MoodMate Technical Report

Executive Summary

MoodMate is a cross-platform mental wellness application built with Flutter and Firebase, featuring AI-powered mood analysis using Mistral AI. This technical report documents the implementation details, architectural decisions, and solutions to technical challenges encountered during development.

---

1. Development Approach and Workflow

1.1 Development Methodology

The project followed an **iterative incremental development** approach, organized into four main phases:

1. **Phase 1: Foundation** - Project setup, authentication, and user management
2. **Phase 2: Core Features** - Mood tracking and AI integration
3. **Phase 3: Analytics** - History viewing and trend visualization
4. **Phase 4: Counsellor Features** - Support system and messaging

1.2 Project Structure

The Flutter project follows a feature-based architecture with clear separation of concerns:

```
lib/
├── main.dart                 # Application entry point
├── firebase_options.dart     # Firebase configuration
├── models/                   # Data models
│   ├── user_model.dart
│   ├── mood_entry_model.dart
│   ├── message_model.dart
│   ├── counsellor_model.dart
│   ├── support_request_model.dart
│   └── counsellor_assignment_model.dart
├── providers/                # State management
│   └── auth_provider.dart
├── services/                 # Business logic & Firebase operations
│   ├── auth_service.dart
│   ├── mood_entry_service.dart
│   ├── mood_analysis_service.dart
│   ├── message_service.dart
│   ├── counsellor_service.dart
│   ├── support_request_service.dart
│   ├── counsellor_assignment_service.dart
│   └── fcm_service.dart
├── screens/                  # UI screens
│   ├── auth/
│   ├── home/
│   ├── mood/
│   └── counsellor/
├── widgets/                  # Reusable components
└── utils/                    # Helper utilities
```

1.3 Version Control Workflow

- **Git** for version control
- Feature branches for new functionality
- Commit messages following conventional commit standards

---

2. Software, Tools, and Frameworks Used

2.1 Frontend Framework

Technology	Version	Purpose
Flutter	3.x	Cross-platform UI framework
Dart	^3.10.4	Programming language

2.2 Backend Services (Firebase)

Service	Purpose
Firebase Authentication	User authentication and session management
Cloud Firestore	NoSQL document database
Firebase Cloud Functions	Serverless backend for AI integration
Firebase Cloud Messaging (FCM)	Push notifications

2.3 Key Dependencies

```
dependencies:
  # Firebase Suite
  firebase_core: ^3.8.1
  firebase_auth: ^5.3.4
  cloud_firestore: ^5.5.2
  cloud_functions: ^5.2.1
  firebase_messaging: ^15.1.5
  # State Management
  provider: ^6.1.2

  # UI & Visualization
  fl_chart: ^0.69.2
  google_fonts: ^6.2.1

  # Utilities
  email_validator: ^3.0.0
  intl: ^0.19.0
```

2.4 Cloud Functions (Backend)

Technology	Purpose
Node.js/TypeScript	Cloud Functions runtime
Mistral AI SDK	AI-powered mood analysis
Firebase Admin SDK	Server-side Firestore access

2.5 Development Tools

- **VS Code** - Primary IDE
- **Android Studio** - Android emulator and build tools
- **Xcode** - iOS simulator and build tools
- **Firebase CLI** - Deployment and emulation
- **FlutterFire CLI** - Firebase configuration

---

3. Database Design and Data Handling

3.1 Firestore Collections Schema

Users Collection (`/users/{userId}`)

```
{
  name: string,           // User's display name
  email: string,          // Email address
  role: 'user' | 'counsellor' | 'admin',
  createdAt: Timestamp,
  updatedAt: Timestamp,
  emailVerified: boolean,
  fcmToken?: string       // Push notification token
}
```

