestone 1 Start)

**Objective**: To validate and refine the Human-AI collaboration workflow, communication patterns, logging utility, and handoff process during the initial coding tasks of Milestone 1.

**Procedure**:

1. **Task Assignment**: The Human Developer (HD) selects 1-2 simple, representative coding tasks from the Milestone 1 plan (as designed in Session #1, Task 5).
2. **Initiation**: HD provides the task requirements, relevant Bible sections ([BX.Y]), and any specific context (e.g., code snippets to modify, expected output) to the AI Collaborator (AIC).
3. **Collaborative Development**:
   * AIC assists with code generation, referencing Bible standards (style guides, component structures, logging implementation).
   * HD and AIC engage in dialogue to clarify requirements, discuss implementation choices, and review generated code.
   * Emphasis on using the defined logging (logger.xxx) for key steps and potential error paths within the task scope.
4. **Code Review & Integration**: HD reviews the final code snippet(s), integrates them into the local codebase, and performs basic functional checks.
5. **Log Review**: HD accesses the generated logs (via console or backend extraction mechanism as per [B6.4.2]) related to the completed task. HD assesses the clarity, usefulness, and structure of the logs for debugging or understanding the execution flow.
6. **Handoff Simulation**: HD and AIC collaboratively generate the Session Handoff document for this mini-session, focusing on accurately capturing achievements, challenges (related to workflow or code), and planning for a hypothetical next task.
7. **Debrief & Refinement**: Immediately following the test, HD analyzes the process using the evaluation guidelines ([B7.11]) and documents observations, pain points, and suggested refinements to the workflow, communication, logging, or handoff process. These refinements are incorporated into the Bible ([B7]) and applied going forward.

## 7.11 Workflow Evaluation Guidelines

**Objective**: To systematically evaluate the effectiveness of the Human-AI collaboration workflow after specific testing phases and periodically throughout the project.

**Evaluation Criteria**:

1. **Efficiency**: Was the task completed within a reasonable timeframe compared to estimates? Were there significant delays caused by workflow friction?
2. **Clarity of Communication**: Were instructions from HD clear? Was AIC's understanding accurately reflected in responses and generated outputs? Was the dialogue concise and effective?
3. **Accuracy & Relevance of AI Assistance**: Did the AIC provide accurate code snippets, relevant documentation references, and helpful suggestions aligned with project standards (Bible)?
4. **Usefulness of Logs**: Did the logs generated during the task provide sufficient context for debugging or understanding the process? Was the log extraction/viewing process practical? ([B6.4.2])
5. **Handoff Document Effectiveness**: Did the generated handoff document accurately reflect the session's progress, challenges, and next steps? Was it easy to generate and use for context continuity?
6. **Task Completion Quality**: Was the final output of the task(s) functionally correct and aligned with requirements and Bible standards?

**Evaluation Method & Frequency**:

* **Method**: A combination of:
  + **Post-Session Debrief**: HD explicitly reviews the session against the criteria above, noting strengths and weaknesses. (Can be a structured prompt/response with AIC).
  + **Output Review**: Assessing the quality and completeness of deliverables (code, documentation) against the session plan.
  + **Process Analysis**: Identifying bottlenecks or points of confusion in the interaction flow.
* **Frequency**:
  + After the dedicated Early Workflow Testing ([B7.10]).
  + Briefly during the generation of each Session Handoff document (Section 3 - Challenges, Section 4 - Plan adjustments).
  + More formally at the end of each Milestone.

## 7.12 Milestone Review and Adjustment Process

**Objective**: To ensure milestone timelines remain realistic and achievable by incorporating regular review and adaptation based on progress and challenges.

