isep
École d'ingénieurs du numérique

II.3510 Mobile Development in Android

Academic Year 2025-2026

ORA

AI Agents Interaction Platform

Native Android Application

Kotlin - Jetpack Compose - Clean Architecture - MVI

Authors

Louis Grignola

Amaury Allemand

GitHub: https://github.com/grignolalouis/Ora-mobile/

Contents

List of Figures

List of Tables

1 Functional Specifications

1.1 Project Information

Property	Value
Project name	Ora
Team members	Louis Grignola, Amaury Allemand
GitHub Frontend	https://github.com/grignolalouis/Ora-mobile/

Table 1.1: General project information

1.2 Project Description

1.2.1 Context

Ora is a platform designed for interacting with conversational AI agents. The project is composed of two distinct components: a mobile frontend developed as a native Android application in Kotlin, which is the subject of this report, and a backend API built in Go following a hexagonal architecture pattern.

The mobile application serves as the primary interface for end users to communicate with various AI agents. It provides a seamless chat experience with real-time response streaming, enabling natural and fluid conversations with artificial intelligence systems.

1.2.2 Vision

The primary objective of Ora is to provide a maintainable and scalable boilerplate for building AI agent interaction systems. The platform has been architected with several key principles in mind.

First, the system is designed to be **agnostic**, meaning it can work with different types of agents including LLMs, specialized assistants, and custom AI implementations. Each agent operates within its own isolated scope without interfering with others, ensuring a **modular** architecture that promotes clean separation of concerns.

The platform offers **flexibility** in agent definition, allowing developers to create agents ranging from simple rule-based systems to complex multi-step reasoning engines. Finally, the entire architecture has been designed with **scalability** in mind, enabling the system to evolve according to business requirements without requiring fundamental restructuring.

1.2.3 Target Audience

The Ora platform addresses the needs of multiple user segments. Software developers seeking to integrate AI agents into their applications will find Ora provides a robust foundation with well-defined patterns and abstractions. Organizations looking for a customizable AI chat solution can leverage the platform's extensible architecture to build tailored experiences. End users benefit from an intuitive interface that makes interacting with virtual assistants both accessible and enjoyable.

1.2.4 Main Features

The application delivers a comprehensive set of features designed to provide a complete AI chat experience. The authentication system supports user registration, login, and session management through JWT tokens, ensuring secure access to the platform.

Users can browse a catalog of available agents, each with its own description and capabilities. The real-time chat functionality implements SSE Streaming, allowing responses to appear progressively as they are generated by the AI agent.

Conversation history is persisted per agent, enabling users to continue previous discussions or review past interactions. The user profile section provides account management capabilities including profile picture customization and preference settings. The application supports both light and dark themes, with the interface available in three languages: English, French, and Spanish.

Feature	Description
Authentication	Registration, login, session management with JWT
Agent catalog	List of available agents with descriptions
Real-time chat	Conversation with SSE Streaming responses
History	Conversation persistence by agent
User profile	Account management, profile picture, preferences
Theme	Light/dark mode support
Multilingual	Interface in 3 languages (EN, FR, ES)

Table 1.2: Main application features

1.3 Global Architecture

The Ora system follows a client-server architecture where the mobile frontend communicates with a Go backend through HTTPS requests and SSE connections for real-time streaming.

The backend implements a hexagonal architecture using the tRPC Agent framework, which allows for clean separation between the domain logic and external adapters. Multiple agents can be deployed within the backend, each operating independently with its own configuration and capabilities.

The persistence layer consists of PostgreSQL for relational data storage, Redis for caching and session management, and Minio for file storage including user profile pictures. Phoenix is used for distributed tracing and observability, providing insights into system performance and behavior.

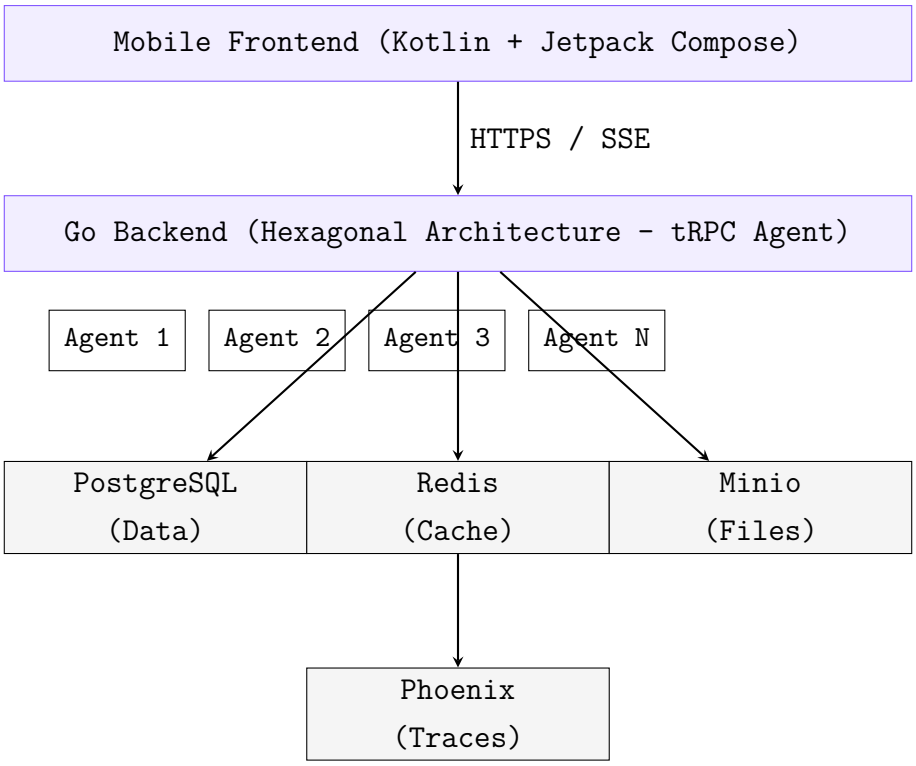


Figure 1.1: Ora system global architecture

1.4 Use Case Diagram

The use case diagram illustrates the main interactions between users and the Ora application. Users must first authenticate before accessing the core features of the platform. Once authenticated, they can browse available agents, select one to interact with, and engage in conversations.