Mood Entries Collection (`/mood_entries/{entryId}`)

```
{
  userId: string,              // Reference to user
  text: string,                // Journal entry text
  date: Timestamp,             // Entry date (normalized to day)
  timestamp: Timestamp,        // Exact creation time
  emotion?: string,            // AI-detected emotion
  confidenceScore?: number,    // AI confidence (0-1)
  recommendations?: string[],  // AI-generated suggestions
  analyzedAt?: Timestamp,      // Analysis completion time
  analysisStatus: 'pending' | 'completed' | 'failed'
}
```

Support Requests Collection (`/support_requests/{requestId}`)

```
{
  userId: string,           // User requesting support
  counsellorId: string,     // Assigned counsellor
  status: 'pending' | 'accepted' | 'inProgress' | 'completed' | 'rejected',
  conversationThreadId: string,
  createdAt: Timestamp,
  updatedAt: Timestamp
}
```

Conversation Threads (`/conversation_threads/{threadId}`)

```
{
  userId: string,
  counsellorId: string,
  supportRequestId: string,
  createdAt: Timestamp,
  lastMessageAt: Timestamp
}
```

Messages Subcollection (`/conversation_threads/{threadId}/messages/{messageId}`)

```
{
  conversationThreadId: string,
  senderId: string,
  receiverId: string,
  content: string,
  timestamp: Timestamp,
  isRead: boolean,
  readAt?: Timestamp
}
```

3.2 Data Model Implementation

Each Firestore collection has a corresponding Dart model class with serialization methods:

```
class MoodEntry {
  final String id;
```

```dart
final String userId;
final String text;
final DateTime date;
final DateTime timestamp;
final String? emotion;
final double? confidenceScore;
final DateTime? analyzedAt;
final String? analysisStatus;
final List<String>? recommendations;

MoodEntry({
  required this.id,
  required this.userId,
  required this.text,
  required this.date,
  required this.timestamp,
  this.emotion,
  this.confidenceScore,
  this.analyzedAt,
  this.analysisStatus = 'pending',
  this.recommendations,
});

// Serialize to Firestore document
Map<String, dynamic> toFirestore() {
  return {
    'userId': userId,
    'text': text,
    'date': Timestamp.fromDate(date),
    'timestamp': Timestamp.fromDate(timestamp),
    'emotion': emotion,
    'confidenceScore': confidenceScore,
    'analyzedAt': analyzedAt != null ? Timestamp.fromDate(analyzedAt!) :
       null,
    'analysisStatus': analysisStatus,
    'recommendations': recommendations,
  };
}

// Deserialize from Firestore document
factory MoodEntry.fromFirestore(
  DocumentSnapshot<Map<String, dynamic>> snapshot,
) {
  final data = snapshot.data()!;
  return MoodEntry(
    id: snapshot.id,
    userId: data['userId'] ?? '',
    text: data['text'] ?? '',
    date: (data['date'] as Timestamp).toDate(),
    timestamp: (data['timestamp'] as Timestamp).toDate(),
    emotion: data['emotion'],
      confidenceScore: data['confidenceScore']?.toDouble(),
      analyzedAt: data['analyzedAt'] != null
          ? (data['analyzedAt'] as Timestamp).toDate()
          : null,
      analysisStatus: data['analysisStatus'] ?? 'pending',
      recommendations: data['recommendations'] != null
          ? List<String>.from(data['recommendations'])
          : null,
    );
  }
}
```

3.3 Security Rules

Firestore Security Rules enforce role-based access control:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {

    // Helper functions
    function isAuthenticated() {
      return request.auth != null;
    }

    function isOwner(userId) {
      return isAuthenticated() && request.auth.uid == userId;
    }

    function getUserRole() {
      return
        get(/databases/$(database)/documents/users/$(request.auth.uid)).data.role;
    }

    function isCounsellor() {
      return isAuthenticated() && getUserRole() == 'counsellor';
    }

