MessageAI

Building Cross-Platform Messaging Apps with AI Features

Background

WhatsApp transformed how billions communicate by making messaging fast, reliable, and secure. The app works seamlessly across mobile platforms, handles offline scenarios gracefully, and delivers messages instantly even on poor network connections.

What's remarkable is that WhatsApp was originally built by just two developers—Brian Acton and Jan Koum—in a matter of months. They created an app that would eventually serve over 2 billion users worldwide. With today's AI coding tools, you can absolutely build a production-quality messaging app in one week—and potentially take it even further than they initially did.

This required solving complex technical challenges: message persistence, real-time delivery, optimistic UI updates, efficient data sync, and cross-platform compatibility.

Now imagine adding AI to this. What if your messaging app could automatically summarize long conversation threads? Or translate messages in real-time? Or provide an AI agent that helps you draft responses, schedule messages, or extract action items from group chats?

This project challenges you to build both a production-quality messaging infrastructure—like WhatsApp—and AI features that enhance the messaging experience using LLMs, agents, and RAG pipelines.

Why This Matters

The future of messaging isn't just about sending texts—it's about intelligent communication. You'll be building the foundation for how AI can make conversations more productive, accessible, and meaningful.

Project Overview

This is a one-week sprint with three key deadlines:

- **MVP: Tuesday (24 hours)**
- **Early Submission: Friday (4 days)**
- **Final: Sunday (7 days)**

You'll build in two phases: first the core messaging infrastructure with real-time sync and offline support, then AI features tailored to a specific user persona.

MVP Requirements (24 Hours)

This is a hard gate. To pass the MVP checkpoint, you must have:

- One-on-one chat functionality
- Real-time message delivery between 2+ users
- Message persistence (survives app restarts)
- Optimistic UI updates (messages appear instantly before server confirmation)
- Online/offline status indicators
- Message timestamps
- User authentication (users have accounts/profiles)
- Basic group chat functionality (3+ users in one conversation)
- Message read receipts
- Push notifications working (at least in foreground)
- **Deployment**: Running on local emulator/simulator with deployed backend (TestFlight/APK/Expo Go if possible, but not required for MVP)

The MVP isn't about features—it's about proving your messaging infrastructure is solid. A simple chat app with reliable message delivery is worth more than a feature-rich app with messages that don't sync reliably.

Platform Requirements

Choose **ONE** of the following:

- **Swift (iOS native)** - SwiftUI or UIKit
- **Kotlin (Android native)** - Jetpack Compose or XML
- **React Native with Expo** - Must use Expo Go or custom dev client

Core Messaging Infrastructure

Essential Features

Your messaging app needs one-on-one chat with real-time message delivery. Messages must persist locally—users should see their chat history even when offline. Support text messages with timestamps and read receipts.

Implement online/offline presence indicators. Show when users are typing. Handle message delivery states: sending, sent, delivered, read.

Include basic media support—at minimum, users should be able to send and receive images. Add profile pictures and display names.

Build group chat functionality supporting 3+ users with proper message attribution and delivery tracking.

Real-Time Messaging

Every message should appear instantly for online recipients. When users go offline, messages queue and send when connectivity returns. The app must handle poor network conditions gracefully—3G, packet loss, intermittent connectivity.

Implement optimistic UI updates. When users send a message, it appears immediately in their chat, then updates with delivery confirmation. Messages never get lost—if the app crashes mid-send, the message should still go out.

Testing Scenarios

We'll test with:

1. Two devices chatting in real-time
2. One device going offline, receiving messages, then coming back online
3. Messages sent while app is backgrounded
4. App force-quit and reopened to verify persistence
5. Poor network conditions (airplane mode, throttled connection)
6. Rapid-fire messages (20+ messages sent quickly)
7. Group chat with 3+ participants

Choose Your User Persona

You must build for **ONE** of these specific user types. Your AI features should be tailored to their needs.

For each persona, you must implement:

1. All 5 required AI features listed below
2. **ONE** advanced AI capability from the options provided

Persona Comparison

Persona	Who They Are	Core Pain Points	Required AI Features (All 5)	Advanced Features (Only 1)
Remote Team Professional	Software engineers, designers, PMs in distributed teams.	• Drowning in threads • Missing important messages • Context switching • Time zone coordination	1. Thread summarization 2. Action item extraction 3. Smart search 4. Priority message detection 5. Decision tracking	**A) Multi-Step Agent**: Plans team offsites, coordinates schedules autonomously **B) Proactive Assistant**: Auto-suggests meeting times, detects scheduling needs
International Communicator	People with friends/family/colleagues speaking different languages.	• Language barriers • Translation nuances • Copy-paste overhead • Learning difficulty	1. Real-time translation (inline) 2. Language detection & auto-translate 3. Cultural context hints 4. Formality level adjustment 5. Slang/idiom explanations	**A) Context-Aware Smart Replies**: Learns your style in multiple languages **B) Intelligent Processing**: Extracts structured data from multilingual conversations
Busy Parent/Caregiver	Parents coordinating schedules, managing multiple responsibilities.	• Schedule juggling • Missing dates/appointments • Decision fatigue • Information overload	1. Smart calendar extraction 2. Decision summarization 3. Priority message highlighting 4. RSVP tracking 5. Deadline/reminder extraction	**A) Proactive Assistant**: Detects scheduling conflicts, suggests solutions **B) Multi-Step Agent**: Plans weekend activities based on family preferences

