Prompt 1:

Build a Flutter Android app called "TuTu" - a personal AI agent manager with persistent memory and local RAG system.

PROJECT STRUCTURE & SETUP

Create a complete Flutter project with this structure:

```
tutu_app/
├── lib/
│   ├── main.dart
│   ├── models/
│   │   ├── agent_model.dart
│   │   ├── message_model.dart
│   │   ├── conversation_model.dart
│   │   └── qa_bank_model.dart
│   ├── services/
│   │   ├── storage_service.dart
│   │   ├── api_service.dart
│   │   ├── rag_service.dart
│   │   ├── offline_qa_service.dart
│   │   └── memory_manager.dart
│   ├── screens/
│   │   ├── splash_screen.dart
│   │   ├── onboarding_screen.dart
│   │   ├── home_screen.dart
│   │   ├── agent_list_screen.dart
│   │   ├── chat_screen.dart
│   │   ├── create_agent_screen.dart
│   │   ├── settings_screen.dart
│   │   └── api_setup_screen.dart
│   ├── widgets/
│   │   ├── agent_card.dart
│   │   ├── message_bubble.dart
│   │   ├── custom_app_bar.dart
│   │   └── typing_indicator.dart
│   └── utils/
│       ├── constants.dart
│       ├── helpers.dart
│       └── themes.dart
└── assets/
    ├── qa_bank.json
    └── images/
```

DEPENDENCIES (pubspec.yaml)

Add these dependencies:

```yaml
dependencies:
  flutter:
    sdk: flutter
  shared_preferences: ^2.2.2
  sqflite: ^2.3.0
```

```
  path_provider: ^2.1.1
  http: ^1.1.0
  provider: ^6.1.1
  uuid: ^4.2.1
  intl: ^0.18.1
  flutter_markdown: ^0.6.18
  sembast: ^3.5.0
```
1. DATA MODELS
agent_model.dart
Create an Agent model with:
- id (unique UUID)
- name (e.g., "Maya", "Paro", "CA Agent")
- role (e.g., "girlfriend", "lawyer", "financial advisor")
- personality (description for system prompt)
- voiceId (for future voice integration)
- avatar (emoji or icon identifier)
- createdAt
- lastInteractionAt
- Methods: toJson(), fromJson()
message_model.dart
Create a Message model with:
- id
- agentId
- role ("user" or "assistant")
- content
- timestamp
- metadata (map for storing context, embeddings, etc.)
- Methods: toJson(), fromJson()
conversation_model.dart
Create a Conversation model that groups messages by agentId with methods to retrieve conversation history.
qa_bank_model.dart
Create a QABank model for offline question-answering:
- id
- question (normalized/tokenized)
- answer
- category (e.g., "app_usage", "agent_creation", "troubleshooting")
- keywords (list of searchable terms)
2. CORE SERVICES
storage_service.dart
Implement using sembast (NoSQL) and shared_preferences:
- saveAgent(Agent agent)
- getAgent(String agentId)
- getAllAgents()
- deleteAgent(String agentId)
- saveMessage(Message message)
- getMessagesByAgent(String agentId, {int limit, int offset})
- saveApiKey(String apiKey, String provider)

- getApiKey()
- saveUserPreferences(Map<String, dynamic> prefs)
- Initialize database on app start
- Implement efficient chunking (save messages in batches of 50)
api_service.dart
Create a flexible LLM API service that supports:
- Multiple providers (OpenAI, Anthropic, Gemini, DeepSeek, OpenRouter)
- Provider enum: openai, anthropic, gemini, deepseek, openrouter, custom
- sendMessage(String message, String agentId, List<Message> conversationHistory)
- buildSystemPrompt(Agent agent) - create personality-based prompts
- Error handling with user-friendly messages
- Retry logic (3 attempts with exponential backoff)
- Token counting estimation
- API endpoint configuration per provider
For OpenRouter:
- Base URL: https://openrouter.ai/api/v1/chat/completions
- Headers: Authorization: Bearer {API_KEY}, HTTP-Referer: {app_url}
- Support model selection (let user choose from list)
rag_service.dart
Implement a simple RAG system:
- addToMemory(String agentId, String content, Map<String, dynamic> metadata)
- searchMemory(String agentId, String query, {int limit = 5})
- Simple keyword-based search (split into words, match against stored content)
- Store embeddings as simple word frequency vectors (avoid external APIs)
- Implement TF-IDF scoring for retrieval
- Auto-summarization: when conversation exceeds 50 messages, create summary and store separately
offline_qa_service.dart
Build the 1000+ question bank system:
- Load qa_bank.json on app initialization
- normalizeQuery(String query) - lowercase, remove punctuation, tokenize
- findBestMatch(String query) - use fuzzy string matching
- Scoring algorithm: Levenshtein distance + keyword overlap
- Return answer if confidence > 75%, otherwise say "Let me connect you to the internet for that"
- Categories: app_usage, agent_creation, agent_customization, troubleshooting, features
memory_manager.dart
Manage TuTu's memory operations:
- Auto-create RAG entries from user conversations
- Detect important information (dates, preferences, names, events)
- Store in structured format with timestamps
- Implement memory retrieval for context injection
- Keep last 20 messages in active memory, older messages in RAG
3. SCREENS
splash_screen.dart
- Show "TuTu" logo with loading animation
- Initialize database, load QA bank
- Check if API key exists, navigate to home or onboarding
onboarding_screen.dart
3-step onboarding:
1. Welcome to TuTu - explain the concept

