Contract Central
Technical Documentation

M1 Software Development & Big Data – ISEN Toulon
Academic Year: 2024–2025

Team Members:
Capucine Debailleul (Project Manager)
Alexis Dutaud
Fabio Sintoni
Larry Jason Tueno

Supervised by Thomas Dandelot and Tony Varnier

Contents

1 Introduction 3
1.1 Overall Architecture 4

2 Microservice "Data Management" 5
2.1 Overview 5
2.2 Main Features 5
2.3 Dependencies 5
2.3.1 Service Dependencies 6
2.3.2 Dev Dependencies 7
2.4 User Module 7
2.5 Firebase Module 9
2.6 File Module 10
2.7 VertexAI Module 12
2.8 Dashboard Module 14
2.9 PubSub Module 16
2.10 VertexAI Module 17
2.11 Auth Module 19

3 Microservice "Document Analysis" 22
3.1 Overview 22
3.2 Main Features 22
3.3 Dependencies 22
3.3.1 Libraries Used 22
3.3.2 Custom Modules 23
3.4 Architecture and Workflow 23
3.4.1 File Upload 23
3.4.2 Coverage Query 23
3.4.3 File Deletion 23
3.5 Endpoints (For Debugging Purpose ONLY) 24
3.6 Testing 24
3.6.1 Unit Tests 24
3.6.2 Example: 24
3.7 CI/CD Pipeline 24
3.7.1 Automated Tests and Linting 24
3.7.2 Cloud Run Deployment 25
3.8 Deployment 25
3.9 Security 25
3.10 Reactive Use Case Sequence Diagram 25

4 Frontend Architecture and User Manual **27**

4.1 Technology Choices . . . 27

4.1.1 Flutter . . . 27

4.1.2 Dart . . . 27

4.1.3 Provider (State Management) . . . 27

4.1.4 flutter_markdown . . . 27

4.1.5 Syncfusion Charts . . . 27

4.1.6 Firebase Auth . . . 27

4.2 Workflow Overview . . . 28

4.3 Project Structure . . . 28

4.4 Component Highlights . . . 28

4.5 User Manual . . . 29

Conclusion **38**

Chapter 1

Introduction

Contract Central is a cloud-based mobile and web application designed to help users manage and understand their insurance-related documents. The platform leverages microservices, AI-powered document analysis, and semantic search to provide intuitive and intelligent access to contract information.

The key goals of Contract Central are:

- To centralize the storage of administrative and insurance documents in a secure personal space.
- To assist users in understanding their contracts through natural language interaction.
- To detect redundant or overlapping coverage across different contracts.
- To answer common questions such as "Am I covered in case of theft?" or "What services are included with my credit card?"

The system is composed of multiple interconnected microservices, each responsible for a specific task in the processing pipeline. These services communicate asynchronously using Google Pub/Sub and are hosted on Google Cloud Platform.

The backend is divided into two main services:

- **Data Management**: handles user authentication, file uploads, and Pub/Sub message orchestration.
- **Document Analysis**: processes files using Google Vertex AI and ChromaDB to enable semantic understanding and search.

On the client side, the application is developed with Flutter, allowing it to run seamlessly on both Android and iOS devices, as well as the web. The user interface supports uploading contracts, consulting AI-generated summaries, and querying coverage information via a conversational assistant.

By combining structured document storage, machine learning models, and a modern frontend, Contract Central aims to reduce the complexity of insurance management for everyday users.

1.1 Overall Architecture

The diagram below illustrates the overall architecture of Contract Central. It highlights the main components of the system and how they interact: the mobile/web frontend built with Flutter, the backend services running on Google Cloud, the use of Pub/Sub for asynchronous communication, and the integration of AI services such as Gemini and ChromaDB.

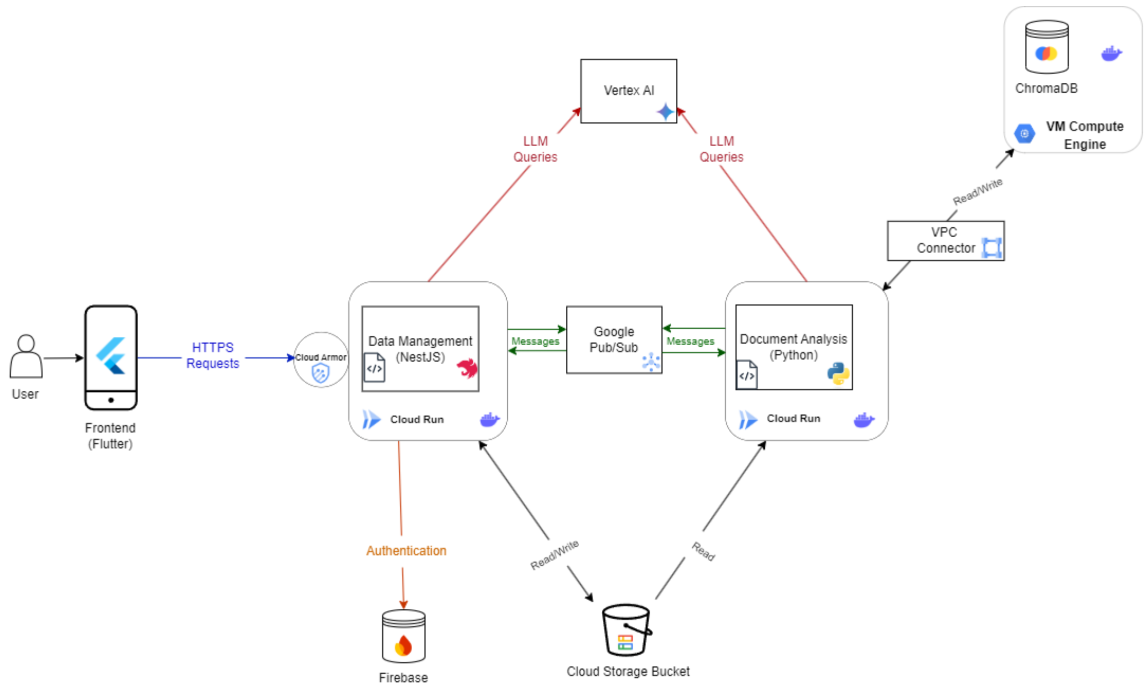


Figure 1.1: High-level architecture of the Contract Central platform

Chapter 2

Microservice "Data Management"

2.1 Overview

Data Management is the main backend service of the system. It integrates several key modules.

2.2 Main Features

- **Authentication** using Firebase Auth
- **Firebase** in a GCP Cloud Storage bucket
- **VertexAI** with Gemini pro 1.5
- **Pub/Sub** messaging to communicate with DocAnalysis
- **Dashboard** for user to manage their documents
- **User** for user management

2.3 Dependencies

This section outlines the dependencies required for the backend service, categorized into service dependencies and development dependencies. Each dependency is listed with its purpose to provide clarity on its role within the project. The backend service is built using the NestJS framework and integrates with various Google Cloud services, including Firebase and Vertex AI. The dependencies are managed using npm and are specified in the `package.json` file.

2.3.1 Service Dependencies

Package	Purpose
@google-cloud/aiplatform	Vertex AI SDK for ML requests
@google-cloud/pubsub	Google Pub/Sub messaging system
@google-cloud/secret-manager	Secure storage and access to secrets
@google-cloud/vertexai	Vertex AI high-level client (simplified)
@nestjs/common	NestJS decorators and helpers
@nestjs/config	Environment configuration management
@nestjs/core	NestJS core engine
@nestjs/microservices	Microservice architecture support
@nestjs/platform-express	Express integration with NestJS
@nestjs/swagger	OpenAPI documentation generator
@types/multer	Type definitions for multer
class-transformer	DTO class serialization/deserialization
class-validator	DTO input validation
firebase-admin	Firebase Admin SDK (Auth, Firestore, GCS)
multer	Middleware for file uploads
nestjs-pino	Structured logging with Pino in NestJS
pino	High-performance JSON logger
reflect-metadata	Required for decorators/reflection
rxjs	Reactive programming and Observables
swagger-ui-express	Swagger UI middleware for Express

2.3.2 Dev Dependencies

Package	Purpose
@nestjs/cli	NestJS project scaffolding and commands
@nestjs/schematics	Code generation templates for NestJS
@nestjs/testing	Utilities for unit and integration tests
@types/express	Type definitions for Express.js
@types/jest	Type definitions for Jest
@types/node	Type definitions for Node.js
@types/supertest	Type definitions for Supertest
@typescript-eslint/eslint-plugin	ESLint rules for TypeScript
@typescript-eslint/parser	Parser to enable ESLint for TS
eslint	Linter for code quality
eslint-config-prettier	Disables rules that conflict with Prettier
eslint-plugin-prettier	Integrates Prettier into ESLint
jest	Testing framework
prettier	Code formatter
source-map-support	Stack trace source map translation
supertest	HTTP assertions for integration tests
ts-jest	Jest preprocessor for TypeScript
ts-loader	TypeScript loader for Webpack
ts-node	Run TypeScript files directly
tsconfig-paths	Support for path aliases in tsconfig
typescript	TypeScript compiler

2.4 User Module

Overview

The **UserModule** manages user-specific metadata and preferences that are not handled by Firebase Auth. It stores and retrieves information such as first name, last name, birthdate, and contact data using Firestore.