The chat functionality encompasses sending messages, viewing Streaming responses in real-time, accessing conversation history, and managing sessions. Profile management allows users to customize their experience through avatar uploads, theme selection, and language preferences.

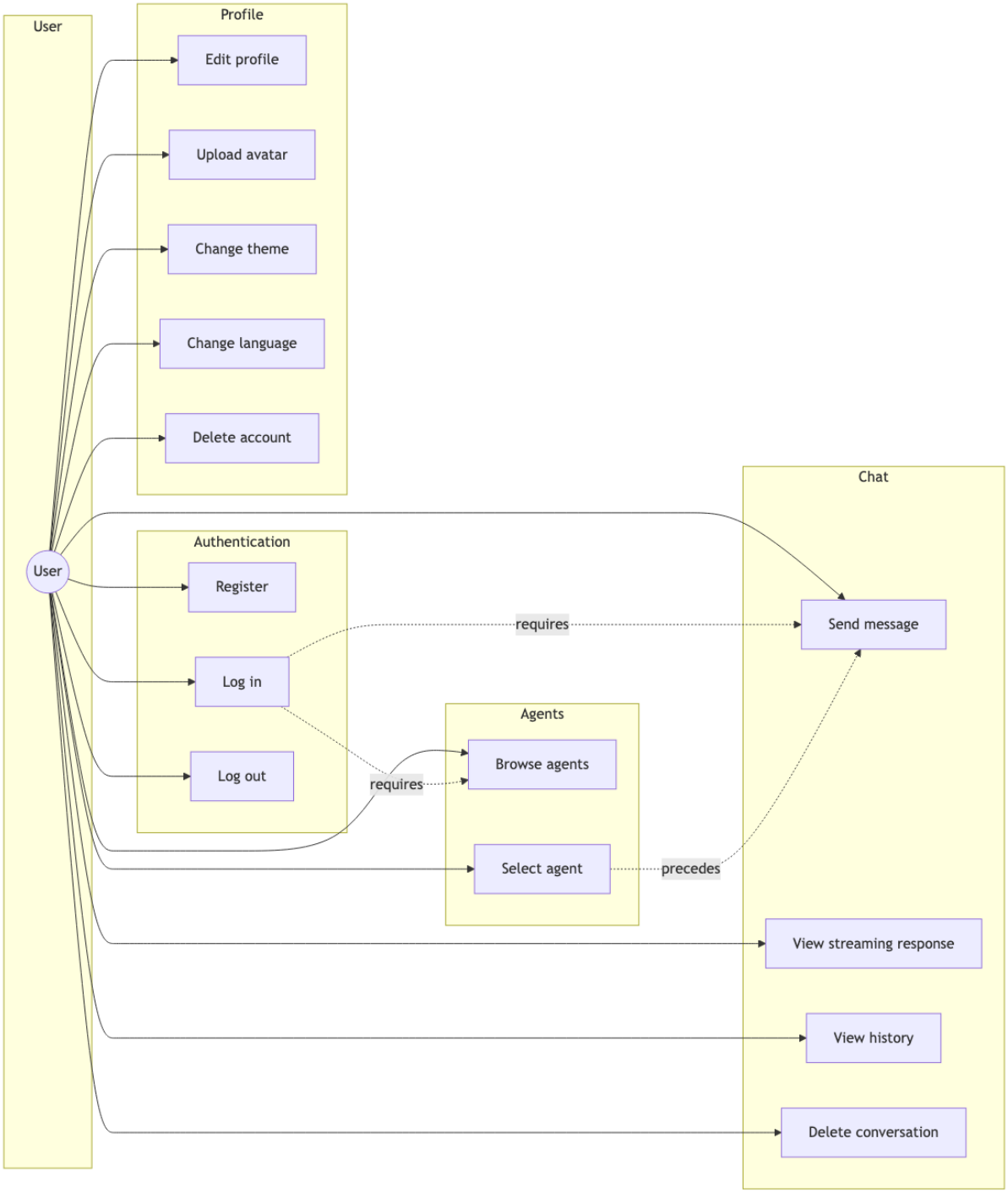


Figure 1.2: Use case diagram showing user interactions with the Ora platform

1.5 Class Diagram

The complete class diagram contains over 100 classes spanning all architectural layers. Due to its size, a simplified version focusing on core domain entities is presented below.