    // Mood entries - users can only access their own
    // Counsellors can read any (app-level consent enforcement)
    match /mood_entries/{entryId} {
      allow read: if isOwner(resource.data.userId) || isCounsellor();
      allow create: if isAuthenticated()
                    && request.resource.data.userId == request.auth.uid;
      allow update: if isOwner(resource.data.userId)
                    && request.time < resource.data.timestamp +
        duration.value(24, 'h');
      allow delete: if isOwner(resource.data.userId);
    }
  }
}
```

4. Implementation of CRUD Operations

4.1 Mood Entry CRUD Operations

The `MoodEntryService` class handles all mood entry operations:

CREATE - Creating a New Mood Entry

```dart
class MoodEntryService {
  final FirebaseFirestore _firestore = FirebaseFirestore.instance;
  final String _collectionName = 'mood_entries';

  // Create a new mood entry
  Future<String> createMoodEntry({
    required String userId,
    required String text,
  }) async {
    try {
      final now = DateTime.now();
      final today = DateTime(now.year, now.month, now.day);

      final entry = MoodEntry(
        id: '', // Will be set by Firestore
        userId: userId,
        text: text,
        date: today,
        timestamp: now,
        analysisStatus: 'pending',
      );

      final docRef = await _firestore
          .collection(_collectionName)
          .add(entry.toFirestore());

      return docRef.id;
    } catch (e) {
      throw Exception('Failed to create mood entry: $e');
    }
  }
}
```

READ - Fetching Mood Entries

```dart
// Get all mood entries for a user
Future<List<MoodEntry>> getUserMoodEntries(String userId) async {
  try {
    final querySnapshot = await _firestore
        .collection(_collectionName)
    .where('userId', isEqualTo: userId)
    .orderBy('timestamp', descending: true)
    .get();

  return querySnapshot.docs
    .map((doc) => MoodEntry.fromFirestore(doc))
    .toList();
 } catch (e) {
  throw Exception('Failed to fetch mood entries: $e');
 }
}

// Get entries within a date range
Future<List<MoodEntry>> getUserMoodEntriesByDateRange(
 String userId,
 DateTime startDate,
 DateTime endDate,
) async {
 try {
  final querySnapshot = await _firestore
    .collection(_collectionName)
    .where('userId', isEqualTo: userId)
    .where('timestamp', isGreaterThanOrEqualTo:
     Timestamp.fromDate(startDate))
    .where('timestamp', isLessThanOrEqualTo: Timestamp.fromDate(endDate))
    .orderBy('timestamp', descending: true)
    .get();

  return querySnapshot.docs
    .map((doc) => MoodEntry.fromFirestore(doc))
    .toList();
 } catch (e) {
  throw Exception('Failed to fetch mood entries by date range: $e');
 }
}

// Real-time stream for live updates
Stream<List<MoodEntry>> streamUserMoodEntries(String userId) {
 return _firestore
   .collection(_collectionName)
   .where('userId', isEqualTo: userId)
   .orderBy('timestamp', descending: true)
   .snapshots()
   .map((snapshot) => snapshot.docs
     .map((doc) => MoodEntry.fromFirestore(doc))
     .toList());
}
```

UPDATE - Modifying Entries (via Cloud Functions)

Mood entries are primarily updated by Cloud Functions after AI analysis:

```
// Cloud Function: Update mood entry with analysis results
await admin.firestore().collection("mood_entries").doc(entryId).update({
  emotion: analysis.emotion,
  confidenceScore: analysis.confidenceScore,
  recommendations: recommendations,
  analyzedAt: admin.firestore.FieldValue.serverTimestamp(),
  analysisStatus: "completed",
});
```

DELETE - Removing Entries

```
Future<void> deleteMoodEntry(String entryId) async {
  try {
    await _firestore.collection(_collectionName).doc(entryId).delete();
  } catch (e) {
    throw Exception('Failed to delete mood entry: $e');
  }
}
```

4.2 Message CRUD Operations

CREATE - Sending Messages