Responsibilities

- Create and update user profile information in Firestore.
- Retrieve authenticated user metadata (from Firebase Auth).
- Provide a minimal contact endpoint for support messages (placeholder).

Dependencies

- `Firestore` - Used to store extended user info.
- `AuthService` - Used to fetch Firebase Auth metadata (e.g., display name, email).

Key Service Methods

- `createUserInfo(uid, dto)`: Creates a Firestore document with full name and email pulled from Firebase Auth.
- `updateUserInfo(uid, dto)`: Updates the user info with merge semantics and a timestamp.
- `getUserInfo(uid)`: Returns user info from Firestore, or `null` if not found.

Exposed Routes

POST /user/me

Method	POST
Route	/user/me
Auth required	Yes
Status codes	201 Created, 400 Bad Request, 401 Unauthorized

Description: Creates the Firestore user profile based on the authenticated user's UID and metadata from Firebase Auth.

Payload Example:

```
{
  "birthdate": "2000-01-01"
}
```

PATCH /user/me

Method	PATCH
Route	/user/me
Auth required	Yes
Status codes	200 OK, 400 Bad Request, 401 Unauthorized

Description: Updates user information in Firestore. Uses `merge: true` semantics to preserve existing fields.

Payload Example:

```
{
  "firstname": "Jane",
  "lastname": "Smith",
  "email": "jane@example.com"
}
```

POST /user/contact

Method	POST
Route	/user/contact
Auth required	No
Status codes	200 OK (placeholder)

Description: Endpoint for users to contact admin/support. Currently a stub (to be implemented).

Payload Example:

```
{
  "email": "user@example.com",
  "message": "I need help with..."
}
```

Testing

Unit tests validate:

- Creation of user info from Firebase metadata.
- Updating user info with merge semantics.
- Retrieval of user data (existing and non-existing cases).

Mocks are used for:

- `AuthService.getUserByUid()`
- Firestore collection, doc, set, and get operations

2.5 Firebase Module

Overview

The **FirebaseModule** is a global dynamic module that initializes the Firebase Admin SDK and exposes key Firebase services throughout the application. It centralizes configuration and ensures consistent access to Firebase features like Firestore, Auth, and Cloud Storage.

Responsibilities

- Initialize Firebase Admin with default credentials.
- Provide access to:
  - Firestore (NoSQL database)
  - Firebase Authentication
  - Cloud Storage bucket
- Export these providers globally to be used by other modules (e.g., Auth, Files, Dashboard).

Dynamic Initialization

FirebaseModule.forRoot() returns a `DynamicModule` configured with the following providers:

- **FIREBASE_APP** - Firebase application instance
- **FIRESTORE** - Firestore database instance
- **FIREBASE_AUTH** - Firebase authentication handler
- **FIREBASE_BUCKET** - Google Cloud Storage bucket handler

These tokens are injected via NestJS's dependency injection system.

Global Scope

The module is marked with `@Global()` so it only needs to be imported once and is accessible throughout the entire application without re-importing in each feature module.

Testing

A dedicated unit test verifies that:

- All Firebase services are correctly provided via dependency tokens.
- The module exports all necessary services.

Mocked Firebase services are used in unit tests to avoid external dependency on Firebase during test execution.

Configuration

- **Project ID:** `contract-central-c710c`
- **Storage Bucket:** `contract-central-c710c.firebasestorage.app`
- **Credential:** Application Default Credential

2.6 File Module

Overview

The **FileModule** manages all user file operations including upload, retrieval, deletion, and temporary storage. It integrates Google Cloud Storage via Firebase and uses Pub/Sub to notify other microservices of file-related events.

Responsibilities

- Upload and store user files under structured paths in GCS.
- Generate temporary signed URLs for accessing files.
- Delete specific or all files for a user.
- Notify other services of file upload and deletion via Pub/Sub.

Dependencies

- `@google-cloud/storage` - for GCS bucket access.
- `FirebaseModule` - to inject the GCS bucket.
- `PubSubService` - to publish `file-upload` and `file-delete` events.

Key Service Methods

- `uploadFile(uid, file, dto)` - Uploads a categorized file and emits Pub/Sub event.
- `getFileUrl(uid, fileName)` - Returns a temporary signed URL for a user file.
- `getUserFiles(uid)` - Lists all files uploaded by the user.
- `deleteFile(uid, fname, category)` - Deletes a file and emits a Pub/Sub event.
- `deleteAllUserFiles(uid)` - Deletes all user files and emits a single Pub/Sub event.
- `uploadTmpImage(uid, file, dto)` - Special-purpose method to upload temporary images.
- `deleteTempFile(fpath)` - Deletes a temporary file if it exists.

Exposed Routes

POST /files/upload

Method	POST
Route	/files/upload
Auth required	Yes
Consumes	multipart/form-data

Form fields:

- `file`: PDF file to upload
- `category`: required category label (e.g., `HEALTH`, `EMPLOYMENT`)

GET /files/:fileName

Returns a temporary signed URL for the requested file.

GET /files

Returns a list of all files uploaded by the authenticated user.

DELETE /files/:category/:fileName

Deletes a specific file in a category and emits a `file-delete` event.

DELETE /files

Deletes all files for the user and emits a single `file-delete` event.

Error Handling

- **400 Bad Request** - Invalid or missing category.
- **404 Not Found** - Requested file does not exist.
- **500 Internal Server Error** - Upload/delete/signing failures.

Testing

Unit tests validate:

- File upload with event publishing.
- Temporary signed URL generation.
- Accurate file listing.
- Conditional deletion (specific or all files).
- Handling of missing files and errors.

Mocks are used for:

- GCS Bucket and File objects.
- Pub/Sub message publishing.

2.7 VertexAI Module

Overview

The **VertexAI Module** integrates the Google Cloud Vertex AI API into the backend using the `@google-cloud/vertexai` client. It handles AI-driven reasoning and recommendation tasks by interacting with Gemini 1.5 Pro.

Responsibilities

- Load a generative model (Gemini 1.5 Pro) via the Vertex AI SDK.
- Process user prompts and file references (PDF, images).
- Generate structured reasoning and actionable recommendations.
- Log AI interactions to Firestore for traceability.

Workflow Summary

The method `generateTextContent` executes two chained LLM calls:

1. **Structured reasoning generation** based on:
    - User metadata (`firstname`, `lastname`, etc.)
    - Attached files from Firebase Storage
    - Custom domain-specific reasoning prompt
2. **Final decision generation** based on the AI's reasoning output.

Results are logged in Firestore under the `logs/{uid}/ai_interactions` collection with full metadata.

Prompt Logic

- Prompts are composed dynamically using the user's identity and structured business rules.
- **Reasoning Prompt:** guides the model to extract risk analysis and contract redundancies.
- **Final Decision Prompt:** asks the model to produce an actionable summary.
- All file attachments are referenced using GCS URIs (e.g., `gs://bucket/path`).

Key Method: `generateTextContent`

- **Inputs:** user ID, prompt string, list of Firebase Storage URLs, user metadata.
- **Output:** response object from Vertex AI's LLM (Gemini 1.5 Pro).
- **Side-effects:**
    - AI interaction logs written to Firestore.
    - Token usage, model version, and response IDs tracked.

Firestore Logging Structure

Each interaction creates or updates a Firestore document with:

- `prompt`, `reasoning`, `finalDecision`
- `vertexResponse` and `finalVertexResponse` metadata
- `createdAt`, `deletedAt`

Error Handling

- If reasoning or decision generation fails, the error is logged and rethrown.
- GCS URI generation fallback uses MIME type detection to support different file types.

Testing

Unit tests validate:

- Proper generation of both reasoning and final decision via mocked LLM.
- Firestore logs are created and updated accordingly.
- Token usage, model version, and response IDs are extracted correctly.

Mocks are used for:

- VertexAI client and model generation responses.
- Firestore document creation and update.

2.8 Dashboard Module

Overview

The **Dashboard** module provides a synthetic view of a user's activity and data. It aggregates file statistics and coverage request history from Firestore and returns a structured dashboard summary.

Responsibilities

- Aggregate file statistics per user from Cloud Storage.
- Count and classify AI coverage requests (pending / completed).
- Return a structured summary to the frontend.

Dependencies

- `FileService` - for retrieving user files from storage.
- `Firestore` - for querying the `coverage_requests` collection.
- `@nestjs/common` - for DI and controller structure.

Main Service Method

- **buildDashboard(uid: string)**: builds and returns a `DashboardDto` object containing:
  - Total number of user files
  - Breakdown of files by category
  - Coverage requests (pending and completed)
  - Recent request history (max 20)

Resilience Handling

If the required Firestore index is missing, a partial dashboard is returned with empty coverage history. This ensures continued availability even when Firestore is misconfigured.