2. API Setup options:
   - "I have an API key" → manual input screen
   - "Help me get started" → OpenRouter signup flow
   - "I'll do this later" → skip (limited to TuTu agent with offline QA)
3. Create your first agent tutorial
home_screen.dart
Display:
- Greeting: "Welcome back, [user name]"
- TuTu agent card (always present, cannot be deleted)
- Grid/List of user-created agents
- FAB button "Create New Agent"
- Bottom nav: Home, Agents, Settings
agent_list_screen.dart
- Show all agents with last interaction time
- Search/filter functionality
- Swipe to delete (except TuTu)
- Tap to open chat
chat_screen.dart
Full-featured chat interface:
- AppBar: Agent name, avatar, settings icon
- Message list (reverse ListView)
- Message bubbles (different colors for user/agent)
- Typing indicator when waiting for response
- Input field with send button
- Context injection: automatically include last 20 messages + relevant RAG memories
- Handle long responses (streaming would be ideal but complex, so show loading)
- Error handling with retry button
create_agent_screen.dart
Form to create custom agents:
- Name (text field)
- Role (dropdown: Girlfriend, Lawyer, Financial Advisor, Teacher, Friend, Custom)
- Personality description (multi-line text field)
- Avatar selection (emoji picker)
- Voice preference (placeholder for now - Male/Female selection)
- Save button validates and creates agent
settings_screen.dart
- API Configuration (change provider, update key)
- Default model selection (if OpenRouter)
- User profile (name, preferences)
- Memory management (clear all data, export data)
- About section
- View API usage/costs (if OpenRouter)
api_setup_screen.dart
Two tabs:
1. Manual Setup:
   - Provider dropdown (OpenAI, Anthropic, Gemini, DeepSeek, OpenRouter, Custom)
   - API Key input field (secure, masked)
   - Custom endpoint field (for "Custom" provider)
   - Test connection button

   - Step-by-step guide showing how to get API key for selected provider
2. OpenRouter Setup (placeholder for now):
   - Email/phone input
   - "Create Account" button
   - Instructions: "We'll help you create an account and manage API access"
   - Note: "This feature connects you to OpenRouter for easier API management"
4. QA BANK SETUP (assets/qa_bank.json)
Create a comprehensive JSON file with 1000+ Q&A pairs covering:
Categories:
- app_usage (100+ questions)
- agent_creation (200+ questions)
- agent_customization (200+ questions)
- troubleshooting (150+ questions)
- features (150+ questions)
- api_setup (200+ questions)
Format:
```json
[
  {
    "id": 1,
    "question": "How do I create a new agent?",
    "answer": "Tap the '+' button on the home screen, fill in the agent's name, role, and personality, then tap 'Create Agent'.",
    "category": "agent_creation",
    "keywords": ["create", "new", "agent", "add"]
  },
  ...
]
```
Include variations of common questions. Generate comprehensive coverage of:
- What is TuTu? How does it work?
- How to create agents?
- What are agents? Can I customize them?
- How does memory work?
- Why does my agent remember/forget things?
- API key setup for each provider
- Troubleshooting connection issues
- How to switch between agents?
- Can agents talk to each other? (No, but future feature)
- Privacy and data storage
- How is my data stored?
- Can I export my conversations?
5. TUTU DEFAULT AGENT BEHAVIOR
TuTu should:
- Always check offline QA bank first before using API
- Only use API for:
  - Questions not in QA bank (confidence < 75%)
  - User explicitly asks to "search online" or "check latest info"
  - Complex reasoning beyond stored Q&A

- Personality: Helpful, friendly, efficient assistant
- System prompt: "You are TuTu, a smart assistant managing this personal AI agent app. You help users understand the app, create agents, and manage their AI companions. Be concise, friendly, and helpful."