```
Future<String> sendMessage({
  required String conversationThreadId,
  required String senderId,
  required String receiverId,
  required String content,
}) async {
  try {
    final message = MessageModel(
      id: '',
      conversationThreadId: conversationThreadId,
      senderId: senderId,
      receiverId: receiverId,
      content: content,
      timestamp: DateTime.now(),
      isRead: false,
    );

    final docRef = await _firestore
        .collection('conversation_threads')
        .doc(conversationThreadId)
        .collection('messages')
        .add(message.toFirestore());

    // Update conversation thread's last message timestamp
    await _firestore
        .collection('conversation_threads')
        .doc(conversationThreadId)
        .update({'lastMessageAt': Timestamp.now()});

    return docRef.id;
  } catch (e) {
    throw Exception('Failed to send message: $e');
  }
}
```

READ - Streaming Messages (Real-time)

```
Stream<List<MessageModel>> streamMessages(String conversationThreadId) {
  return _firestore
      .collection('conversation_threads')
      .doc(conversationThreadId)
      .collection('messages')
      .orderBy('timestamp', descending: false)
      .snapshots()
      .map((snapshot) => snapshot.docs
          .map((doc) => MessageModel.fromFirestore(doc))
          .toList());
}
```

UPDATE - Marking Messages as Read

```
Future<void> markAllMessagesAsRead(
  String conversationThreadId,
  String userId,
) async {
  try {
    final querySnapshot = await _firestore
        .collection('conversation_threads')
        .doc(conversationThreadId)
        .collection('messages')
        .where('receiverId', isEqualTo: userId)
        .where('isRead', isEqualTo: false)
        .get();

    // Batch update for efficiency
    final batch = _firestore.batch();
    for (final doc in querySnapshot.docs) {
      batch.update(doc.reference, {
        'isRead': true,
        'readAt': Timestamp.now(),
      });
    }

    await batch.commit();
  } catch (e) {
    throw Exception('Failed to mark all messages as read: $e');
  }
}
```

5. Important Source Code Snippets with Explanations

5.1 AI Mood Analysis (Cloud Function)

The core AI integration uses Mistral AI to analyze mood entries:

```
/**
 * Cloud Function triggered when a new mood entry is created.
 * Analyzes the text using Mistral AI and stores results.
 */
export const analyzeMoodEntry = functions.firestore
  .document("mood_entries/{entryId}")
  .onCreate(async (snap, context) => {
    const entryId = context.params.entryId;
    const entryData = snap.data();

    try {
      // Analyze mood using AI
      const analysis = await analyzeMoodWithMistral(entryData.text);

      // Generate personalized recommendations
      let recommendations: string[] = [];
      try {
        recommendations = await generateRecommendations(
          analysis.emotion,
          entryData.text
        );
      } catch (error) {
        // Fallback to static recommendations if AI fails
        recommendations = getFallbackRecommendations(analysis.emotion);
      }

      // Update the mood entry with results
      await
         admin.firestore().collection("mood_entries").doc(entryId).update({
        emotion: analysis.emotion,
        confidenceScore: analysis.confidenceScore,
        recommendations: recommendations,
        analyzedAt: admin.firestore.FieldValue.serverTimestamp(),
        analysisStatus: "completed",
      });

      return { success: true, analysis, recommendations };
    } catch (error) {
      // Mark as failed for retry mechanism
      await
         admin.firestore().collection("mood_entries").doc(entryId).update({
        analysisStatus: "failed",
        analyzedAt: admin.firestore.FieldValue.serverTimestamp(),
      });
      throw error;
    }
  });

/**
 * Analyzes mood text using Mistral AI API
 */
async function analyzeMoodWithMistral(
  text: string
): Promise<{ emotion: string; confidenceScore: number }> {
  const EMOTIONS = [
    "joy",
    "sadness",
    "anxiety",
    "anger",
    "fear",
    "contentment",
    "excitement",
    "frustration",
    "loneliness",
    "hope",
    "overwhelmed",
    "peaceful",
    "confused",
    "grateful",
    "stressed",
  ];

  const prompt = `Analyze the following mood journal entry and determine the primary emotion.
Choose only ONE emotion from this list: ${EMOTIONS.join(", ")}.