Exposed Route

GET /dashboard

Method	GET
Route	/dashboard
Auth required	Yes (via Firebase ID token)
Status codes	200 OK, 500 Internal Server Error

Headers:

```
Authorization: Bearer <ID_TOKEN>
```

Success Response:

```
Status: 200 OK
{
  "totalFiles": 3,
  "filesByCategory": {
    "health": 1,
    "employment": 2
  },
  "pendingCoverageRequests": 1,
  "completedCoverageRequests": 4,
  "coverageHistory": [
    {
      "requestId": "abc123",
      "prompt": "What is my health coverage?",
      "status": "done",
      "response": "...",
      "updatedAt": 1716170000000
    }
  ]
}
```

Error Responses:

- **500 Internal Server Error** - Firestore failure or exception.

2.9 PubSub Module

Overview

The **PubSubModule** encapsulates the Google Cloud Pub/Sub integration used by the application to handle asynchronous communication between microservices. It publishes and subscribes to various event topics related to files and AI coverage analysis.

Responsibilities

- Publish events to specific Pub/Sub topics.
- Subscribe to the `coverage-response-sub` subscription.
- Update Firestore with AI coverage results upon message reception.

Topics Used

- **coverage-query** - Trigger coverage analysis requests.
- **coverage-response** - Receives responses from the AI service.
- **file-upload** - Event triggered when a file is uploaded.
- **file-delete** - Event triggered when a file is deleted.

Key Methods

- `publishMessage(topicName: string, data: object)`
  Publishes a JSON-encoded message to the specified Pub/Sub topic.
- `subscribeToCoverageResponse()`
  Subscribes to `coverage-response-sub` and processes AI results to update the Firestore `coverage_requests` collection.

Error Handling

If a received message is missing expected fields (`request_id`, `user_uuid`, or `response`), the message is rejected with `nack()`. Other exceptions are logged and also result in a negative acknowledgment.

Lifecycle Hook

The service implements `OnModuleInit` and only subscribes to incoming Pub/Sub messages if the environment is not `"test"`. This allows for isolated unit testing without background consumers.

Testing

Unit tests verify:

- That messages are published correctly with expected parameters.
- That the `subscribeToCoverageResponse` method is invoked on module initialization (except in test mode).
- That Firestore is updated when a message is handled successfully.

Mocks are used for:

- Pub/Sub client, topics and subscriptions.
- Firestore client and document operations.

2.10 VertexAI Module

Overview

The **VertexAI Module** integrates the Google Cloud Vertex AI API into the backend using the `@google-cloud/vertexai` client. It handles AI-driven reasoning and recommendation tasks by interacting with Gemini 1.5 Pro.

Responsibilities

- Load a generative model (Gemini 1.5 Pro) via the Vertex AI SDK.
- Process user prompts and file references (PDF, images).
- Generate structured reasoning and actionable recommendations.
- Log AI interactions to Firestore for traceability.

Workflow Summary

The method `generateTextContent` executes two chained LLM calls:

1. **Structured reasoning generation** based on:
   - User metadata (`firstname`, `lastname`, etc.)
   - Attached files from Firebase Storage
   - Custom domain-specific reasoning prompt
2. **Final decision generation** based on the AI's reasoning output.

Results are logged in Firestore under the `logs/{uid}/ai_interactions` collection with full metadata.

Prompt Logic

- Prompts are composed dynamically using the user's identity and structured business rules.
- **Reasoning Prompt:** guides the model to extract risk analysis and contract redundancies.
- **Final Decision Prompt:** asks the model to produce an actionable summary.
- All file attachments are referenced using GCS URIs (e.g., `gs://bucket/path`).

Key Method: `generateTextContent`

- **Inputs:** user ID, prompt string, list of Firebase Storage URLs, user metadata.
- **Output:** response object from Vertex AI's LLM (Gemini 1.5 Pro).
- **Side-effects:**
  - AI interaction logs written to Firestore.
  - Token usage, model version, and response IDs tracked.

Firestore Logging Structure

Each interaction creates or updates a Firestore document with:

- `prompt`, `reasoning`, `finalDecision`
- `vertexResponse` and `finalVertexResponse` metadata
- `createdAt`, `deletedAt`

Error Handling

- If reasoning or decision generation fails, the error is logged and rethrown.
- GCS URI generation fallback uses MIME type detection to support different file types.

Testing

Unit tests validate:

- Proper generation of both reasoning and final decision via mocked LLM.
- Firestore logs are created and updated accordingly.
- Token usage, model version, and response IDs are extracted correctly.

Mocks are used for:

- VertexAI client and model generation responses.
- Firestore document creation and update.

2.11 Auth Module

Overview

The **Auth** module handles user creation and authentication token validation using the Firebase Admin SDK. It provides a simple REST interface to register new users and verify existing sessions.

Responsibilities

- Creating new users via Firebase.
- Verifying ID tokens for secure API access.
- Fetching user information by UID.

Dependencies

- `firebase-admin` - for user management and token verification.
- `@nestjs/common` - core NestJS features.
- `class-validator` - request payload validation (via DTO).

Main Service Methods

- **create(dto: CreateUserDto)**: registers a user in Firebase.
- **checkToken(token: string)**: verifies a Firebase ID token.
- **getUserByUid(uid: string)**: fetches user data from Firebase.

Testing

Unit tests are implemented with Jest and cover:

- User creation with full name.
- Token verification.
- User retrieval by UID.

Exposed Route

POST /auth/signin

Method	POST
Route	/auth/signin
Auth required	No
Status codes	201 Created, 400 Bad Request, 500 Internal Server Error

Request Body:

```
{
  "email": "john@example.com",
  "password": "securePass123",
  "firstname": "John",
  "lastname": "Doe"
}
```

Success Response:

```
Status: 201 Created
{
  "statusCode": 201,
  "message": "User created",
  "data": {
    "uid": "user123",
    ...
  }
}
```

Error Responses:

- **400 Bad Request** - missing or invalid fields.
- **500 Internal Server Error** - Firebase failure or exception.

AI Interaction Sequence

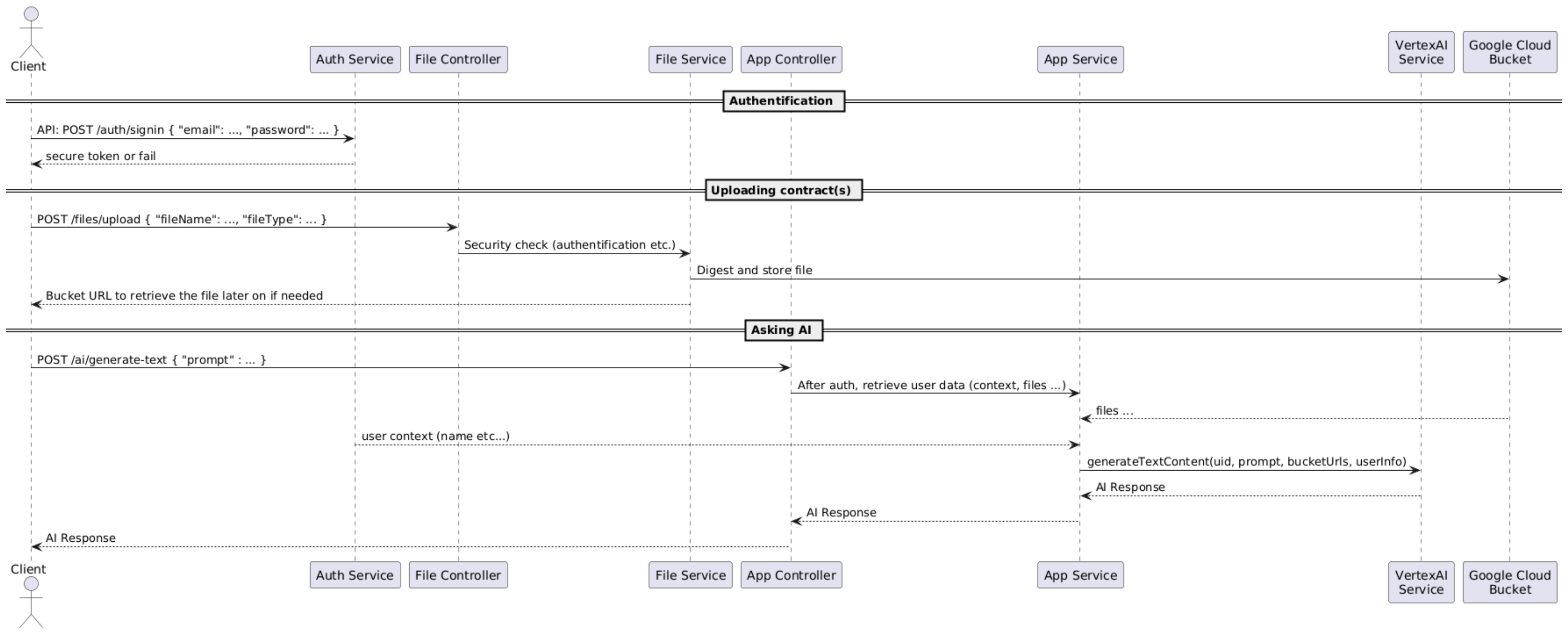

Client
Auth Service
File Controller
File Service
App Controller
App Service
VertexAI Service
Google Cloud Bucket
Authentification
API: POST /auth/signin { "email": ..., "password": ... }
secure token or fail
Uploading contract(s)
POST /files/upload { "fileName": ..., "fileType": ... }
Security check (authentification etc.)
Digest and store file
Bucket URL to retrieve the file later on if needed
Asking AI
POST /ai/generate-text { "prompt" : ... }
After auth, retrieve user data (context, files ...)
files ...
user context (name etc...)
generateTextContent(uid, prompt, bucketUrls, userInfo)
AI Response
AI Response
AI Response

Figure 2.1: Sequence diagram illustrating the AI processing flow

Chapter 3

Microservice "Document Analysis"

3.1 Overview

Document Analysis is a Python microservice that handles document ingestion, semantic entity extraction, vector storage, and natural language response generation. It is a core component of the Contract Central platform, enabling users to query their insurance documents using natural language.