**Process**:

* **Timing**: Performed during the generation of each Session Handoff document, specifically documented in Section 4.6 ('Milestone Review & Adjustment').
* **Input**:
  + Achievements and Challenges from the completed session (Handoff Section 3).
  + Original Milestone Plan/Tasks.
  + Session plan for the completed session (Previous Handoff Section 4).
* **Review Steps** (Led by HD, Assisted by AIC):
  1. **Compare**: Assess completed tasks against the session plan. Were tasks completed, partially completed, or blocked?
  2. **Analyze**: Identify the root cause of any deviations (e.g., underestimation, unexpected complexity, external blocker, workflow issue).
  3. **Re-estimate**: Briefly re-evaluate the effort required for remaining tasks within the current milestone based on new information.
  4. **Adjust**: If deviations significantly impact the milestone end date or scope:
     + Modify task estimates or sequence.
     + Adjust the overall milestone timeline.
     + Consider scope adjustments (deferring non-critical tasks) if the timeline is fixed.
     + Document any required follow-up actions (e.g., need more info on X).
* **Output**: The agreed-upon assessment and adjustments are recorded in Section 4.6 of the new Session Handoff document being generated. This ensures the updated status is the basis for the next session's planning.

## 7.13 Milestone 1 Workflow Test Plan

### 7.13.1 Objective

To validate and refine the Human-AI collaboration workflow ([B7]), the logging implementation ([B6.4.2]), component structure standards ([B6.5]), and the session handoff process during the initial coding phase of Milestone 1. This test provides a low-risk environment to identify and address process issues before tackling more complex features.

### 7.13.2 Scope

* Execute 2-3 simple, representative coding tasks using the established React project structure.
* Intentionally trigger and handle predefined error scenarios.
* Utilize the defined logging service (logger.js) extensively.
* Simulate the full collaboration loop, including task assignment, code generation/review, logging analysis, and session handoff.
* Track and document time spent on each scenario to refine future estimates.

### 7.13.3 Test Scenarios

#### 7.13.3.1 Scenario 1: Basic Presentational Component

**Task**: Create a simple, reusable InfoCard component within /src/components/ui/.

**Input**: Component should accept title (string), message (string), and variant ('info', 'warning', 'error') props.

**Output**: The component should render a styled card (using a basic div structure) with the title and message. The background color or border color should change based on the variant prop, using colors defined in the UI Style Guide ([B3.8.1] - e.g., Primary Light for info, Bootstrap Yellow/Orange for warning, Bootstrap Red for error).

**Logging**:

* Log (logger.debug) when the component mounts, including its props.
* Log (logger.warn) if an invalid variant prop is received.

**Accessibility Requirements**:

* Use semantic HTML structure with appropriate heading levels for the title.
* Ensure color contrast meets WCAG AA standards for all variant states.
* Include appropriate ARIA attributes if the card contains interactive elements.

**Success Criteria**:

* Component renders correctly with all three variant styles.
* Debug logs are generated on mount with correct component and operation metadata.
* Warning logs are generated when invalid variant props are provided.
* HTML structure passes basic semantic validation.

**Time Allocation**: 30-45 minutes

**Deliverables**: Code, screenshot of each variant state, console logs

#### 7.13.3.2 Scenario 2: Data Display and Logging

**Task**: Create a CategoryDisplay component within /src/features/categories/.

**Input**: Component receives a hardcoded array of simple category objects: [{ id: 'cat1', name: 'Épicerie', color: '#198754' }, { id: 'cat2', name: 'Transport', color: '#0DCAF0' }].

**Output**: The component should render an unordered list (<ul>) where each list item (<li>) displays the category name, preceded by a small colored circle/dot using the category's color property.

**Logging**:

* Log (logger.info) when the component starts processing the category list, including the number of categories received.
* Log (logger.debug) for each category being rendered, including its name and color.

**Error Scenario (Integrated)**: Modify the hardcoded data source temporarily to include an item with name: null. The component should gracefully handle this (e.g., display 'Unnamed Category' or skip the item) and log an error (logger.error) indicating the invalid data encountered, including the problematic category object structure.

**Accessibility Requirements**:

* Use semantic list structure with appropriate labeling.
* Ensure color indicators have sufficient size and are not the only means of identification.

**Success Criteria**:

* Component correctly renders the list with color indicators.
* All logging levels (info, debug, error) work as expected with appropriate metadata.
* Component gracefully handles the null name case with proper error logging.
* List structure follows semantic HTML best practices.