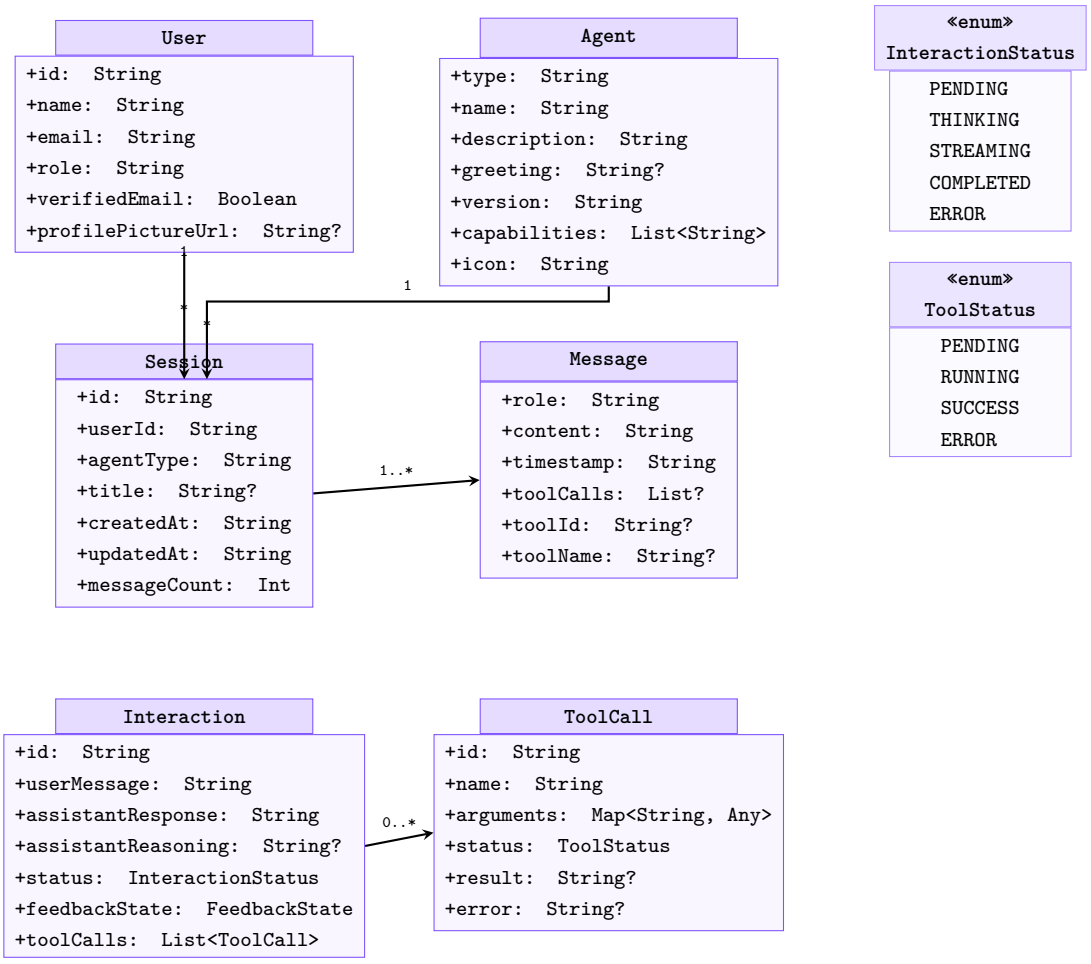


Figure 1.3: Simplified class diagram - Domain entities

The domain layer defines the fundamental entities: User represents authenticated users, Agent encapsulates AI agent metadata, Session tracks conversation instances, and Message stores chat messages with optional tool call information.

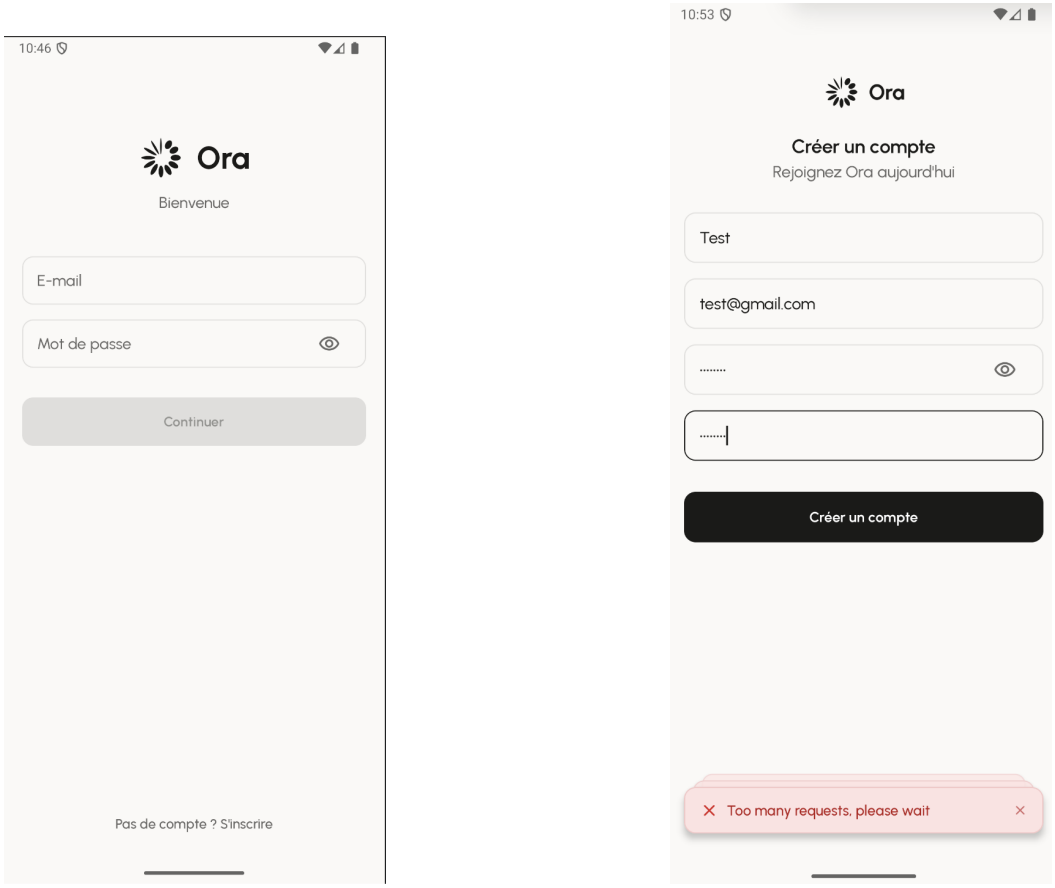
The Interaction class represents a complete exchange between user and assistant, including any intermediate tool calls. Repository interfaces (AuthRepository, AgentRepository, SessionRepository) define data access contracts following the dependency inversion principle.

The presentation layer implements MVI through dedicated ViewModels: AuthViewModel, ChatViewModel, and UserProfileViewModel.

1.6 Screenshots

1.6.1 Authentication

The authentication screens provide a clean and intuitive interface for user onboarding. The login screen presents email and password fields with the Ora logo prominently displayed, while the registration screen includes additional fields for username and password confirmation. Both screens implement real-time validation, providing immediate feedback when users enter invalid data.