3.2 Main Features

- Asynchronous message processing via Google Cloud Pub/Sub
- Entity extraction from PDF documents using Gemini 1.5 Pro (VertexAI)
- Preparation and flattening of extracted JSON entities
- Vector storage and semantic search using ChromaDB
- Post-processing with Gemini to generate user-friendly answers
- Flask-based API for manual testing and debugging

3.3 Dependencies

3.3.1 Libraries Used

- `google-cloud-aiplatform` – VertexAI SDK for Gemini models
- `google-generativeai` – Google embedding function for ChromaDB
- `google-cloud-pubsub` – Pub/Sub messaging system
- `flask` – Lightweight API framework for local testing
- `chromadb` – Vector database client
- `pytest` – Unit testing framework
- `pylint` – Linter for enforcing code quality

3.3.2 Custom Modules

- `extractor.py` – LLM-based entity extraction
- `prepare_entities.py` – Transforms JSON into flat entities
- `operations.py` – Handles CRUD on the vector database
- `post_processing.py` – Natural language response generation
- `entity.py` – Entity class definition
- `topics.py` – Message routing logic
- `pubsub_config.py` – Pub/Sub client configuration
- `app.py` – Main Flask app and listener threads

3.4 Architecture and Workflow

3.4.1 File Upload

1. Triggered via the `file-upload` Pub/Sub topic.
2. The file category is extracted from its path.
3. Gemini extracts structured entities (based on domain-specific prompts).
4. Entities are flattened and indexed in ChromaDB.

3.4.2 Coverage Query

1. Triggered via `coverage-query` topic.
2. A semantic search retrieves relevant entities from ChromaDB.
3. Gemini reformulates a user-facing answer.
4. The result is published on `coverage-response`.

3.4.3 File Deletion

1. Triggered via `file-delete`.
2. All related vectors are deleted from the database.

3.5 Endpoints (For Debugging Purpose ONLY)

- `/create` – Initialises ChromaDB collection
- `/store` – Stores hardcoded test file in vector DB
- `/search` – Performs semantic query using Flask route
- `/delete` – Deletes entities for given user and file
- `/printdb`, `/cleandb` – Debug endpoints to inspect or clean database

3.6 Testing

3.6.1 Unit Tests

Tests are located in the `src/tests/` folder. They cover:

- **extractor**: tests response parsing and LLM call behavior using mocks
- **prepare_entities**: ensures correct transformation from JSON to Entity
- **entity**: ensures the integrity and representation of the `Entity` class

3.6.2 Example:

```
pytest src/tests/
```

3.7 CI/CD Pipeline

The Document Analysis microservice integrates a continuous integration and deployment (CI/CD) pipeline using GitHub Actions.

3.7.1 Automated Tests and Linting

On every push to the `main` branch, the pipeline performs the following checks:

- Sets up a clean Python 3.10 environment
- Installs all dependencies listed in `requirements.txt`
- Runs all unit tests using `pytest`
- Performs static code analysis using `pylint`, enforcing a minimum score of 8

This ensures that the codebase remains stable, maintainable, and compliant with quality standards.

3.7.2 Cloud Run Deployment

Following a successful test and linting stage, the pipeline automatically deploys the latest version of the service to Google Cloud Run.

The deployment script (`deploy.sh`) builds a Docker image, pushes it to Google Container Registry (`gcr.io`), and triggers a Cloud Run update with the following specifications:

- Service account with restricted permissions for deployment
- Custom port configuration (`5000`)
- VPC connector for secure access to internal services
- Deployment region set to `europe-west1`

This setup enables fully automated and secure delivery from source code to production in less than a minute, with zero manual intervention.

3.8 Deployment

The microservice is deployed using Docker. A script `deploy.sh` is available to manually deploy the microservice.

3.9 Security

Only messages from authorised users are processed. The system verifies user UIDs before extracting or storing entities. No direct access is allowed to the internal vector database.

In addition, the Document Analysis microservice does not have write access to the Cloud Storage bucket. It can only read PDF documents from pre-authorised URIs for analysis. This ensures that uploaded files remain immutable and protected from unintended modification.

The service also does not store or process any personal user information beyond what is strictly necessary for document analysis. Only the document content, its category, and associated user UUID are referenced during extraction and search. No names, emails, or authentication tokens are accessible to this microservice, thereby minimizing exposure and respecting data minimization principles.

3.10 Reactive Use Case Sequence Diagram

This diagram illustrates the asynchronous flow involved when a user asks a natural language question regarding their insurance coverage. The sequence shows how the backend, Pub/Sub, Document Analysis microservice, ChromaDB, and Gemini interact to generate a response.

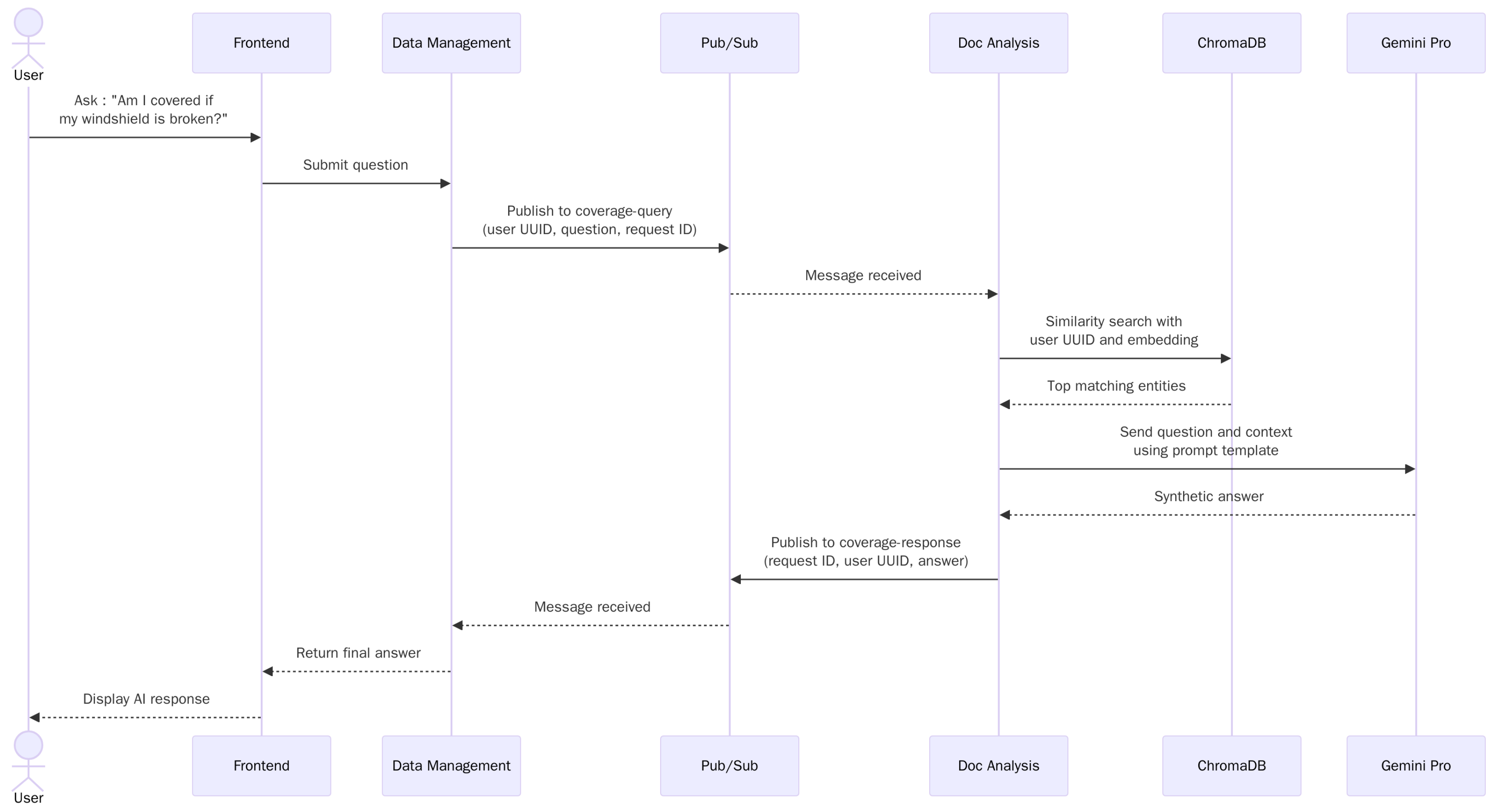

User
Frontend
Data Management
Pub/Sub
Doc Analysis
ChromaDB
Gemini Pro
Ask : "Am I covered if my windshield is broken?"
Submit question
Publish to coverage-query (user UUID, question, request ID)
Message received
Similarity search with user UUID and embedding
Top matching entities
Send question and context using prompt template
Synthetic answer
Publish to coverage-response (request ID, user UUID, answer)
Message received
Return final answer
Display AI response

Figure 3.1: Sequence diagram of the reactive use case: question to AI-generated answer

Chapter 4

Frontend Architecture and User Manual

4.1 Technology Choices

4.1.1 Flutter

We chose **Flutter** as the frontend framework due to its:

- Cross-platform capabilities
- High-performance rendering engine (Skia)
- Rich widget ecosystem
- Strong Material 3 and community support

4.1.2 Dart

Dart provides AOT compilation and a simple async model, ideal for UI-driven apps.