**Time Allocation**: 30-45 minutes

**Deliverables**: Code, screenshot of rendered list including the error case handling, console logs

#### 7.13.3.3 Scenario 3: Basic Interaction and AI Context Logging

**Task**: Create a very simple ActionConfirm component within /src/components/ui/.

**Input**: Component accepts an actionName (string, e.g., "Delete Transaction") and an onConfirm (function) prop.

**Output**: Renders the actionName text and a "Confirm" button.

**Interaction**: When the "Confirm" button is clicked:

* Call the onConfirm function passed in props.
* Log the confirmation action using logger.aiContext.

**Logging**:

* Log (logger.aiContext) when the confirm button is clicked, clearly stating the actionName being confirmed and including any relevant metadata passed (e.g., itemId: 'tx123' if provided via props).

**Accessibility Requirements**:

* Button should be keyboard accessible.
* Include appropriate ARIA attributes for the confirmation action.
* Button text should clearly indicate the action being taken.

**Success Criteria**:

* Button click successfully invokes the provided callback function.
* AI Context logs are generated with the correct structure and metadata.
* Component is usable with keyboard navigation.
* Button follows UI Style Guide appearance standards.

**Time Allocation**: 30-45 minutes

**Deliverables**: Code, screenshot of component, console logs with AI Context entry

### 7.13.4 Execution Procedure

Follow the steps outlined in [B7.10] Early Workflow Testing Procedures:

1. HD assigns one scenario at a time to AIC.
2. Collaborative development occurs (code generation, discussion).
3. HD integrates and performs functional checks.
4. After each scenario, HD creates a Git commit with a descriptive message mentioning the scenario completed.
5. HD captures screenshots of the component in various states.
6. HD reviews logs generated by the task (via console initially, then simulates extraction from backend file source as per [B6.4.2.6]).
7. Simulate session handoff after completing the scenarios.
8. Conduct debrief using evaluation guidelines ([B7.11]).

### 7.13.5 Evaluation Criteria

Evaluate the execution of these scenarios using the criteria defined in [B7.11] Workflow Evaluation Guidelines:

* **Efficiency**: Compare actual time spent to allocated time for each scenario.
* **Clarity of Communication**: Assess whether requirements were clearly understood and implemented correctly.
* **Accuracy & Relevance of AI Assistance**: Evaluate code quality, adherence to project standards.
* **Usefulness of Logs**: Assess log structure, content richness, and extraction process.
* **Handoff Document Effectiveness**: Determine if the handoff document accurately captures progress and challenges.
* **Task Completion Quality**: Verify that all success criteria for each scenario are met.

### 7.13.6 Refinement Process

Based on the evaluation outcomes:

* Identify specific areas for improvement in the workflow, logging practices, documentation (Bible sections B3, B6, B7), or component standards.
* Document these proposed refinements with concrete examples of what worked and what didn't.
* Create a prioritized list of Bible sections that need updating based on findings.
* Update the relevant Bible sections accordingly before proceeding further into Milestone 1.
* Adjust Milestone 1 task estimates or plans if the test reveals significant underestimations or complexities ([B7.12]).
* Document any new best practices discovered during the test for future reference.

### 7.13.7 Version Control Integration

To ensure the test process integrates with our version control strategy:

* Create a dedicated Git branch for the workflow test (e.g., milestone1-workflow-test).
* Commit each scenario implementation separately with descriptive commit messages.
* Include the scenario number and name in each commit message.
* Reference relevant Bible sections in commit messages where appropriate.
* After successful completion and evaluation, consider whether to merge these components into the main development branch or treat them purely as test components.

# SECTION 8: HEALTHY WORK PRACTICES GUIDELINES

## 8.1 Guiding Principle

Prioritize sustainable development pace, focus, and well-being to avoid burnout, maintain high-quality output, and ensure effective learning/review of AI-generated code. Recognizing that AI tools can accelerate development but may also create pressure to work longer hours, these guidelines establish boundaries for a healthy and sustainable workflow.