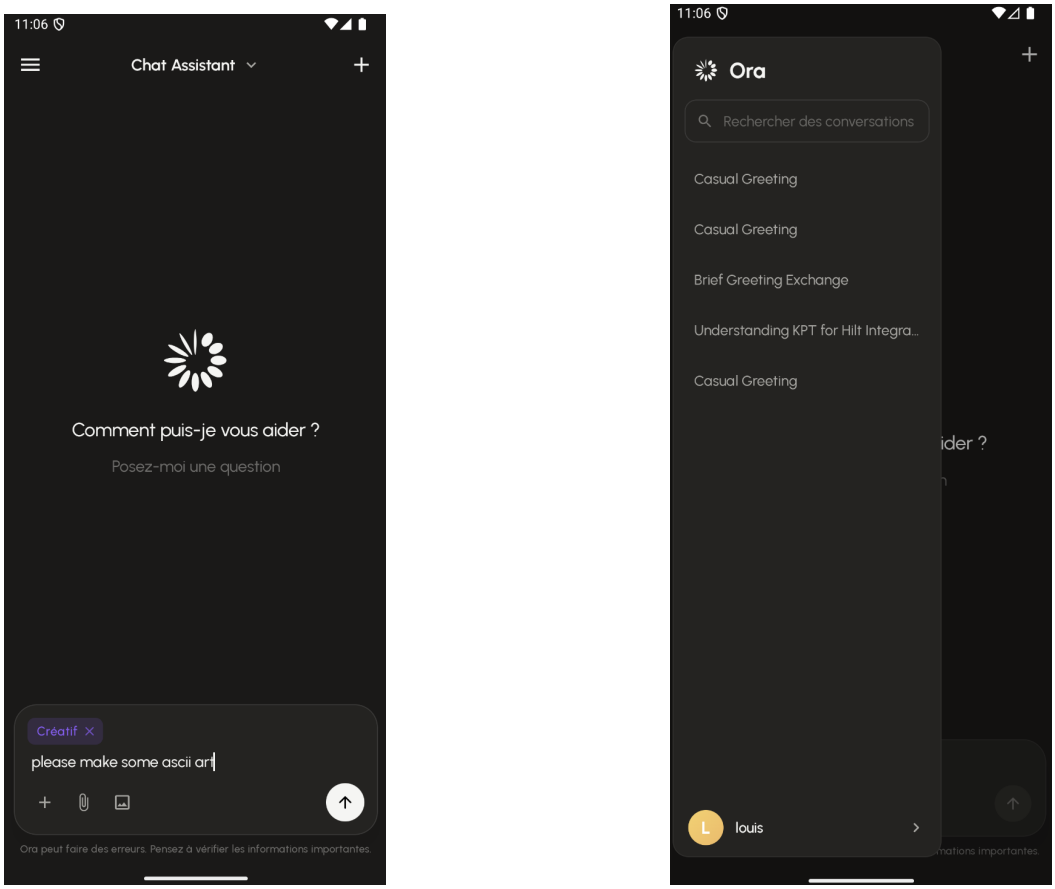


(a) Login screen (b) Registration screen

Figure 1.4: Authentication screens with real-time field validation

1.6.2 Home Page and Navigation

The home screen displays the agent catalog as a collection of cards, each showing the agent's name, description, and icon. Users can browse available agents and select one to begin a conversation. The sidebar navigation drawer provides access to conversation history and allows users to quickly switch between previous sessions or navigate to profile settings.



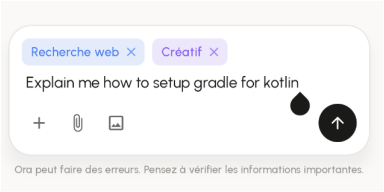
(a) Agent catalog (b) Side menu (drawer)

Figure 1.5: Main application navigation

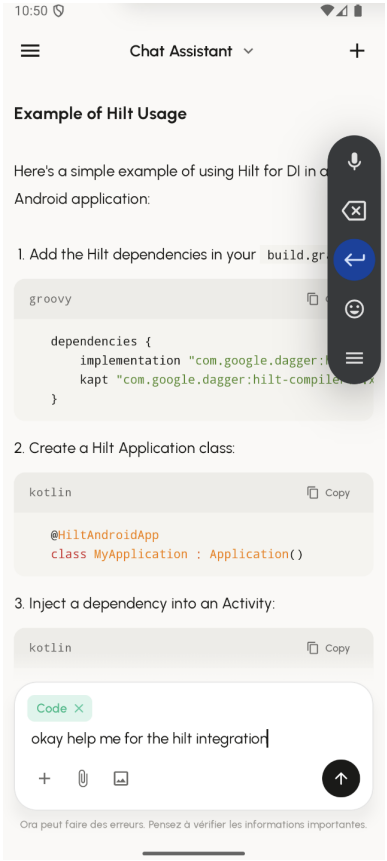
1.6.3 Chat Interface

The chat interface is the core feature of the Ora application. The message input area at the bottom of the screen provides a text field with a send button, optimized for quick and intuitive message composition. User messages appear on the right side of the conversation view, while agent responses are displayed on the left.

The application supports full Markdown rendering including syntax highlighting for code blocks. This is particularly important for technical agents that may provide code examples or technical documentation in their responses.



(a) Message input area



(b) Conversation with agent

Figure 1.6: Chat interface with Markdown rendering and syntax highlighting

During request processing, the application displays a thinking indicator that informs users the agent is analyzing their message. This visual feedback is essential for maintaining a responsive feel even when complex reasoning operations are being performed on the backend.