4.1.3 Provider (State Management)

Used for lightweight, app-wide state handling (e.g., tokens, theme, user info).

4.1.4 flutter_markdown

Renders AI-generated content with headers, bold, lists, and emphasis.

4.1.5 Syncfusion Charts

Professional, animated charts used for contract visualizations.

4.1.6 Firebase Auth

Handles user authentication, offloading token validation to the backend.

4.2 Workflow Overview

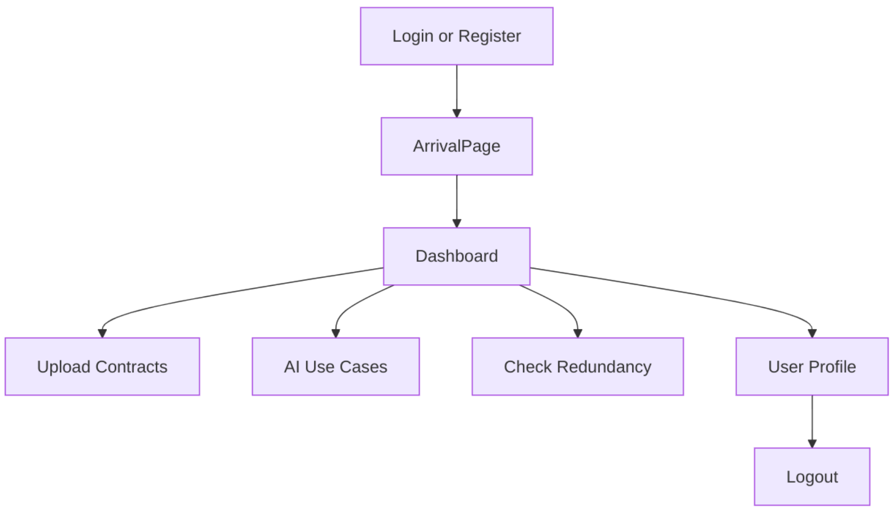


Figure 4.1: Frontend workflow from login to dashboard and feature pages

4.3 Project Structure

```
lib/
 main.dart
 http_client.dart
 pages/
    auth_page.dart
    register_page.dart
    arrival_page.dart
    home.dart
    upload_contract_page.dart
    usecaseun_page.dart
    usecasedeux_page.dart
    user_profile.dart
    side_menu.dart
    navigation_page.dart
 providers/
    app_state.dart
 widgets/
    custom_app_bar.dart
    navbar.dart
```

4.4 Component Highlights

main.dart: Initializes the app, theme, routing and Provider.
http_client.dart: Authenticated API wrapper.
app_state.dart: Global state (token, theme, user info).
home.dart: Dashboard with Syncfusion pie charts and Markdown summaries.

4.5 User Manual

Step 1: Login / Register

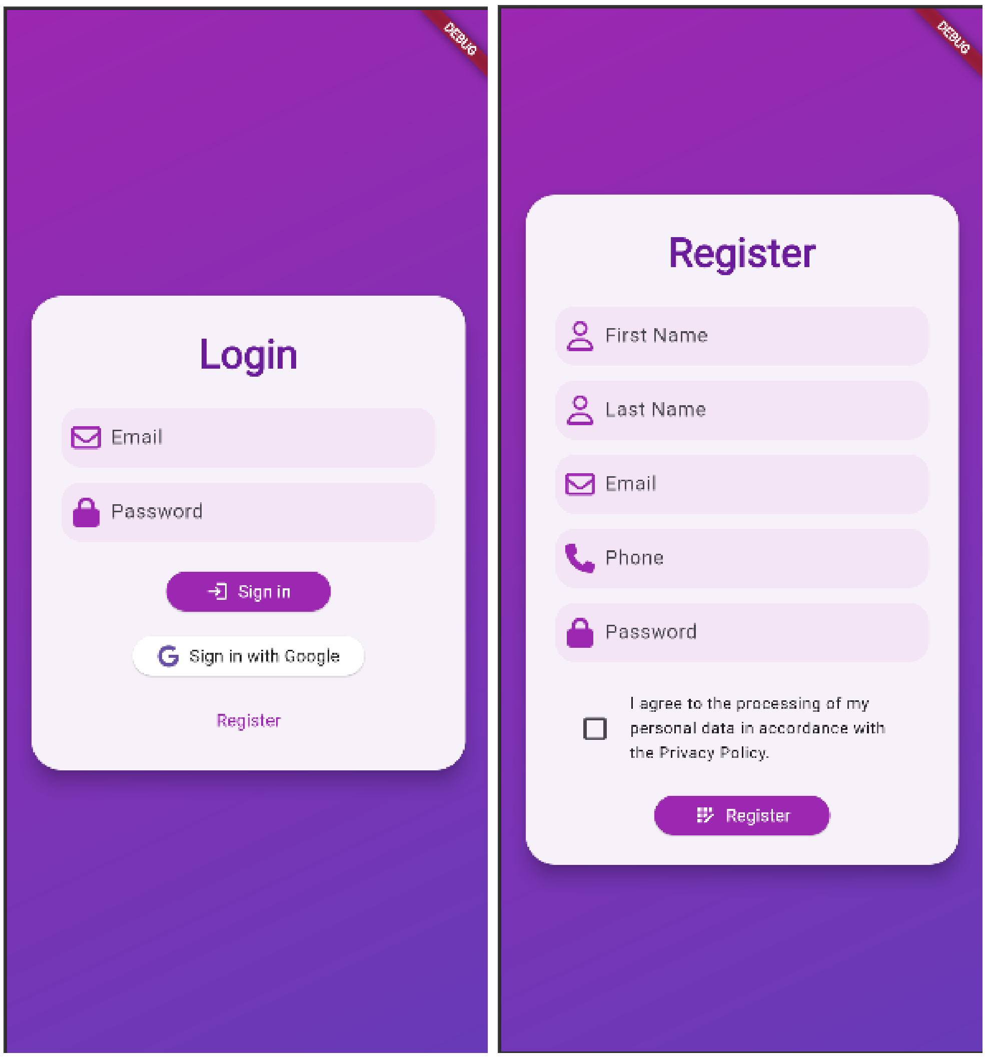
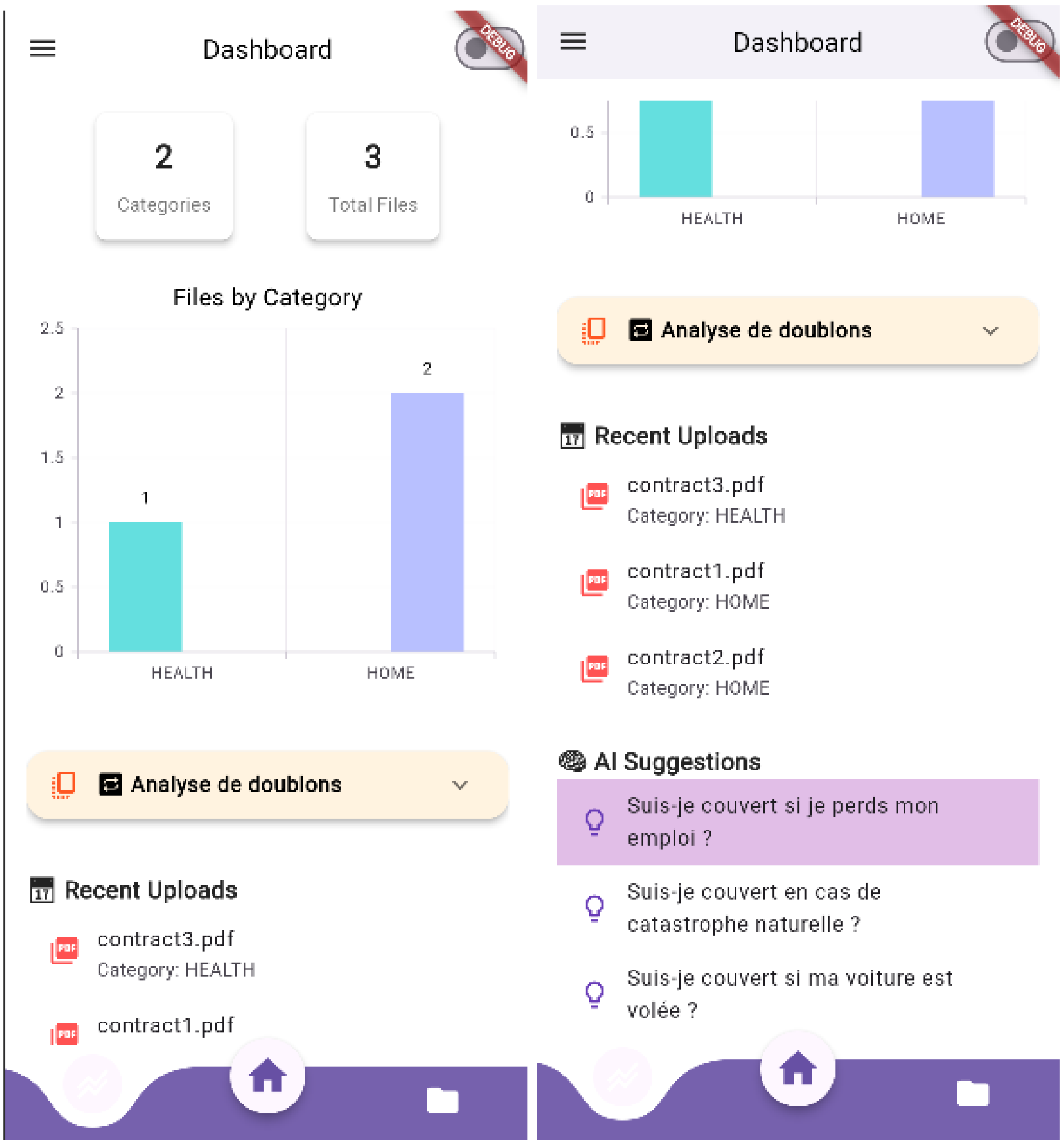