## 8.2 Agreed Practices

* **Strict Work Window**: Development work will be strictly limited to the hours between 20:00 and 22:00 [Your Timezone - e.g., EDT] daily. Work outside this window should be avoided to maintain focus and balance.
* **Breaks**: Take short breaks (e.g., 10-15 minutes away from the screen) if needed during the 2-hour work window.
* **End-of-Session Review**: Dedicate the last 15-30 minutes of each session to review progress, commit code, and generate the session summary/plan for next time. Avoid ending abruptly mid-task.
* **Stop When Fatigued**: Even within the defined work window, consciously stop work when feeling overly tired, frustrated, or unproductive. It's better to end early than to push through and make mistakes that require more time to fix later.
* **Physical Activity**: Incorporate some physical activity or time completely away from the computer screen before and/or after the development session.
* **Context Switching**: If possible, establish a brief transition ritual before and after the work session to help mentally separate development time from other activities.
* **Weekend Policy**: Decide in advance whether weekend sessions will follow the same schedule or if weekends will be development-free. Consistency helps establish the routine.
* **Task Selection**: Match task complexity to energy levels. Save complex problem-solving for times when you're most alert within your work window.

## 8.3 AI Role

AI assistants can help by:

* Structuring tasks to fit within the defined 2-hour work blocks during session planning
* Providing reminders when prompted by the user during a session
* Helping plan session goals realistically based on time estimates
* Assisting with summarizing progress at the end of sessions
* Breaking down complex tasks into smaller units that fit comfortably within the time constraints

(Acknowledgement: AI cannot enforce these; discipline rests with the user.)

## 8.4 Progress Tracking & Satisfaction

To balance progress with healthy work habits:

* Set realistic expectations for each milestone based on the 2-hour daily window
* Celebrate small wins and completed components
* Focus on quality of implementation rather than speed
* Document progress visually (e.g., checked-off tasks, screenshots of working features)
* Recognize that consistent, sustainable progress is more valuable than sporadic intense work sessions

## 8.5 Preventing Context Loss Between Sessions

Since sessions are time-boxed to 2 hours daily, maintaining context between sessions is crucial:

* End each session with a clear, detailed summary
* Begin the next session by reviewing the previous summary
* Use descriptive commit messages that future-you will understand
* Comment code-in-progress with // TODO notes for the next session
* Take brief notes about any "aha moments" or ideas that emerge toward the end of a session

By adhering to these practices, the development process will remain enjoyable, sustainable, and aligned with overall well-being while still making consistent progress toward project completion.

# SECTION 9: COMMERCIAL CONSIDERATIONS & COST ANALYSIS

## 9.1 Potential Commercial Paths

The primary path envisioned is a Freemium model. A core set of budgeting features (aligned with the MVP) will be offered for free. Advanced features developed post-MVP (e.g., Shared Budgets, Goal-Based Savings, Advanced Analytics/Reporting, potentially Notifications) will be part of a paid premium subscription tier (monthly/annual).

Advertising will not be used as a monetization method, as it would compromise the clean user experience and potentially create conflicts of interest in a financial management application.

## 9.2 MVP Stack Cost Analysis

| **Component** | **Free Tier Limits** | **MVP Assessment** |
| --- | --- | --- |
| **Firebase Authentication** | Free for email/password, phone auth (limits apply but usually high), limited free social logins. | Sufficient for MVP. |
| **Firestore** | Generous free tier for reads, writes, storage (e.g., 50k reads/day, 20k writes/day, 1 GiB storage). | Highly likely sufficient for development and initial MVP launch for 2 users. Usage needs monitoring. |
| **Hosting** | Generous free tier for storage and data transfer (e.g., 10 GB storage, 360 MB/day transfer). | Sufficient for MVP. |
| **Cloud Functions** | Free tier includes a significant number of invocations, compute time, and network egress (e.g., 2M invocations/month). | Sufficient for MVP, especially since most logic is frontend/Firestore direct. Scheduled functions for recurrence need monitoring. |

**Conclusion**: The core Firebase stack for the MVP can realistically be operated within the free tier limits during development and initial use.

