YAML

```yaml
title: 'Analyse Approfondie du Projet : Tontine Éthique'
author: 'Manus AI'
date: '30 janvier 2026'
```

Analyse Approfondie du Projet : Tontine Éthique

1. Introduction

Ce document présente une analyse détaillée du projet **Tontine Éthique**, basée exclusivement sur l'exploration du code source de la version disponible. L'objectif est de fournir une compréhension claire de l'architecture réelle, des flux de données, d'identifier les incohérences ou les zones incomplètes, et de dresser une liste d'actions prioritaires pour stabiliser l'application en vue d'une mise en production.

Aucune modification n'a été apportée au code. L'analyse se fonde sur les fichiers présents dans le dépôt GitHub au moment de l'examen.

2. Architecture Générale et Flux de Données

L'application est développée en **Flutter** et s'appuie sur une architecture **multi-plateforme** (mobile, web, desktop). Le backend est principalement géré par **Firebase** (Firestore, Authentication, Cloud Functions) avec une intégration poussée de **Stripe** pour les paiements et les abonnements.

Le flux de données principal est réactif et géré par le package `flutter_riverpod`. L'état de l'utilisateur et les données métier (cercles, transactions) sont synchronisés en temps réel avec la base de données Firestore.

Carte du Fonctionnement de l'Application

mermaid

Composant	Technologie	Rôle principal
Frontend	Flutter	Interface utilisateur, gestion de l'état local
Gestion d'état	Riverpod	Flux de données réactifs, injection de dépendances
Base de données	Firestore	Stockage des données utilisateur, tontines, transactions
Authentification	Firebase Auth	Connexion (Email, Google, Téléphone)
Paiements & Abos	Stripe	Gestion des plans payants, mandats SEPA, comptes Connect
Paiements (Afrique)	APIs (Wave, OM)	Intégration des paiements par mobile money (simulée)
Logique Backend	Firebase Functions	Tâches planifiées (CRON), webhooks, opérations sécurisées

---

3. Analyse par Module

3.1. Authentification (`/features/auth`)

- **À quoi ça sert ?** Gère l'inscription et la connexion des utilisateurs via Email/Mot de passe, Google, et Numéro de téléphone (avec vérification OTP). Le flux est séparé pour les particuliers et les entreprises.
- **Connexion à Firebase : Oui, entièrement.** Le service `AuthService` communique directement avec `FirebaseAuth` pour toutes les opérations. La création de l'utilisateur dans Firestore est également gérée à ce niveau.
- **Fonctionnel ou Mocké ? Fonctionnel.** Le code pour `signInWithEmail` , `signInWithGoogle` , et le flux OTP (`signInWithPhone` et `verifyOtp`) est complet et interagit réellement avec Firebase.

- **Code mort ou utilisé ? Utilisé.** Tous les écrans (`auth_screen` , `registration_screen` , etc.) sont reliés par le routeur et fonctionnels.

3.2. Gestion Utilisateur, Compte & Abonnements (`/core/providers` , `/features/subscription`)

- **À quoi ça sert ?** Le `user_provider.dart` est le cœur de l'état de l'utilisateur. Il centralise toutes les informations (UID, nom, e-mail, statut, plan d'abonnement, etc.) et les synchronise avec le document utilisateur dans Firestore. Le module d'abonnement gère la sélection des plans et le passage à la caisse via Stripe.
- **Connexion à Firebase : Oui, entièrement.** `UserNotifier` écoute en temps réel les modifications sur le document `users/{uid}` et `entitlements/{uid}` dans Firestore pour mettre à jour l'état de l'application.
- **Fonctionnel ou Mocké ? Partiellement fonctionnel.**
  - La lecture et la mise à jour de l'état de l'utilisateur sont fonctionnelles.
  - La logique de sélection de plan est fonctionnelle et lit les plans depuis Firestore.
  - Le passage au paiement Stripe (`createCheckoutSession`) est **fonctionnel** et appelle une Cloud Function pour créer une session de paiement.
  - La **mise à jour automatique du plan après paiement est fragile**. Elle repose sur un fallback côté client (`PaymentSuccessScreen`) qui écrit directement dans Firestore, en plus d'attendre un webhook. C'est une source potentielle de problèmes de synchronisation.
- **Code mort ou utilisé ? Utilisé.** Le `userProvider` est injecté dans presque tous les écrans de l'application.

3.3. Tontines (`/features/tontine`)

- **À quoi ça sert ?** C'est le module principal de l'application. Il permet de créer, explorer, rejoindre et participer à des cercles de tontine. Il inclut la messagerie de groupe (`circle_chat_screen.dart`).
- **Connexion à Firebase : Oui, entièrement.** `CircleService` et `CircleNotifier` gèrent toutes les opérations CRUD (Create, Read, Update, Delete) sur la collection `tontines` de Firestore. Les messages du chat sont également stockés en temps réel dans une sous-collection.
- **Fonctionnel ou Mocké ? Partiellement fonctionnel.**
  - La création et l'affichage des tontines sont **fonctionnels**.
  - Le chat de groupe (`circle_chat_screen.dart`) est **fonctionnel** et connecté à Firestore pour les messages texte, l'envoi d'images et de fichiers audio (via Firebase Storage).