Figure 4.2: Login and registration pages

Step 2: Explore Dashboard



Dashboard

2 Categories

3 Total Files

Analyse de doublons

Recent Uploads

contract3.pdf
Category: HEALTH

contract1.pdf

Dashboard

Analyse de doublons

Recent Uploads

contract3.pdf
Category: HEALTH

contract1.pdf
Category: HOME

contract2.pdf
Category: HOME

AI Suggestions

Suis-je couvert si je perds mon emploi ?

Suis-je couvert en cas de catastrophe naturelle ?

Suis-je couvert si ma voiture est volée ?

Figure 4.3: Dashboard with charts and AI insights

Step 3: Upload Documents

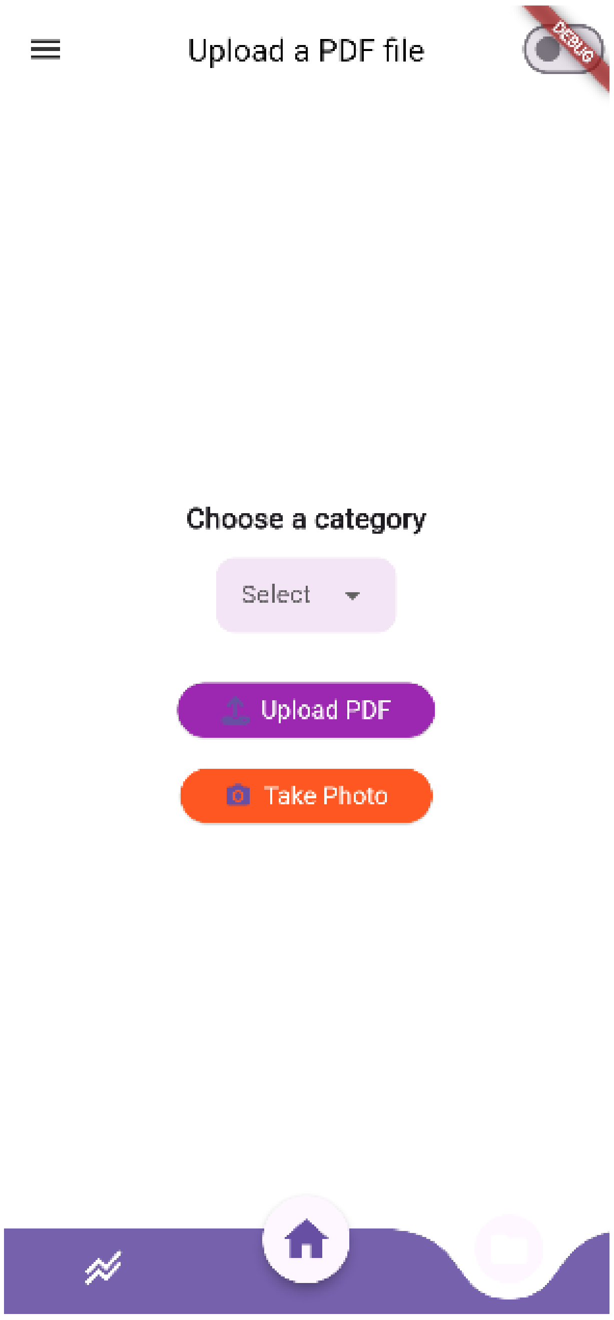
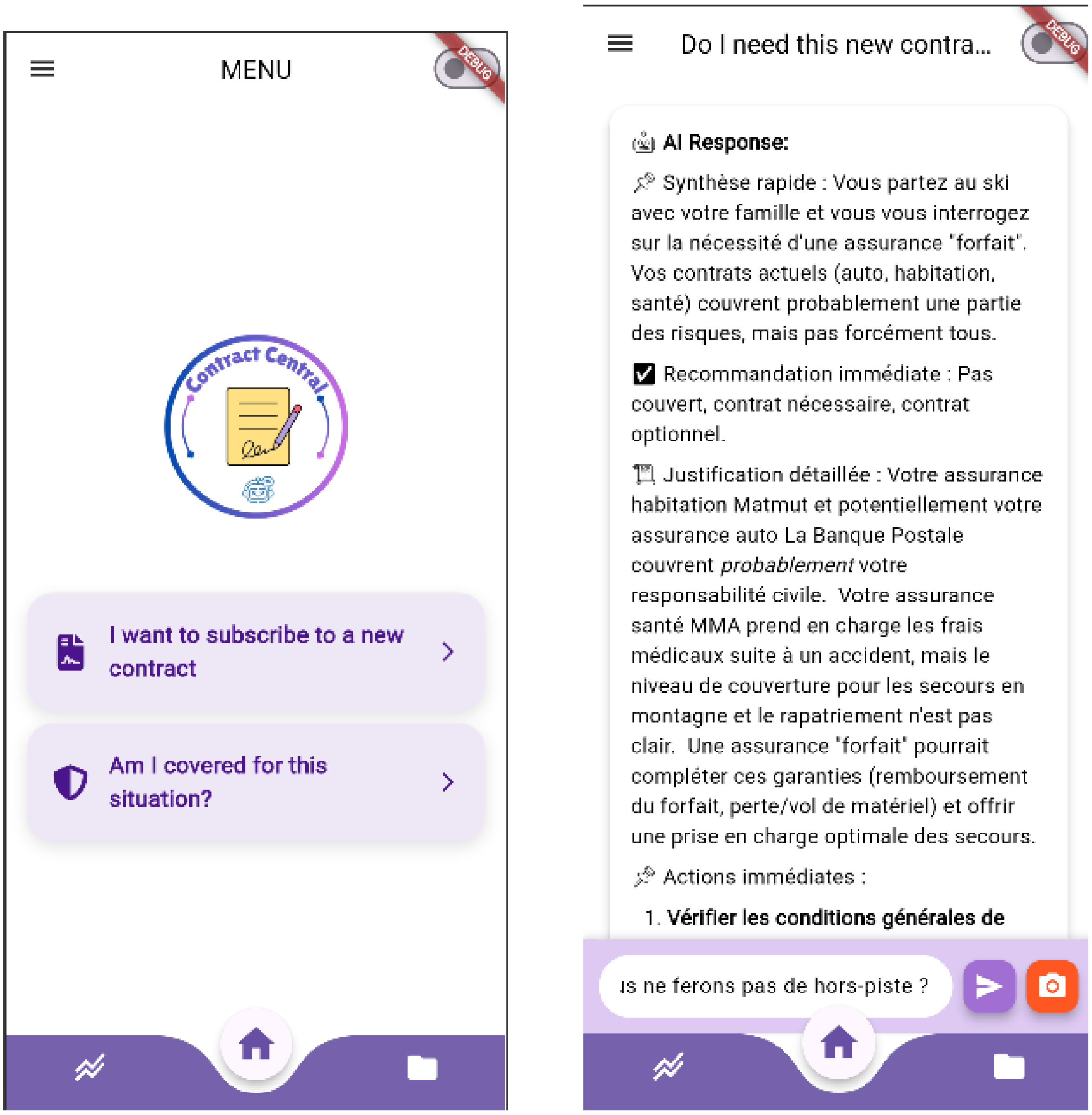


Figure 4.4: Upload screen with file picker and category selector

Step 4: Use the AI Assistant


MENU
DEBUG
Contract Central
I want to subscribe to a new contract
Am I covered for this situation?
Do I need this new contra...
DEBUG

🤖 **AI Response:**

📌 Synthèse rapide : Vous partez au ski avec votre famille et vous vous interrogez sur la nécessité d'une assurance 'forfait'. Vos contrats actuels (auto, habitation, santé) couvrent probablement une partie des risques, mais pas forcément tous.