**AI Tools (Claude/Cursor)**: Costs depend entirely on the user's specific subscription plans and usage intensity. These are development costs, not operational costs of the deployed app. User should monitor their usage against their plan limits.

**Other**: Potential small annual cost for a custom domain name if desired for Firebase Hosting (optional).

## 9.3 Future Scaling Costs

| **Component** | **Post-MVP Scaling Considerations** |
| --- | --- |
| **Firebase (Blaze Plan - Pay-as-you-go)** | If app usage exceeds free tier limits (due to user growth or feature intensity), transitioning to the Blaze plan will be necessary. Costs are based on consumption (per read/write, per GB stored, per function invocation second, etc.). Pricing is generally considered competitive for cloud services. Firebase Pricing should be consulted for details. |
| **Payment Gateway** | Implementing the Freemium model (Post-MVP) will require integrating a payment processor like Stripe or PayPal. These services typically charge a percentage + fixed fee per transaction (e.g., 2.9% + $0.30 per transaction). |
| **Other Services** | Future features might require additional paid services (e.g., more advanced analytics, email services for notifications, additional cloud functions for shared budgets). |

### Estimated Monthly Costs for Post-MVP Commercial Deployment:

| **User Base Size** | **Estimated Monthly Cost Range** |
| --- | --- |
| Small (< 1,000 active users) | $20-50/month |
| Medium (1,000-10,000 active users) | $50-300/month |
| Large (> 10,000 active users) | Custom pricing analysis required |

These estimates assume a typical usage pattern and would need to be refined based on actual metrics once the application is in use.

### Specific Usage Thresholds for Blaze Plan Upgrade

| **Firebase Service** | **Free Tier Limit** | **Upgrade Trigger** |
| --- | --- | --- |
| Firestore Reads | 50,000/day | >40,000/day consistently (80% of limit) |
| Firestore Writes | 20,000/day | >16,000/day consistently (80% of limit) |
| Firestore Storage | 1 GiB | >800 MiB (80% of limit) |
| Cloud Functions Invocations | 2M/month | >1.6M/month (80% of limit) |
| Cloud Functions Compute Time | 400,000 GB-s/month | >320,000 GB-s/month (80% of limit) |
| Hosting Storage | 10 GB | >8 GB (80% of limit) |
| Hosting Transfer | 360 MB/day | >288 MB/day consistently (80% of limit) |

It's recommended to set up Firebase budget alerts at 50%, 70%, and 90% of anticipated monthly spending limits to avoid unexpected costs when transitioning to the Blaze plan.

# SECTION 10: COMPETITIVE ANALYSIS & DIFFERENTIATION (POST-MVP PLANNING)

## 10.1 Target Competitors

### Personal Finance Apps:

* **You Need A Budget (YNAB)**: Premium subscription-based budgeting app with a strong zero-based budgeting philosophy
* **Mint**: Free budgeting app with ad-supported revenue model, bank account connections, and basic budgeting features
* **Personal Capital**: Free wealth management platform with paid financial advisory services

### Expense Sharing Apps:

* **Splitwise**: Focused on expense sharing and IOUs between friends, roommates, and travel groups
* **Tricount**: Simple expense splitting with group tallying
* **Shareroo**: Newer app featuring both budget management and expense sharing capabilities

### Banking Apps with Budgeting Features:

* Many commercial banks now offer integrated budgeting features within their mobile apps
* Features typically include spending analysis, basic category tracking, and savings goals

## 10.2 Key Feature Comparison

### Zero-Based Budgeting Implementation:

* **YNAB**: Strong implementation with "give every dollar a job" philosophy, but steep learning curve
* **Most other apps**: Basic category budgeting without the discipline of fully allocating available funds
* **KarmaCash**: Zero-based budgeting with a focus on ease-of-use and visual feedback on allocation progress