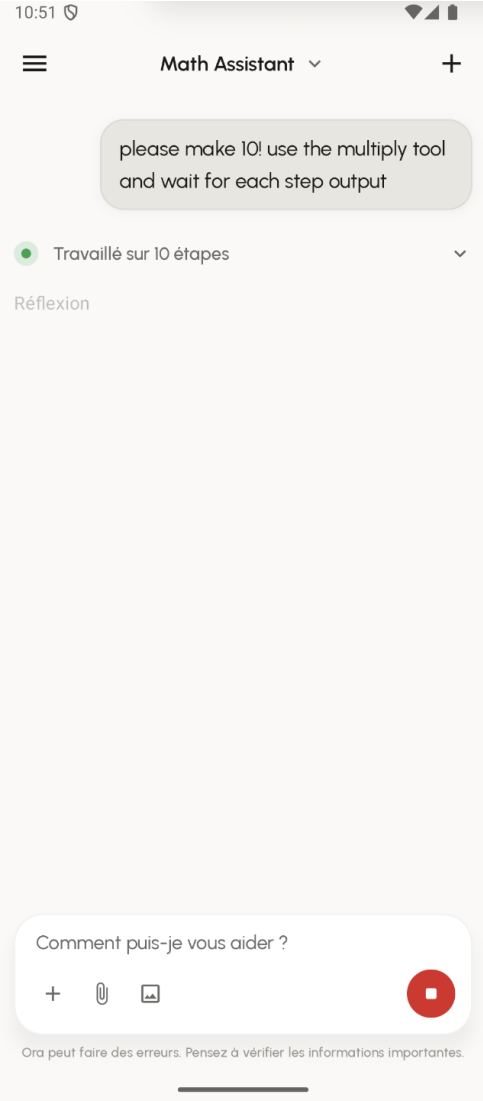
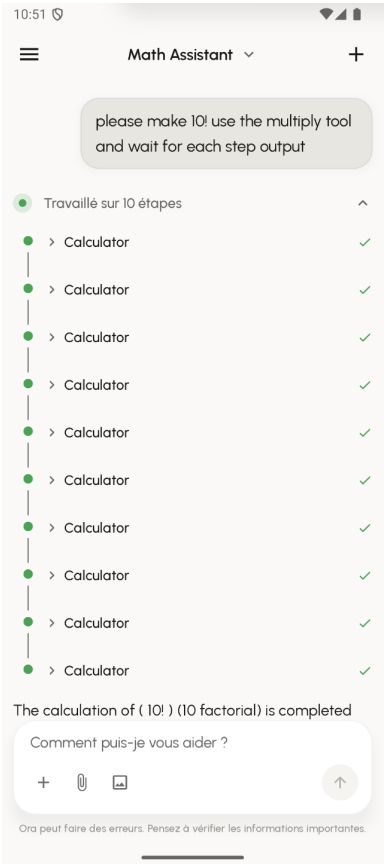


Figure 1.7: Thinking indicator during request processing

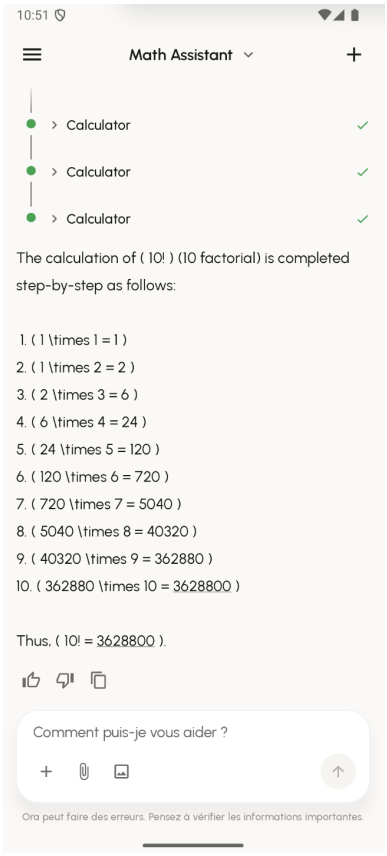
1.6.4 Tool Calls

One of the distinguishing features of Ora is its support for agent tool calls. When an agent needs to perform actions such as searching for information, executing code, or accessing external services, these operations are displayed transparently to the user.

The tool call interface shows which tools are being invoked along with their parameters, allowing users to understand exactly what actions the agent is taking on their behalf. Once tool execution completes, the results are integrated into the conversation flow alongside the agent's final response.



(a) Tool execution



(b) Result with response

Figure 1.8: Display of tool calls made by the agent

1.6.5 User Profile

The user profile screen centralizes all account management and preference settings. Users can view and edit their personal information, upload a custom profile picture, and customize the application experience through theme and language selection. Account actions including logout and account deletion are also accessible from this screen.

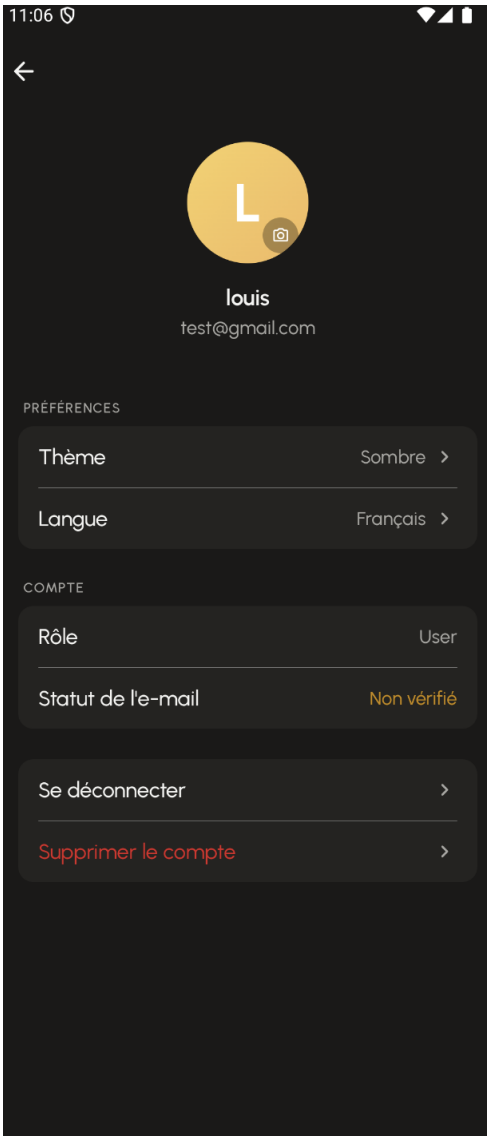


Figure 1.9: Profile page with settings (theme, language, account)

1.7 Technical Challenges Overcome

The development of Ora presented several significant technical challenges that required careful consideration and innovative solutions. These challenges primarily arose from the need to provide a seamless real-time chat experience while maintaining clean architecture principles.