Journal Entry: "${text}"

Respond in JSON format with:
{
  "emotion": "the primary emotion from the list",
  "confidence": a number between 0 and 1,
  "reasoning": "brief explanation"
}`;

  const mistral = getMistralClient();
  const response = await mistral.chat({
    model: "mistral-small-latest",
    messages: [
      {
        role: "system",
        content:
          "You are an empathetic mental health assistant that analyzes mood journal entries.",
      },
      { role: "user", content: prompt },
    ],
    temperature: 0.3,
    maxTokens: 200,
    responseFormat: { type: "json_object" },
  });

  const result = JSON.parse(response.choices[0].message.content);
  return {
    emotion: result.emotion.toLowerCase(),
    confidenceScore: Math.max(0, Math.min(1, result.confidence || 0.5)),
  };
}
```

Explanation:

- Uses Firestore triggers to automatically analyze new entries
- Prompts the AI with a structured request for JSON output
- Includes fallback recommendations if AI fails
- Updates the document with analysis results asynchronously

5.2 Authentication Provider (State Management)

```
class AuthProvider extends ChangeNotifier {
  final AuthService _authService = AuthService();
  final FCMService _fcmService = FCMService();

  User? _firebaseUser;
  UserModel? _userModel;
  bool _isLoading = true;

  User? get firebaseUser => _firebaseUser;
  UserModel? get userModel => _userModel;
  bool get isLoading => _isLoading;
  bool get isAuthenticated => _firebaseUser != null;

  AuthProvider() {
    _init();
  }

  void _init() {
    // Listen to Firebase Auth state changes
    _authService.authStateChanges.listen((User? user) async {
      _firebaseUser = user;

      if (user != null) {
        try {
          // Fetch user profile from Firestore
          _userModel = await _authService.getUserProfile(user.uid);

          // Initialize push notifications
          await _fcmService.initializeFCM();
        } catch (e) {
          debugPrint('Error fetching user profile: $e');
          _userModel = null;
        }
      } else {
        _userModel = null;
      }

      _isLoading = false;
      notifyListeners(); // Notify UI to rebuild
    });
  }

  Future<void> signOut() async {
    try {
      // Clean up FCM token before signing out
      await _fcmService.removeToken().timeout(
        const Duration(seconds: 5),
        onTimeout: () {
          debugPrint('FCM token removal timed out');
        },
      );
    } catch (e) {
      debugPrint('Error removing FCM token: $e');
    }

    await _authService.signOut();
    _firebaseUser = null;
    _userModel = null;
    notifyListeners();
  }
}
```

Explanation:

- Uses the Provider pattern for reactive state management
- Listens to Firebase Auth stream for automatic state updates
- Combines Firebase Auth user with Firestore profile data
- Handles FCM token lifecycle for push notifications

5.3 Mood Trends Visualization

```
class MoodTrendsScreen extends StatefulWidget {
  @override
  State<MoodTrendsScreen> createState() => _MoodTrendsScreenState();
}

class _MoodTrendsScreenState extends State<MoodTrendsScreen> {
  final FirebaseFirestore _firestore = FirebaseFirestore.instance;
  String _chartType = 'line';
```

```
int _daysToShow = 7;

@override
Widget build(BuildContext context) {
  final user = FirebaseAuth.instance.currentUser;
  final startDate = DateTime.now().subtract(Duration(days: _daysToShow));

  return StreamBuilder<QuerySnapshot<Map<String, dynamic>>>(
    stream: _firestore
        .collection('mood_entries')
        .where('userId', isEqualTo: user!.uid)
        .snapshots(),
    builder: (context, snapshot) {
      if (snapshot.connectionState == ConnectionState.waiting) {
        return const Center(child: CircularProgressIndicator());
      }