6. MEMORY PERSISTENCE STRATEGY

Implement multi-layer memory:
1. **Active Memory** (last 20 messages in RAM)
2. **Short-term Memory** (last 500 messages in SQLite/Sembast)
3. **Long-term Memory** (all messages, indexed in RAG)
4. **Episodic Memory** (important events, summaries, user preferences)

Auto-summarization:
- Every 100 messages, create summary of conversation themes
- Store summaries separately
- Inject summaries when context is needed

7. UI/UX REQUIREMENTS

Theme:
- Modern, clean Material Design 3
- Primary color: Purple/Blue gradient
- Agent cards: Colorful, distinct per agent
- Dark mode support

Animations:
- Smooth transitions between screens
- Message bubble animations
- Typing indicator
- Pull-to-refresh in chat

8. ERROR HANDLING

Comprehensive error handling:
- No API key: Guide user to setup
- API errors: Show user-friendly message, offer retry
- Network issues: Detect and inform user
- Storage errors: Graceful degradation
- Validation errors: Clear inline messages

9. IMPLEMENTATION PRIORITIES

Build in this order:
1. Data models and storage service
2. Offline QA service with sample QA bank (100 entries for testing)
3. TuTu agent with offline responses
4. API service with OpenAI provider
5. Basic chat screen with TuTu
6. Agent creation flow
7. Multi-agent chat with memory
8. RAG service integration
9. Settings and API configuration
10. Polish UI/UX

10. TESTING REQUIREMENTS

Create:
- Sample QA bank with 100 entries for testing
- Pre-populated TuTu agent on first launch
- Test mode in settings (use dummy API responses)
- Debug logs for API calls and memory operations

DELIVERABLES
1. Complete Flutter project structure
2. All model classes with serialization
3. All service classes with full implementation
4. All screens with functional UI
5. Sample QA bank JSON file
6. README with setup instructions
7. Comments explaining key logic
Build this as a production-ready app with clean code, proper state management (Provider), and comprehensive error handling. The app should work offline with TuTu's QA bank and seamlessly integrate LLM APIs when configured.
Prompt 2:
Extend the TuTu app with advanced features: voice synthesis, facial recognition, and OpenRouter integration.
ADDITIONAL DEPENDENCIES (pubspec.yaml)
Add:
```yaml
dependencies:
  flutter_tts: ^3.8.5  # Text-to-speech
  camera: ^0.10.5+5  # Camera access
  google_mlkit_face_detection: ^0.9.0  # Face detection
  image: ^4.1.3  # Image processing
  path: ^1.8.3
  url_launcher: ^6.2.1  # For OpenRouter web flows
```
1. VOICE SYNTHESIS SERVICE
voice_service.dart
Create a service using flutter_tts:
- Initialize TTS engine
- setVoice(String voiceId, String gender) - "male" or "female"
- speak(String text, String agentId)
- Stop/pause functionality
- Volume and pitch control
- List available voices (system-dependent)
- Store voice preferences per agent in storage
Add voice selection in create_agent_screen.dart:
- Dropdown: Male/Female
- Voice preview button (speaks sample text)
- Store voiceId and gender in Agent model
Enable voice in chat_screen.dart:
- Speaker icon on each agent message bubble
- Tap to speak the message
- Auto-speak option in settings (speak every agent response)
2. FACIAL RECOGNITION SYSTEM
face_recognition_service.dart
Build a local face recognition system:
Face Detection:
- Use google_mlkit_face_detection to detect faces
- Extract face landmarks (eyes, nose, mouth positions)

- Calculate face encoding:
  - Distance between eyes
  - Eye-nose ratio
  - Nose-mouth ratio
  - Face width/height ratio
  - Face angle
  - Create 20-30 dimensional vector from these measurements

Face Storage Model (face_model.dart):
- id
- personName
- agentId (which agent knows this person)
- faceEncoding (List<double> - the mathematical representation)
- imageVersion (e.g., "with_beard", "short_hair", "childhood")
- imagePath (store actual photo in local storage)
- detectedAt (timestamp)
- metadata (hair color, glasses, beard, etc.)