1.7.1 Real-time SSE Streaming

The most complex challenge involved implementing SSE Streaming for AI responses. Unlike traditional HTTP requests that return complete responses, SSE requires maintaining a persistent connection and processing

data as it arrives token by token.

The Android lifecycle adds additional complexity, as connections must survive configuration changes like screen rotations while being properly cleaned up when the activity is destroyed. Events arrive on IO threads and must be dispatched to the main thread for UI updates without blocking the interface during rapid event bursts.

The solution implements a buffering mechanism with implicit debouncing, combined with Kotlin Flow operators like conflate() to handle backpressure when events arrive faster than the UI can process them.

1.7.2 Custom Markdown Rendering

Jetpack Compose does not provide a native Markdown component, requiring a hybrid approach using the Markwon library within an AndroidView. This creates interoperability challenges between the Compose theming system and traditional Android Views.

Syntax highlighting for code blocks across 13 programming languages was implemented through a custom SyntaxHighlighter with regex-based parsing. To address performance concerns with large code blocks, the solution employs lazy parsing and an LRU cache to memoize results.

1.7.3 MVI Architecture with Streaming State

The standard MVI pattern assumes discrete states with atomic transitions, but Streaming introduces a continuously evolving state that updates multiple times per second. The solution uses composite states with independent sub-states, allowing the streaming content to be updated without triggering unnecessary recomposition of unrelated UI elements.

Challenge	Complexity	Solution
Real-time SSE Streaming	High	Buffering with debounce, StateFlow with conflate
Custom Markdown rendering	High	Markwon + custom SyntaxHighlighter with LRU cache
MVI architecture streaming	Medium	Composite states with independent sub-states
Auth with refresh token	Medium	AuthInterceptor with Mutex to avoid race conditions
OkHttp/Ktor integration	Medium	Shared Hilt module, common factory

Table 1.3: Summary of major technical challenges

2 Technical Specifications

2.1 Architecture

2.1.1 Clean Architecture

Ora follows Clean Architecture principles to ensure separation of concerns, testability, and maintainability. The codebase is organized into four distinct layers, each with clear responsibilities and dependencies flowing inward.

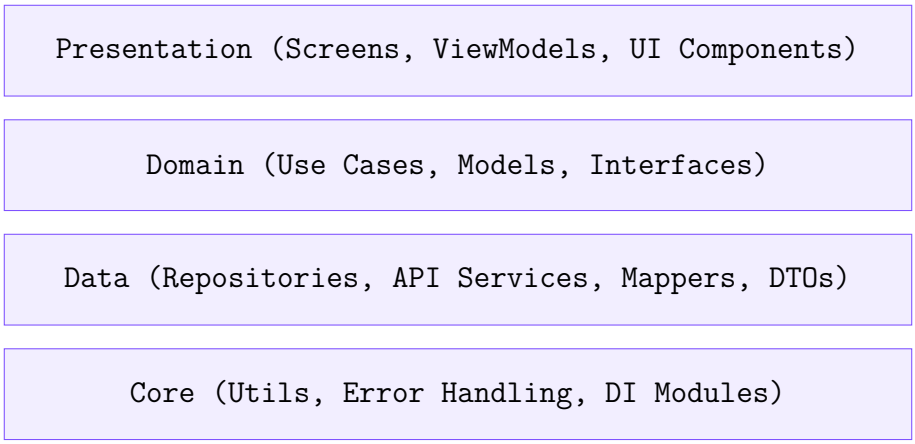


Figure 2.1: Clean Architecture layers in Ora

The **Core Layer** contains utilities, constants, error handling mechanisms, and dependency injection modules. This layer has no dependencies on other layers and provides foundational components used throughout the application.

The **Data Layer** implements the repository interfaces defined in the domain layer. It handles all external data operations including API calls, local storage, and data mapping. DTOs are converted to domain models through dedicated mapper classes, ensuring the domain layer remains independent of external data representations.

The **Domain Layer** represents the business logic of the application. It defines the core models (User, Agent, Session, Message), repository interfaces, and use cases. This layer is completely independent of frameworks and external libraries, making it highly testable and portable.

The **Presentation Layer** handles all UI-related concerns using Jetpack Compose. ViewModels manage UI state and coordinate with use cases to perform business operations. The layer implements the MVI pattern for predictable state management.

2.1.2 MVI Pattern

The application uses the Model-View-Intent (MVI) pattern for state management in the presentation layer.

Why MVI over MVVM?

MVI was chosen over MVVM for several technical reasons specific to Ora's requirements.

The first consideration is **unidirectional data flow**. In a chat application with real-time streaming, predictable state changes are critical. MVI enforces a single direction: Intent → Model → View, eliminating ambiguity about how and when state changes occur. MVVM's bidirectional binding can lead to unpredictable state updates when handling rapid SSE events.

The second reason is **state immutability**. MVI uses immutable state objects, making it trivial to compare previous and current states. This is essential for optimizing Compose recomposition during streaming, where content updates multiple times per second.

The third advantage is **debugging and reproducibility**. Every state transition in MVI is triggered by an explicit Intent. This creates a clear audit trail, making it easier to debug issues in complex flows like tool call sequences or authentication refreshes.