      // Filter and sort on client side for complex queries
      final docs = snapshot.data?.docs ?? [];
      final filteredDocs = docs.where((doc) {
        final timestamp = doc.data()['timestamp'] as Timestamp?;
        return timestamp != null && timestamp.toDate().isAfter(startDate);
      }).toList()
        ..sort((a, b) {
          final aTime = (a.data()['timestamp'] as Timestamp).toDate();
          final bTime = (b.data()['timestamp'] as Timestamp).toDate();
          return aTime.compareTo(bTime);
        });

      // Convert to chart data and render
      return _buildChart(filteredDocs);
    },
  );
}

Widget _buildLineChart(List<MoodEntry> entries) {
  // Map emotions to numeric values for charting
  final emotionValues = {
    'joy': 5, 'excitement': 5, 'grateful': 4.5,
    'contentment': 4, 'peaceful': 4, 'hope': 3.5,
    'confused': 2.5, 'stressed': 2, 'frustrated': 2,
    'anxiety': 1.5, 'anger': 1.5, 'sadness': 1,
    'fear': 1, 'loneliness': 1, 'overwhelmed': 0.5
  };

  final spots = entries.asMap().entries.map((entry) {
    final value = emotionValues[entry.value.emotion] ?? 2.5;
    return FlSpot(entry.key.toDouble(), value);
  }).toList();

  return LineChart(
      LineChartData(
        lineBarsData: [
          LineChartBarData(
            spots: spots,
            isCurved: true,
            color: Theme.of(context).colorScheme.primary,
            barWidth: 3,
            dotData: FlDotData(show: true),
          ),
        ],
        // ... axis configurations
      ),
    );
  }
}
```

Explanation:

- Uses `StreamBuilder` for real-time data updates
- Client-side filtering for complex date range queries
- Maps categorical emotions to numeric values for visualization
- Uses fl_chart library for interactive charts

5.4 Real-time Messaging

```
class ConversationScreen extends StatefulWidget {
  final String threadId;

  @override
  State<ConversationScreen> createState() => _ConversationScreenState();
}

class _ConversationScreenState extends State<ConversationScreen> {
  final MessageService _messageService = MessageService();
  final TextEditingController _controller = TextEditingController();

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        // Real-time message list
        Expanded(
          child: StreamBuilder<List<MessageModel>>(
            stream: _messageService.streamMessages(widget.threadId),
            builder: (context, snapshot) {
              if (!snapshot.hasData) {
                return const Center(child: CircularProgressIndicator());
              }

              final messages = snapshot.data!;
              return ListView.builder(
                reverse: true,
                itemCount: messages.length,
                itemBuilder: (context, index) {
                  final message = messages[messages.length - 1 - index];
                  return MessageBubble(
                    message: message,
                    isMe: message.senderId == currentUserId,
                  );
                },
              );
            },
          ),
        ),

        // Message input
        _buildMessageInput(),
      ],
    );
  }

  Future<void> _sendMessage() async {
    if (_controller.text.trim().isEmpty) return;

    await _messageService.sendMessage(
      conversationThreadId: widget.threadId,
      senderId: currentUserId,
      receiverId: otherUserId,
      content: _controller.text.trim(),
    );

    _controller.clear();
  }
}
```

Explanation:

- Uses Firestore snapshots for real-time message updates
- Messages appear instantly without manual refresh
- Efficient list rendering with `ListView.builder`

---

6. Technical Challenges and Solutions

6.1 Challenge: Complex Firestore Queries

Problem: Firestore doesn’t support compound queries with inequality filters on multiple fields, making it difficult to filter mood entries by user AND date range.

Solution: Implemented client-side filtering for complex queries:

```
// Fetch all user entries, then filter by date on client
final docs = snapshot.data?.docs ?? [];
final filteredDocs = docs.where((doc) {
  final timestamp = doc.data()['timestamp'] as Timestamp?;
  return timestamp != null && timestamp.toDate().isAfter(startDate);
}).toList();
```

Alternative: Created composite indexes in `firestore.indexes.json` for supported query patterns.

6.2 Challenge: AI API Reliability

Problem: External AI APIs (Mistral) can fail or timeout, leaving mood entries without analysis.

Solution: Implemented a multi-layered fallback system:

1. **Status tracking:** Each entry has `analysisStatus` field (‘pending’, ‘completed’, ‘failed’)
2. **Fallback recommendations:** Static recommendations per emotion category
3. **Retry mechanism:** Failed entries can be retried manually or via scheduled function