Content Creator/Influencer	YouTubers, TikTokers managing fan communication.	• Hundreds of DMs daily • Repetitive questions • Spam vs opportunities • Maintaining authentic voice	1. Auto-categorization (fan/business/spam/urgent) 2. Response drafting in creator's voice 3. FAQ auto-responder 4. Sentiment analysis 5. Collaboration opportunity scoring	**A) Context-Aware Smart Replies**: Generates authentic replies matching personality **B) Multi-Step Agent**: Handles daily DMs, auto-responds to FAQs, flags key messages

AI Features Implementation

All AI features should be built using **LLMs** (like GPT-4 or Claude), **function calling/tool use**, and **RAG pipelines** for accessing conversation history. This is not about training ML models—it's about leveraging existing AI capabilities through prompting and tool integration.

Technical Implementation

AI Architecture Options:

Option 1: AI Chat Interface A dedicated AI assistant in a special chat where users can:

- Ask questions about their conversations
- Request actions ("Translate my last message to Spanish")
- Get proactive suggestions

Option 2: Contextual AI Features AI features embedded directly in conversations:

- Long-press message → translate/summarize/extract action
- Toolbar buttons for quick AI actions
- Inline suggestions as users type

Option 3: Hybrid Approach Both a dedicated AI assistant AND contextual features

AI Integration Requirements

The following agent frameworks are recommended:

- **AI SDK by Vercel** - streamlined agent development with tool calling

- **OpenAI Agent SDK (Swarm)** - lightweight multi-agent orchestration
- **LangChain** - comprehensive agent framework with extensive tools

Your agent should have:

- Conversation history retrieval (RAG pipeline)
- User preference storage
- Function calling capabilities
- Memory/state management across interactions
- Error handling and recovery

Technical Stack (Recommended)

The Golden Path: Firebase + Swift

Backend:

- **Firebase Firestore** - real-time database
- **Firebase Cloud Functions** - serverless backend for AI calls
- **Firebase Auth** - user authentication
- **Firebase Cloud Messaging (FCM)** - push notifications

Mobile (iOS):

- **Swift** with SwiftUI
- **SwiftData** for local storage
- **URLSession** for networking
- **Firebase SDK**
- Deploy via **TestFlight**

AI Integration:

- **OpenAI GPT-4** or **Anthropic Claude** (called from Cloud Functions)
- Function calling / tool use
- **AI SDK by Vercel** or **LangChain** for agents

Why This Stack:

- Firebase handles real-time sync out of the box
- Cloud Functions keep API keys secure
- SwiftUI is fastest for iOS development
- Everything deploys easily

Alternative Paths

React Native:

- Expo Router, Expo SQLite, Expo Notifications
- Deploy via Expo Go
- Still use Firebase backend

Android:

- Kotlin with Jetpack Compose
- Room Database
- Firebase SDK
- Deploy via APK

Other Backends:

- AWS (DynamoDB, Lambda, API Gateway, SNS)
- Supabase (PostgreSQL, Realtime, Auth)

Build Strategy

Start with Messages First: Get basic messaging working end-to-end before anything else:

1. Send a text message from User A → appears on User B's device
2. Messages persist locally (works offline)
3. Messages sync on reconnect
4. Handle app lifecycle (background/foreground)

Only after messaging is solid should you add AI features.

Build Vertically: Finish one slice at a time. Don't have 10 half-working features.

Test on Real Hardware: Simulators don't accurately represent performance, networking, or app lifecycle. Use physical devices.

For AI Features:

- Start with simple prompts, iterate to improve accuracy
- Use RAG to give the LLM conversation context
- Test with edge cases (empty conversations, mixed languages, etc.)
- Cache common AI responses to reduce costs

Final Submission Requirements

Submit the following by **Sunday 10:59 PM CT**:

1. **GitHub Repository** - with comprehensive README with setup instructions
2. **Demo Video (5-7 minutes)** showing:
   - Real-time messaging between two devices
   - Group chat with 3+ participants
   - Offline scenario (go offline, receive messages, come online)
   - App lifecycle handling (background, foreground, force quit)
   - All 5 required AI features in action with clear examples
   - Your advanced AI capability with specific use cases
3. **Deployed Application**:
   - **iOS**: TestFlight link
   - **Android**: APK download link or Google Play internal testing link
   - **React Native**: Expo Go link
   - **Note**: If deployment is blocked, provide detailed local setup instructions.
4. **Persona Brainlift** - 1-page document explaining:
   - Your chosen persona and why
   - Their specific pain points you're addressing
   - How each AI feature solves a real problem
   - Key technical decisions you made
5. **Social Post** - Share your project on X (Twitter) or LinkedIn with:
   - Brief description of what you built (2-3 sentences)
   - Key features and your chosen persona
   - Demo video or screenshots
   - Tag @GauntletAI

Final Note

This project mirrors what the best AI communication startups are building right now—combining robust messaging infrastructure with intelligent AI features that genuinely help users.

Remember: WhatsApp was built by two developers in months. With modern AI coding tools, you can build something comparable in one week and push it even further with intelligent features that didn't exist back then.

The closer you get to that experience, the more you'll understand what it takes to build the next generation of messaging apps.

A simple, reliable messaging app with truly useful AI features beats any feature-rich app with flaky message delivery or gimmicky AI.

Build something people would actually want to use every day.