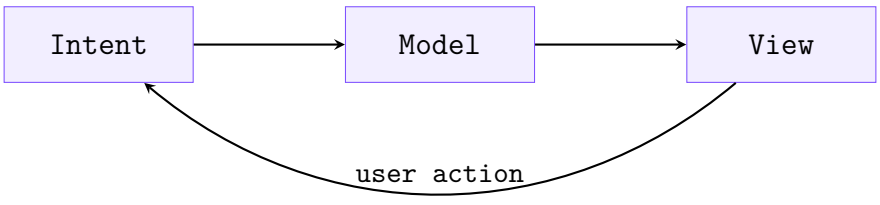


Figure 2.2: MVI unidirectional data flow

2.2 API Interaction

2.2.1 Authentication Endpoints

Endpoint	Method	Description	Returns
/auth/login	POST	Authenticates user with email/password	User + token
/auth/register	POST	Creates new user account	User + token
/auth/logout	POST	Invalidates current session	Status
/auth/refresh	POST	Refreshes expired access token	New token
/auth/me	GET	Retrieves current user info	User profile
/auth/me	DELETE	Deletes user account	Status
/auth/me/profile-picture	POST	Uploads profile picture	Picture URL

Table 2.1: Authentication API endpoints

2.2.2 Agent Endpoints

Endpoint	Method	Description	Returns
/agents	GET	Lists all available agents	Agent array
/agents/{type}	GET	Gets specific agent info	Agent details

Table 2.2: Agent API endpoints

2.2.3 Session Endpoints

Endpoint	Method	Description	Returns
/agents/{type}/sessions	GET	Lists user sessions	Session array
/agents/{type}/sessions	POST	Creates new session	Session ID
/agents/{type}/sessions/{id}	GET	Gets session + history	Messages
/agents/{type}/sessions/{id}	DELETE	Deletes session	Status
/agents/{type}/sessions/{id}/message	POST	Sends message	Stream ID

Table 2.3: Session API endpoints

2.3 SSE Streaming

2.3.1 Overview

Server-Sent Events (SSE) is used for real-time streaming of AI responses. Unlike WebSockets, SSE provides a simpler unidirectional protocol perfectly suited for the chat use case where only the server needs to push data to the client.

2.3.2 Connection Flow

The streaming process follows a specific sequence:

1. Client sends message via POST to /agents/{type}/sessions/{id}/message
2. Server returns a stream ID
3. Client opens SSE connection to /stream/{streamId}
4. Server pushes events as AI generates response
5. Connection closes on "done" or "close" event

2.3.3 Event Types

Event	Description
delta	Incremental content token (streaming mode enabled)
message	Complete message in one block (streaming mode disabled)
reasoning	AI reasoning/thinking content
tool_call	Agent initiating tool execution
tool_response	Result from tool execution
thinking_start	AI begins reasoning phase
thinking_end	AI completes reasoning phase
error	Error during processing
done	Stream completed successfully
close	Connection should be closed
heartbeat	Keep-alive signal

Table 2.4: SSE event types

2.3.4 Implementation

SSE is implemented using OkHttp's EventSource client rather than Ktor, as Ktor does not natively support SSE. The SSEClient class manages connection lifecycle, event parsing, and error handling. Events are parsed by SSEEventMapper and converted to sealed StreamEvent classes, enabling type-safe handling in the ViewModel.

2.4 Markdown Rendering

2.4.1 Architecture

Markdown rendering combines the Markwon library with custom syntax highlighting. Since Jetpack Compose lacks native Markdown support, rendering is performed in an AndroidView wrapper containing a TextView.

2.4.2 Components

MarkdownText is a Composable wrapper that creates and configures the Markwon instance, handles theme changes, and bridges Compose with Android Views.

CodeBlock is a custom component for displaying code blocks with syntax highlighting, language label, and copy-to-clipboard functionality.

SyntaxHighlighter is a custom regex-based highlighter supporting 13 languages. While Prism4j is available in dependencies, a custom implementation was chosen to avoid annotation processing complexity and provide finer control over the Gruvbox color scheme.

2.4.3 Supported Languages

Kotlin, Java, Python, JavaScript, TypeScript, Go, Rust, C, C++, Swift, SQL, JSON, XML/HTML.

2.5 Libraries

Library	Version	Purpose
Ktor Client	3.0.2	HTTP client for REST API (Kotlin-first, coroutines)
OkHttp SSE	4.12.0	SSE client for real-time streaming
Hilt	2.58	Dependency injection (compile-time, ViewModels)
Kotlinx Serialization	1.7.3	JSON serialization (no reflection, type-safe)
Jetpack Compose	BOM 2025.01.00	Declarative UI framework
DataStore	1.1.1	Preferences storage (coroutines, type-safe)
Security Crypto	1.1.0-alpha06	Encrypted token storage (Keystore)
Coil	3.0.4	Image loading (Compose-native, Ktor)
Markwon	4.6.2	Markdown rendering (extensible, tables)
Coroutines	1.9.0	Async programming (Flow, structured)

Table 2.5: Main libraries and their purposes

2.5.1 Library Selection Criteria

Each library was selected based on the following criteria:

Kotlin-first: Native Kotlin API without Java wrappers, leveraging language features like coroutines, sealed classes, and extension functions.

Coroutine support: Native integration with suspend functions and Flow for reactive programming.

Maintenance: Active development, regular updates, and strong community support.

Performance: Minimal impact on APK size and runtime performance.