```
const FALLBACK_RECOMMENDATIONS: Record<string, string[]> = {
  joy: [
    "Share your happiness with someone you care about.",
    "Practice gratitude by writing down three things you're thankful for.",
  ],
  sadness: [
    "It's okay to feel sad. Give yourself permission to feel.",
    "Connect with a friend or loved one for support.",
  ],
  // ... other emotions
};

function getFallbackRecommendations(emotion: string): string[] {
  return (
    FALLBACK_RECOMMENDATIONS[emotion] || FALLBACK_RECOMMENDATIONS["confused"]
  );
}
```

6.3 Challenge: Real-time Data Synchronization

Problem: Keeping UI in sync with database changes across multiple screens and users.

Solution: Used Firestore’s real-time listeners throughout the app:

```
// Stream-based approach for automatic updates
Stream<List<MoodEntry>> streamUserMoodEntries(String userId) {
  return _firestore
      .collection('mood_entries')
      .where('userId', isEqualTo: userId)
      .orderBy('timestamp', descending: true)
      .snapshots()
      .map((snapshot) => snapshot.docs
          .map((doc) => MoodEntry.fromFirestore(doc))
          .toList());
}
```

Combined with `StreamBuilder` widgets for reactive UI updates.

6.4 Challenge: Push Notification Setup

Problem: FCM token management across login/logout cycles and ensuring notifications reach the correct user.

Solution: Integrated FCM token lifecycle with authentication:

```
class AuthProvider extends ChangeNotifier {
  void _init() {
    _authService.authStateChanges.listen((User? user) async {
      if (user != null) {
        // Initialize FCM and save token to user document
        await _fcmService.initializeFCM();
      }
    });
  }

  Future<void> signOut() async {
    // Remove FCM token before signing out
    await _fcmService.removeToken().timeout(
      const Duration(seconds: 5),
      onTimeout: () => debugPrint('Token removal timed out'),
    );
    await _authService.signOut();
  }
}
```

6.5 Challenge: Role-Based Access Control

Problem: Ensuring counsellors can only access data from users who have consented to share.

Solution: Multi-layered security approach:

1. **Firestore Security Rules:** Server-side enforcement
2. **Application Logic:** Filter queries by assignment status
3. **Consent Model:** Support requests explicitly link users to counsellors

```
// Counsellor can only see users who have accepted support requests
final requestsSnapshot = await _firestore
    .collection('support_requests')
    .where('counsellorId', isEqualTo: counsellorId)
    .where('status', whereIn: ['accepted', 'inProgress'])
    .get();
```

6.6 Challenge: Offline Support

Problem: App should remain functional when network is unavailable.

Solution: Leveraged Firestore's built-in offline persistence:

- Firestore automatically caches data locally
- Writes are queued and synced when online
- `StreamBuilder` continues to work with cached data

```
// Firestore offline persistence is enabled by default in Flutter
// No additional configuration required
```

---

7. Conclusion

MoodMate demonstrates a modern approach to mobile app development using Flutter and Firebase. Key technical achievements include:

- **Clean Architecture:** Separation of concerns with models, services, providers, and screens
- **Real-time Sync:** Firestore streams for live data updates
- **AI Integration:** Serverless AI processing via Cloud Functions
- **Security:** Multi-layered security with Firebase Auth and Firestore rules
- **Cross-platform:** Single codebase for iOS, Android, and Web

The project successfully implements all planned features while maintaining code quality and user experience standards.

Document Version: 1.0
Last Updated: January 2026
Project: MoodMate - AI-Powered Mental Wellness Companion