☑ Recommandation immédiate : Pas couvert, contrat nécessaire, contrat optionnel.

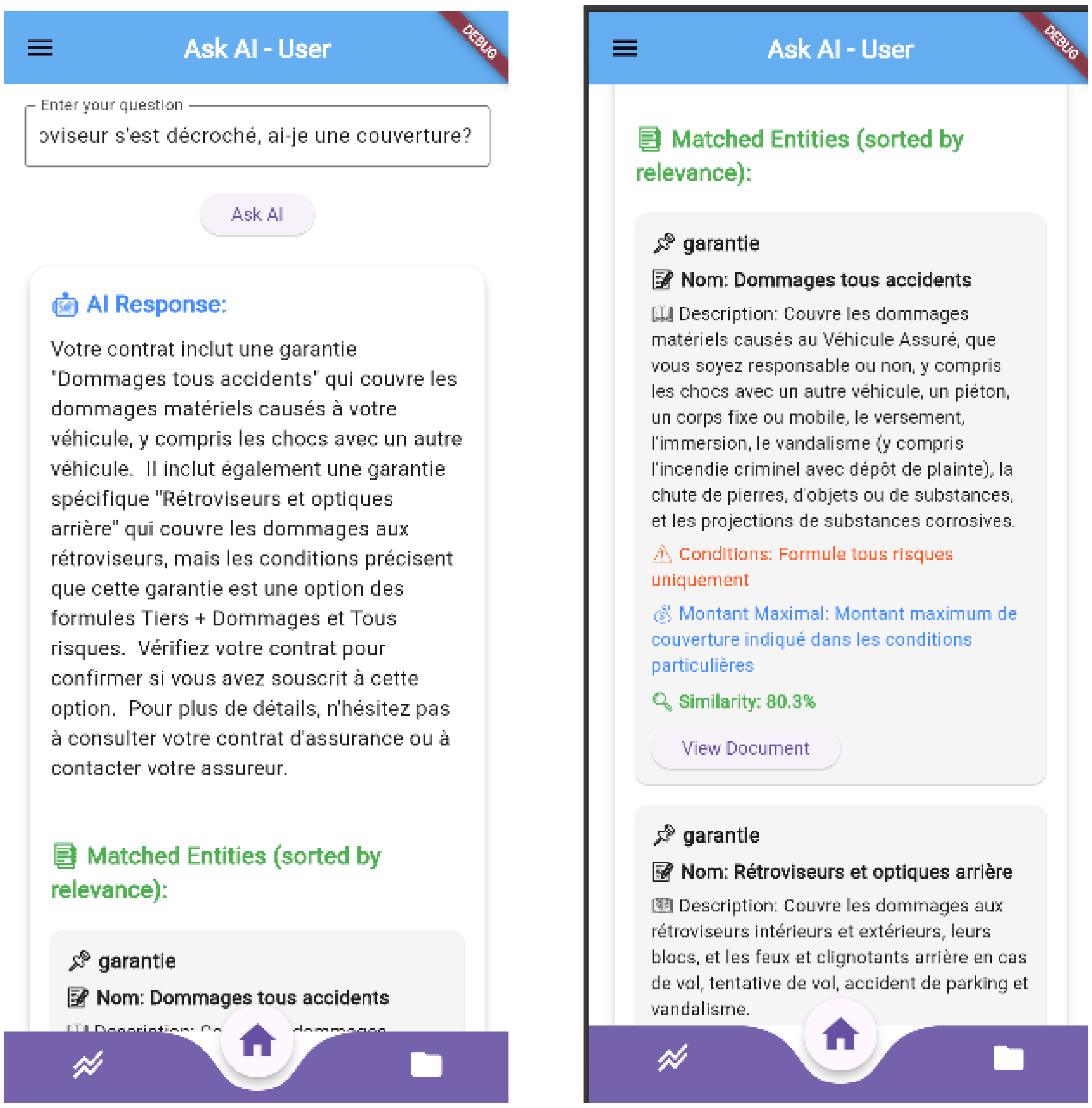
📜 Justification détaillée : Votre assurance habitation Matmut et potentiellement votre assurance auto La Banque Postale couvrent *probablement* votre responsabilité civile. Votre assurance santé MMA prend en charge les frais médicaux suite à un accident, mais le niveau de couverture pour les secours en montagne et le rapatriement n'est pas clair. Une assurance 'forfait' pourrait compléter ces garanties (remboursement du forfait, perte/vol de matériel) et offrir une prise en charge optimale des secours.

📌 Actions immédiates :

1. **Vérifier les conditions générales de**

ıs ne ferons pas de hors-piste ?

Figure 4.5: Use case menu and AI question screen



AI Response:

Votre contrat inclut une garantie 'Dommages tous accidents' qui couvre les dommages matériels causés à votre véhicule, y compris les chocs avec un autre véhicule. Il inclut également une garantie spécifique "Rétroviseurs et optiques arrière" qui couvre les dommages aux rétroviseurs, mais les conditions précisent que cette garantie est une option des formules Tiers + Dommages et Tous risques. Vérifiez votre contrat pour confirmer si vous avez souscrit à cette option. Pour plus de détails, n'hésitez pas à consulter votre contrat d'assurance ou à contacter votre assureur.

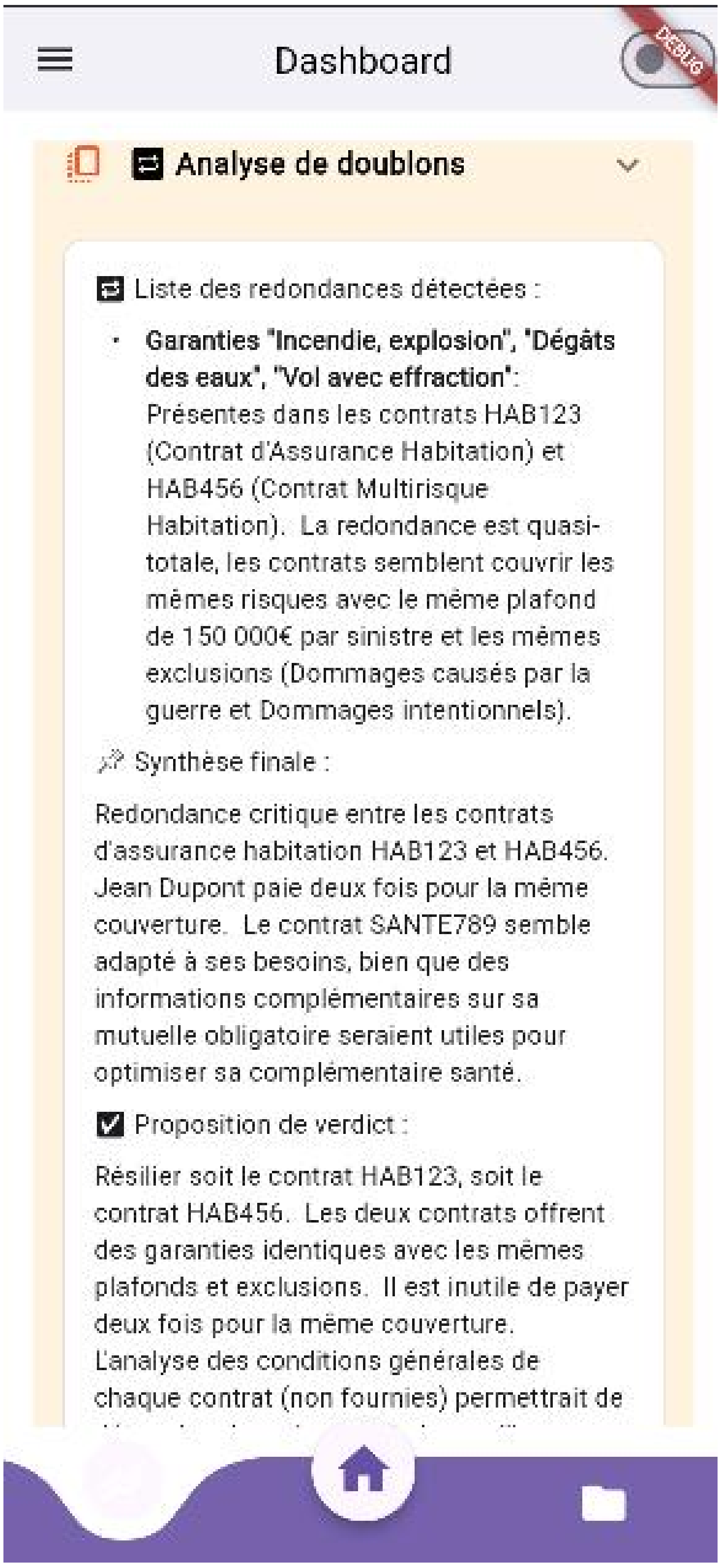
Matched Entities (sorted by relevance):

garantie
Nom: Dommages tous accidents
Description: Couvre les dommages matériels causés au Véhicule Assuré, que vous soyez responsable ou non, y compris les chocs avec un autre véhicule, un piéton, un corps fixe ou mobile, le versement, l'immersion, le vandalisme (y compris l'incendie criminel avec dépôt de plainte), la chute de pierres, d'objets ou de substances, et les projections de substances corrosives.
Conditions: Formule tous risques uniquement
Montant Maximal: Montant maximum de couverture indiqué dans les conditions particulières
Similarity: 80.3%
View Document

garantie
Nom: Rétroviseurs et optiques arrière
Description: Couvre les dommages aux rétroviseurs intérieurs et extérieurs, leurs blocs, et les feux et clignotants arrière en cas de vol, tentative de vol, accident de parking et vandalisme.