    - La logique de **rejoindre un cercle** est présente mais le flux d'approbation par le créateur est implémenté via une collection `join_requests`, ce qui est fonctionnel.
    - La logique de **vote pour l'ordre des paiements** est présente dans le code (`PayoutOrderVotingScreen`) mais semble incomplète et non entièrement reliée.
- **Code mort ou utilisé ?** Principalement **utilisé**. Des sections comme le simulateur de tontine sont également présentes et fonctionnelles.

3.4. Paiement (Stripe & Mobile Money)

- **À quoi ça sert ?** Gère tous les flux financiers : abonnements via Stripe, et cotisations/paiements via Stripe Connect ou Mobile Money.
- **Connexion à Firebase : Oui.** Les Cloud Functions agissent comme un intermédiaire sécurisé pour appeler l'API Stripe avec les clés secrètes. Les transactions sont enregistrées dans Firestore.
- **Fonctionnel ou Mocké ? Partiellement fonctionnel et cassé.**
    - **Abonnements Stripe :** Le flux de création de session de paiement (`createCheckoutSession`) est **fonctionnel**. L'utilisateur est redirigé vers la page de paiement Stripe.
    - **Appels Stripe incomplets/cassés :**
        - Le service `StripeService` utilise une URL de Cloud Function en dur (`https://europe-west1-tontetic-admin.cloudfunctions.net`). Si le projet Firebase a un ID différent, **tous les appels échoueront**.
        - La gestion des paiements de tontine (cotisations) via Stripe Connect est initiée (`createConnectAccount`) mais le flux complet de virement entre participants (`Transfer`) n'est pas implémenté dans le code client.
        - La logique de **garantie solidaire** (`executeGuarantee` dans les Cloud Functions) tente de faire un paiement *off-session*, ce qui nécessite des autorisations spécifiques (mandat SEPA, etc.) et est susceptible d'échouer sans un flux d'enrôlement client robuste.
    - **Mobile Money :** Le code dans `mobile_money_service.dart` est une **simulation/façade**. Il prépare des requêtes HTTP vers les API de Wave et Orange Money mais utilise des URL de base et des logiques qui sont probablement des placeholders. Les clés d'API sont chargées depuis un fichier `.env`, mais le service lui-même n'est pas prêt pour la production.
- **Code mort ou utilisé ? Utilisé**, mais avec des parties non fonctionnelles.

3.5. Back-office / Admin & Firebase Functions

- **À quoi ça sert ?** Le dossier `admin_backoffice` contient une interface web statique (probablement pour l'administration). Les `functions/index.js` contiennent la logique backend critique.
- **Connexion à Firebase : Oui.** Les fonctions interagissent directement avec Firestore et l'API Stripe.
- **Fonctionnel ou Mocké ? Fonctionnel mais avec des risques.**
  - Les fonctions CRON (`calculateHonorScores` , `monitorGuarantees`) sont écrites et se déploieraient, mais leur logique métier peut contenir des bugs.
  - Le webhook Stripe (`stripeWebhook`) est présent mais ne gère que l'événement `payment_intent.succeeded` . D'autres événements cruciaux (échecs, remboursements, fin d'abonnement) ne sont pas gérés, ce qui peut laisser la base de données dans un état incohérent.
  - La fonction `executeGuarantee` est dangereuse car elle peut être appelée depuis le client et initier un paiement. Bien qu'elle vérifie l'authentification, les permissions précises (qui a le droit de la déclencher ?) ne sont pas clairement définies.
- **Code mort ou utilisé ? Utilisé.** Ces fonctions sont essentielles au fonctionnement de l'application.

---

4. Analyse des Points Spécifiques

- **`user_provider.dart`**
  - **Rôle :** C'est le **cerveau de l'état utilisateur**. Il gère la synchronisation bidirectionnelle avec Firestore et expose des données et des méthodes à toute l'application.
  - **État : Fonctionnel mais complexe.** La détection de la zone (`detectZoneFromPhone`) est codée en dur pour retourner `zoneEuro` , ce qui désactive toutes les fonctionnalités liées à l'Afrique (Mobile Money). La gestion de l'état `isPremium` est redondante avec `planId` .
- **`dashboard_screen.dart`**
  - **Rôle :** L'écran principal après connexion, affichant un résumé des activités, des tontines et des actualités.
  - **État : Partiellement mocké.** L'écran est fonctionnel dans sa structure, mais plusieurs de ses composants (`ActivityFeed` , `NewsCarousel`) lisent des données depuis des collections Firestore (`activities` , `news`) qui peuvent ne pas être activement gérées, rendant le contenu statique ou vide.
- **`circle_chat_screen.dart`**