Storage Structure:

```
faces/
├── person_1/
│   ├── version_1.jpg (current look)
│   ├── version_2.jpg (no beard)
│   ├── version_3.jpg (childhood)
│   └── encodings.json
├── person_2/
│   └── ...
```

Implementation:
- captureAndRecognizeFace(String agentId)
- registerNewFace(String personName, String agentId, File imageFile, {String version})
- recognizeFace(File imageFile, String agentId) → returns Person or null
- updateFaceVersion(String personId, File imageFile, String version)

Face Matching:
- Calculate Euclidean distance between face encodings
- Threshold: if distance < 0.6, it's a match
- If match found, return person name
- If no match, offer to register new face

Integration in chat_screen.dart:
- Camera icon button
- Opens camera preview
- Capture photo
- Process face recognition
- If recognized: "I can see it's [Name]! How are you today?"
- If not recognized: "I don't recognize this person. Would you like me to remember them?"
- Store recognition event in memory/RAG

3. OPENROUTER INTEGRATION

openrouter_service.dart

Full OpenRouter integration:

Account Creation Flow:
- openRouterSignup(String email)
- Use url_launcher to open: https://openrouter.ai/auth/signup

- After signup, guide user to: Settings → API Keys → Create new key
- Copy key and paste in app
API Key Management:
- validateOpenRouterKey(String apiKey)
- Test endpoint: GET https://openrouter.ai/api/v1/auth/key
- Store key securely in shared_preferences
Model Selection:
- getAvailableModels() - fetch from OpenRouter API
- Display models with pricing information
- Categories: Free, Cheap, Balanced, Premium
- Popular models:
  - GPT-4 Turbo
  - GPT-3.5 Turbo
  - Claude 3.5 Sonnet
  - Claude 3 Haiku
  - Gemini Pro
  - DeepSeek
  - Llama 3
- Store selected model per agent
Usage Tracking:
- Fetch usage stats: GET https://openrouter.ai/api/v1/auth/key
- Display:
  - Current balance
  - Today's usage
  - This month's usage
  - Cost per agent
- Low balance warning (< $1)
Top-up/Recharge:
- Button: "Add Credits"
- Open OpenRouter billing page: https://openrouter.ai/credits
- User adds credits via their web interface
- Refresh balance in app
4. ENHANCED UI SCREENS
camera_screen.dart
New screen for face recognition:
- Live camera preview
- Face detection overlay (draw rectangle around detected faces)
- Capture button
- Recognition result display
- Options: "Register as new person" or "Add to existing person"
voice_settings_screen.dart
Voice configuration:
- Test different voices
- Adjust speech rate
- Adjust pitch
- Enable/disable auto-speak
- Per-agent voice assignment
openrouter_dashboard_screen.dart
Add to settings:

- Current balance (large, prominent)
- Usage chart (last 7 days)
- Model selection per agent
- Top-up button
- Transaction history
5. ENHANCED AGENT INTERACTION
Update chat_screen.dart:
Voice Integration:
- Toggle button: Enable/disable voice responses
- When enabled, agent responses are spoken automatically
- User can still manually tap to replay
Vision Integration:
- Camera button in input area
- Take photo → send to agent
- If face detected: Include recognition data in context
- Agent responds: "I can see [Name] in this photo!"
- Store photo reference in conversation
Context Awareness:
- When face recognized, inject into prompt:
  - "You are talking to [Name]. Remember past conversations with them."
  - Retrieve past interactions with that person from RAG
  - Personalize response based on relationship
6. MEMORY ENHANCEMENTS
Enhanced RAG with Vision:
- Store face recognition events
- "I met [Name] on [Date]. They looked [description]."
- Link conversations to recognized faces
- Retrieve past conversations when person is recognized again
Multi-modal Memory:
```json
{
  "type": "face_recognition",
  "personName": "John",
  "agentId": "maya_agent",
  "timestamp": "2024-01-15T14:30:00",
  "imageVersion": "current",
  "context": "John showed me his new haircut"
}
```
7. OFFLINE CAPABILITIES
Ensure offline functionality:
- Face recognition works completely offline
- Voice synthesis works offline (uses device TTS)
- Only LLM API calls require internet
8. PRIVACY & SECURITY
Implement:
- Face data stored locally only (never sent to cloud)
- Encryption for stored face encodings
- User consent before using camera

- Clear data deletion (faces, voices, conversations)
- Privacy policy screen

9. PERFORMANCE OPTIMIZATIONS

- Lazy load face encodings (only for current agent)
- Compress stored images (max 512x512)
- Cache voice synthesis
- Limit face database to 100 people per agent

10. TESTING

Create:
- Mock face recognition for testing without camera
- Sample face encodings
- Voice synthesis testing with multiple voices
- OpenRouter mock responses

INTEGRATION CHECKLIST

1. Voice service integrated in all agent chats
2. Camera accessible from chat screen
3. Face recognition working end-to-end
4. OpenRouter account creation flow functional
5. Balance display and top-up flow working
6. Multi-modal memory storing face + voice + text
7. Privacy controls accessible
8. Offline modes working correctly

Build these features incrementally, ensuring each works before moving to the next. Prioritize: Voice → Face Recognition → OpenRouter integration.