Figure 4.6: AI response and matched clauses display

Step 5: Detect Redundancy



Analyse de doublons

Liste des redondances détectées :

- **Garanties "Incendie, explosion", "Dégâts des eaux", "Vol avec effraction":** Présentes dans les contrats HAB123 (Contrat d'Assurance Habitation) et HAB456 (Contrat Multirisque Habitation). La redondance est quasi-totale, les contrats semblent couvrir les mêmes risques avec le même plafond de 150 000€ par sinistre et les mêmes exclusions (Dommages causés par la guerre et Dommages intentionnels).

Synthèse finale :

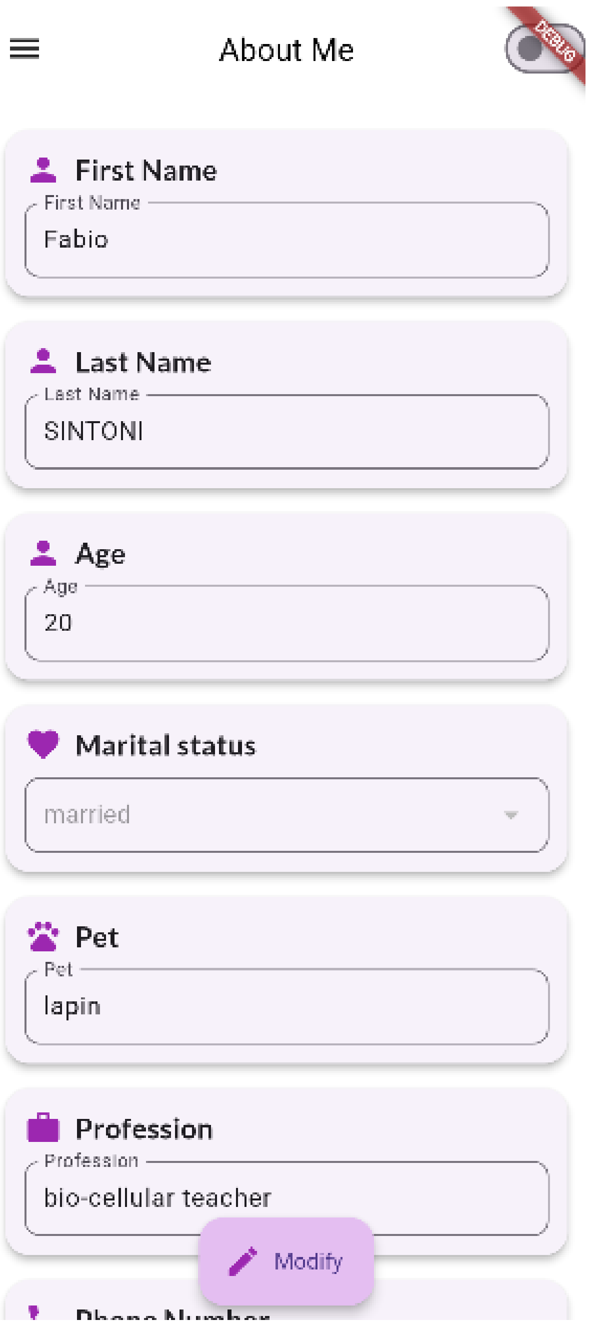
Redondance critique entre les contrats d'assurance habitation HAB123 et HAB456. Jean Dupont paie deux fois pour la même couverture. Le contrat SANTE789 semble adapté à ses besoins, bien que des informations complémentaires sur sa mutuelle obligatoire seraient utiles pour optimiser sa complémentaire santé.

☑ Proposition de verdict :

Résilier soit le contrat HAB123, soit le contrat HAB456. Les deux contrats offrent des garanties identiques avec les mêmes plafonds et exclusions. Il est inutile de payer deux fois pour la même couverture. L'analyse des conditions générales de chaque contrat (non fournies) permettrait de

Figure 4.7: Redundancy detection view

Step 6: Profile and Logout



First Name
First Name
Fabio

Last Name
Last Name
SINTONI

Age
Age
20

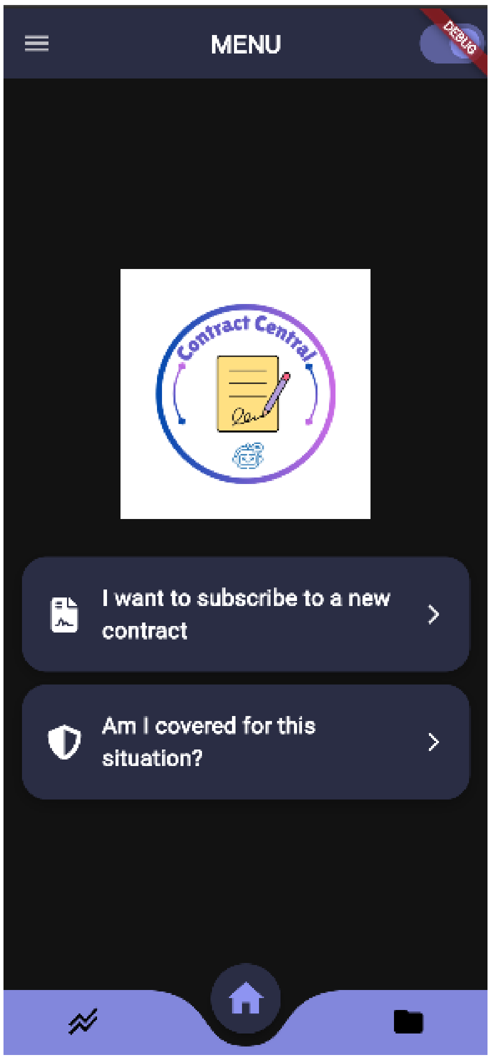
Marital status
married

Pet
Pet
lapin

Profession
Profession
bio-cellular teacher

Figure 4.8: User profile page with logout functionality

Step 7: Dark Mode



I want to subscribe to a new contract

Am I covered for this situation?

Figure 4.9: Dark/light theme toggle

Step 8: Navigation

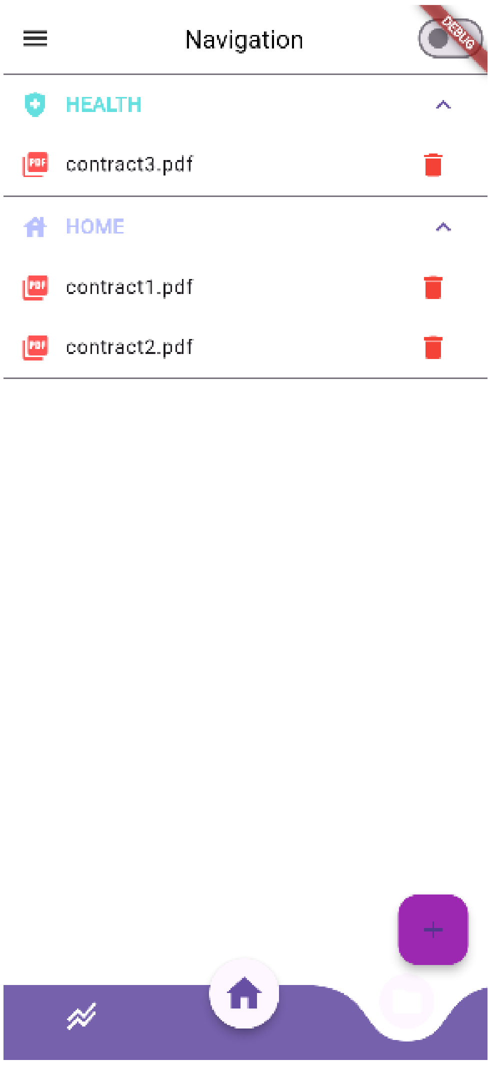


Figure 4.10: Side navigation drawer and menu button

Summary

Contract Central's frontend is designed with performance and clarity in mind. Thanks to Flutter, it offers smooth navigation, AI integration, and clear contract visualization across platforms.

Conclusion

This documentation presented the architecture, technical design, and implementation details of the Contract Central project. The system combines a mobile/web frontend built with Flutter, a scalable backend hosted on Google Cloud, and AI-driven document analysis using Gemini and ChromaDB.

Each component was developed as an independent microservice to promote modularity and ease of maintenance. The CI/CD pipeline, based on GitHub Actions, enables automated testing and deployment to Cloud Run.

The frontend was designed to offer a smooth, cross-platform user experience, while the backend ensures security and scalability through asynchronous messaging and vector-based search.

This project highlights the power of combining cloud infrastructure, AI, and cross-platform development to deliver a real-world solution for contract management.