2.6 Testing

2.6.1 Testing Strategy

The application follows a comprehensive testing strategy organized in four tiers based on criticality:

- **Tier 1 - Critical:** Security-sensitive components and complex parsing logic (SSE events, session reconstruction, authentication interceptor)
- **Tier 2 - High Priority:** Business logic in use cases that orchestrate application behavior
- **Tier 3 - Medium Priority:** Data layer components including mappers and repository implementations
- **Tier 4 - Lower Priority:** Presentation layer ViewModels with UI state management

2.6.2 Test Libraries

Library	Version	Purpose
JUnit	4.13.2	Test framework
MockK	1.13.10	Kotlin mocking library
Truth	1.4.2	Fluent assertions
Turbine	1.1.0	Flow testing utilities
Coroutines Test	1.9.0	Coroutine testing

Table 2.6: Testing libraries

2.6.3 Test Coverage

Layer	Components	Test Files	Tests
Core	Utils, Error Mapping, Network	4	30
Domain	Use Cases, Models	14	72
Data	Mappers, Repository	5	69
Presentation	ViewModels	3	48
Total	-	**26**	**219**

Table 2.7: Test coverage by layer

2.6.4 Test Structure

Tests are organized mirroring the source code structure, ensuring clear correspondence between implementation and tests:

- core/ - ResultTest, DateTimeUtilTest, ErrorMapperTest, AuthInterceptorTest
- data/mapper/ - SSEEventMapperTest, SessionMapperTest, UserMapperTest, AgentMapperTest
- data/repository/ - AuthRepositoryImplTest
- domain/model/ - SessionInteractionsTest
- domain/usecase/auth/ - Login, Register, GetCurrentUser, Logout, DeleteAccount, UploadProfilePicture
- domain/usecase/agent/ - GetAgentsUseCaseTest

- domain/usecase/session/ - GetSessions, CreateSession, DeleteSession, GetSessionHistory, SendMessage, StreamResponse
- presentation/features/ - AuthViewModelTest, ChatViewModelTest, UserProfileViewModelTest

2.6.5 Critical Tests

SSEEventMapperTest (39 tests) validates parsing of all 13+ SSE event types including delta content accumulation, tool call/response structures, error handling with codes, and graceful degradation for malformed JSON.

SessionInteractionsTest (15 tests) tests the complex algorithm that reconstructs user-assistant interaction pairs from raw message history, including tool call matching, pending status detection, and metadata conversion.

AuthInterceptorTest (12 tests) verifies token injection for authenticated requests, public endpoint bypassing, 401 response handling, and concurrent refresh prevention using atomic flags.

2.6.6 Testing Patterns

Use Case Tests: Each use case is tested with mocked repository dependencies. Tests verify input validation, successful execution paths, error propagation, and correct repository method invocation.

ViewModel Tests: ViewModels are tested using UnconfinedTestDispatcher for synchronous execution. Turbine is used to test Flow emissions (effects) such as navigation events and toasts.

Mapper Tests: Data mappers are tested to ensure correct field mapping between DTOs and domain models, with special attention to nullable fields and nested objects.

2.7 Future Evolution

The Ora platform has been designed with extensibility in mind, and several enhancements are planned for future development phases. This section presents the two major features on the roadmap along with additional improvements.

2.7.1 Multimodal Support

The current architecture already supports multimodal interactions at the backend level, with existing endpoints for attachment uploads that are not yet integrated into the mobile application. The next major release will enable users to enrich their conversations with images, documents, and audio content.

Users will be able to attach images directly to their messages, providing visual context that agents can analyze and reference in their responses. Document support will include PDF, Word, and plain text formats, allowing users to share technical documentation or reference materials during conversations. Voice input through audio recording will offer an alternative interaction mode, with server-side transcription converting speech to text before processing.

From a technical perspective, this feature requires implementing an AttachmentRepository to handle file upload and download operations, adding multipart form data support to the Ktor client, and creating new UI components for attachment selection and preview. File compression and validation will ensure optimal performance and security.

2.7.2 Knowledge Base Management

One of the most powerful upcoming features is the exposure of the vector knowledge base to end users. The backend already maintains a dedicated vector database for each agent and user, enabling contextual retrieval during conversations. Future versions will provide a complete interface for users to manage their personal knowledge base.

Users will be able to upload documents that become part of their personal context, viewable and searchable through an in-app browser. When agents reference information from the knowledge base during responses, the interface will display which documents contributed to the answer, providing transparency into the retrieval process. Bulk import and export operations will facilitate migration and backup of knowledge bases.

Method	Endpoint	Description
GET	/knowledge/documents	List all documents in user's knowledge base
POST	/knowledge/documents	Upload a new document
GET	/knowledge/documents/{id}	Retrieve document content and metadata
DELETE	/knowledge/documents/{id}	Remove document from knowledge base
GET	/knowledge/search	Search documents by semantic similarity

Table 2.8: Planned knowledge base API endpoints

The implementation will leverage the existing server-side embedding pipeline for vector generation. Document chunking strategies will need to balance retrieval precision with context coherence. Preview rendering will require format-specific handlers, such as PDF.js for PDF documents and the existing Markwon renderer for markdown files. Storage quotas and document size limits will be defined to ensure fair resource allocation.

2.7.3 Additional Improvements

Beyond these major features, several quality-of-life improvements are planned. Offline mode will cache recent conversations locally using Room database, allowing users to browse their history without network connectivity. Push notifications will alert users when agents complete long-running tasks or when collaborative sessions receive new messages.

Collaborative sessions will enable users to share conversations with colleagues, facilitating team workflows around AI interactions. Agent customization will allow power users to define custom system prompts and adjust model parameters for specific use cases. Finally, an analytics dashboard will provide insights into usage patterns, helping users understand their interaction history and optimize their workflows.

Glossary

API Application Programming Interface - Interface allowing applications to communicate with each other. 1

Flow Kotlin type representing an asynchronous data stream. 13

JWT JSON Web Token - Standard for creating secure access tokens. 2, 3

LLM Large Language Model - Large-scale language model used for text generation. 1

MVI Model-View-Intent - Architectural pattern for reactive applications. 6, 13, 14

SSE Server-Sent Events - Technology enabling the server to push data to the client via a persistent HTTP connection. 2, 3, 12, 14, 18

Streaming Continuous data transmission allowing progressive display of responses. 2-4, 12-14