    - **Rôle :** L'écran de messagerie pour un cercle de tontine.
    - **État : Fonctionnel.** C'est l'une des fonctionnalités les plus complètes. Elle gère l'envoi/réception de messages texte, images, et audio en temps réel via Firestore et Firebase Storage. Le code est robuste.
- **Flux d'abonnement Stripe**
    - **Rôle :** Permettre aux utilisateurs de souscrire à un plan payant.
    - **État : Fonctionnel mais fragile.** Le flux redirige correctement vers Stripe. Cependant, la confirmation post-paiement (`PaymentSuccessScreen`) est un point faible. Elle tente de mettre à jour le plan de l'utilisateur côté client comme solution de secours si le webhook est lent. En production, cela peut causer des incohérences. **La seule source de vérité devrait être le webhook Stripe.**
- **Deep Links (`/join/:circleId`)**
    - **Rôle :** Permettre à un utilisateur de rejoindre une tontine via un lien d'invitation.
    - **État : Fonctionnel.** Le routeur (`router.dart`) intercepte correctement les liens `/join/:id` . Si l'utilisateur n'est pas connecté, il stocke le lien, le redirige vers la connexion, puis le redirige à nouveau vers l'invitation après connexion. La logique est en place et semble correcte.

---

5. Problèmes Bloquants et Priorités

5.1. Carte des Problèmes

Problème	Module(s) affecté(s)	Gravité	Description
URL de Cloud Function en dur	Paiement (Stripe)	**Bloquant**	Tous les paiements et opérations Stripe échoueront si l'ID du projet Firebase change.
Confirmation de paiement fragile	Abonnements (Stripe)	**Critique**	Risque de non-mise à jour du statut premium de l'utilisateur après paiement.
Logique Mobile Money simulée	Paiement (Afrique)	**Critique**	Les paiements en Afrique ne sont pas fonctionnels. L'application ne peut

			pas opérer dans la zone FCFA.
Webhooks Stripe incomplets	Backend (Functions)	**Élevée**	L'application ne peut pas gérer les échecs de paiement, les fins d'abonnement ou les litiges, menant à un état de base de données incorrect.
Détection de zone désactivée	`user_provider`	**Élevée**	L'application est forcée en mode "Europe", rendant les fonctionnalités africaines inaccessibles.
Données du Dashboard mockées	`dashboard_screen`	**Moyenne**	L'écran d'accueil peut paraître vide ou statique, dégradant l'expérience utilisateur.
Flux de paiement de tontine incomplet	Tontines, Paiement	**Moyenne**	Les utilisateurs peuvent créer des tontines mais pas encore échanger de l'argent de manière sécurisée via Stripe Connect.

5.2. Qu'est-ce qui empêche le déploiement en production ?

1. **Le système de paiement n'est pas fiable.** Entre les URL en dur, les webhooks incomplets et la simulation du Mobile Money, il est impossible de traiter des transactions financières réelles de manière sécurisée et fiable.
2. **La logique métier critique est incomplète.** Le cycle de vie complet d'une tontine (création -> invitation -> paiement des cotisations -> décaissement au gagnant -> clôture) n'est pas entièrement implémenté.
3. **L'application est verrouillée sur la zone Euro.** La désactivation de la détection de la zone FCFA la rend inutilisable pour son marché cible principal présumé.

5.3. Actions de Correction Prioritaires (Ordre Logique)

1. **Fiabiliser le Backend et les Paiements (Priorité absolue) :**

   a. **Externaliser la configuration :** Remplacer l'URL de la Cloud Function en dur par une variable d'environnement.

   b. **Compléter les Webhooks Stripe :** Implémenter la gestion pour `invoice.payment_failed`, `customer.subscription.deleted`, et `charge.dispute.created` afin de gérer tous les cas de figure.

   c. **Rendre la confirmation de paiement robuste :** Supprimer la logique de mise à jour du plan côté client (`PaymentSuccessScreen`). L'application doit **uniquement** se fier aux données mises à jour par le webhook via Firestore.

2. **Activer et Finaliser la Logique Métier :**

   a. **Activer la détection de zone :** Réactiver la détection de la zone FCFA dans `user_provider.dart` pour permettre les tests et le développement des fonctionnalités africaines.

   b. **Implémenter le Mobile Money :** Remplacer le code de simulation par une intégration réelle avec les API de Wave, Orange Money, etc., ou une solution d'agrégation.

   c. **Finaliser le cycle de vie des tontines :** Compléter la logique de paiement des cotisations et de décaissement des gains, en s'appuyant sur Stripe Connect ou le Mobile Money.

3. **Améliorer l'Expérience Utilisateur :**

   a. **Connecter le Dashboard à des données réelles :** Mettre en place la logique backend (ou des services tiers) pour alimenter les flux d'actualités et d'activités.

   b. **Revoir et tester les permissions :** Clarifier qui peut déclencher des actions sensibles comme `executeGuarantee` et renforcer les règles de sécurité Firestore.

En suivant cet ordre, les fondations critiques de l'application seront stabilisées avant de finaliser les fonctionnalités plus visibles, assurant ainsi une montée en charge plus sereine vers la production.