### Expense/Income Tracking:

* **Most competitors**: Handle basic expense tracking with categories
  + Many lack robust recurring transaction handling
  + Few handle income/expense within the same category (like our Santé reimbursement feature)
* **KarmaCash**: Comprehensive transaction tracking with proper handling of both expense and income within same categories, clear recurring transaction support

### Budget Rollovers:

* **YNAB**: Handles category rollovers, but with a specific philosophy
* **Most other apps**: Limited or no month-to-month rollover tracking
* **KarmaCash**: Precise rollover calculations including handling negative rollovers properly

### UI/UX Approach:

* **YNAB**: Feature-rich but potentially overwhelming for new users
* **Mint**: Clean UI but ad-supported, leading to clutter
* **Splitwise**: Focused on single use case with minimal learning curve
* **KarmaCash**: Clean, focused interface with progressive disclosure of complexity, French language support (fr-CA)

### Sharing/Collaboration:

* **Splitwise/Tricount**: Strong sharing features but minimal budgeting
* **YNAB**: Limited sharing features (primarily for couples)
* **KarmaCash** (post-MVP): Comprehensive shared budgeting with robust permissions

## 10.3 Our Unique Value Proposition

### Core Differentiators:

1. **Simplified Zero-Based Budgeting**: Making a powerful budgeting methodology accessible without overwhelming complexity
2. **Proper Financial Math**: Getting calculations right, especially rollovers and category net amounts (expense minus income per category)
3. **Clear Visual Feedback**: Making progress, status, and financial health immediately visible
4. **French-First Interface**: Designed primarily for French-speaking users (fr-CA) rather than being a translation afterthought
5. **Fair Pricing Model**: Freemium approach without ads, with clear value in premium features

### Post-MVP Differentiators:

1. **Thoughtful Sharing**: Budget sharing designed from a realistic household perspective
2. **Progressive Complexity**: A layered approach where basic features are immediately accessible, while advanced features can be gradually adopted
3. **PWA Accessibility**: Full functionality across devices without the need for multiple native app installations

## 10.4 Influence on Post-MVP Roadmap

Based on this competitive analysis, we can identify key strategic opportunities for post-MVP development:

### High Priority Features (Immediate Post-MVP):

1. **Goal-Based Savings**: A feature present in several competitors that users value highly
2. **Basic Sharing**: Essential functionality for couples and households
3. **Future Projections**: Help users visualize future financial state based on current patterns

### Medium Priority Features:

1. **Data Export/Reports**: Functionality that adds significant value for financially-minded users
2. **Enhanced Visualization**: More advanced charts and analysis tools
3. **Custom Periods**: Beyond weekly/monthly views to custom date ranges

### Lower Priority Features:

1. **Bank Connections**: Would require significant security investment and regulatory compliance
2. **Investment Tracking**: Already well-served by dedicated platforms
3. **Receipt Scanning**: Nice-to-have but not core to the budgeting value proposition

This strategic prioritization ensures that post-MVP development focuses on features that enhance our core value proposition rather than simply matching competitors feature-for-feature.

## 10.5 Competitive Feature Matrix

| **Feature** | **KarmaCash (MVP)** | **KarmaCash (Post-MVP)** | **YNAB** | **Mint** | **Splitwise** |
| --- | --- | --- | --- | --- | --- |
| **Zero-Based Budgeting** | ✅ Simple, intuitive | ✅ Enhanced | ✅ Complex | ❌ | ❌ |
| **Recurring Transactions** | ✅ | ✅ Enhanced | ✅ | ✅ Limited | ❌ |
| **Budget Rollovers** | ✅ | ✅ Enhanced | ✅ Own system | ❌ | ❌ |
| **French-First Interface** | ✅ | ✅ | ❌ | ❌ | ❌ |
| **Ad-Free Experience** | ✅ | ✅ | ✅ | ❌ | ✅ |
| **Budget Sharing** | ❌ | ✅ | ✅ Limited | ❌ | ✅ |
| **Goal-Based Savings** | ❌ | ✅ | ✅ | ✅ | ❌ |
| **Bank Connections** | ❌ | ❌ | ✅ | ✅ | ❌ |
| **Investment Tracking** | ❌ | ❌ | ❌ | ✅ | ❌ |
| **PWA Support** | ✅ | ✅ Enhanced | ❌ | ❌ | ❌ |
| **Data Export** | ❌ | ✅ | ✅ | ✅ | ✅ |
| **Receipt Scanning** | ❌ | ❓ (Potential) | ❌ | ❌ | ❌ |
| **Custom Categories** | ✅ | ✅ | ✅ | ✅ | ✅ |
| **Cost** | Free | Freemium | Subscription only | Free (with ads) | Free/Premium |

## 10.6 Target Audience Comparison

Understanding how our target audience differs from our competitors' helps focus our differentiation strategy:

| **App** | **Primary Target Audience** | **KarmaCash Differentiation** |
| --- | --- | --- |
| **YNAB** | Financial enthusiasts, budget power users who want disciplined control | Simpler, more approachable zero-based budgeting with French-language focus |
| **Mint** | Casual budgeters who want passive tracking and insights | Ad-free, privacy-focused alternative with more active budgeting control |
| **Personal Capital** | Investment-focused users, wealth management | Pure budgeting focus without investment complexity |
| **Splitwise** | Groups needing to split expenses (roommates, travelers) | Combines expense splitting with comprehensive budgeting (post-MVP) |
| **Banking Apps** | Existing bank customers seeking convenience | Specialized budgeting tools with cross-account perspective |

KarmaCash's primary audience differentiator is the focus on French-speaking users who want simple but effective zero-based budgeting without the complexity of YNAB or the privacy compromises of ad-supported apps.

## 10.7 User Experience Competitive Analysis

Our Zen/Tranquility design philosophy ([B3.4]) provides distinct advantages in the competitive landscape:

| **Experience Element** | **KarmaCash Approach** | **Competitive Analysis** |
| --- | --- | --- |
| **Visual Clarity** | Clean, focused screens with minimal distractions | YNAB: Dense, information-heavy  Mint: Cluttered with ads and promotions |
| **Color Psychology** | Soothing color palette with meaningful, consistent use of color for financial status | Most competitors use alarming reds for overspending without nuance |
| **First-Run Experience** | Progressive disclosure of features, gentle onboarding | YNAB: Steep initial learning curve  Most apps: Information overload |
| **Error Handling** | Calm, constructive guidance using the Zen approach | Typical error messages that create stress or confusion |
| **Financial Stress** | Design specifically acknowledges and mitigates financial anxiety | Few competitors explicitly design for emotional aspects of money management |

## 10.8 Technical Differentiation

While most of our differentiation is user-experience focused, there are several technical advantages in our implementation:

| **Technical Aspect** | **KarmaCash Approach** | **Competitive Advantage** |
| --- | --- | --- |
| **Offline Support** | PWA with robust offline capability | Many competitor web apps have limited offline functionality |
| **Performance** | Lightweight, focused codebase | Some competitors (particularly bank apps) have performance issues |
| **Recurring Logic** | Sophisticated transaction pattern generation | More accurate and flexible than many competitors' implementations |
| **Updates** | Web-based for immediate updates | Native apps require app store approvals and user updates |
| **Cross-Device** | True responsive design with consistent experience | Some competitors have feature disparities between platforms |

## 10.9 Post-MVP Competitive Strategy

As we move beyond MVP, our competitive strategy should emphasize:

1. **Double Down on Strengths**: Further refine our core differentiators (zero-based simplicity, French-first, calming UX)
2. **Strategic Feature Adoption**: Implement the high-priority features identified in [10.4] to address key competitive gaps
3. **Feedback-Driven Refinement**: Use early user feedback to identify which competitor features are truly missed vs. which differentiators are most valued
4. **Community Building**: French-language user community and support materials to strengthen our language-based differentiation
5. **Avoid Feature Bloat**: Resist the temptation to match every competitor feature; maintain our focus on simplicity and calm

By maintaining this focus on our unique strengths while selectively implementing high-value features from competitors, we can develop a distinct and compelling offering in the personal finance application market.