Technical Documentation - Microservices Architecture

Final Year Project

May 22, 2025

Contents

Chapter 1

Introduction

This project is a web application composed of several interconnected microservices. The goal is to provide users with access to their documents, automated analysis of their content, and an intuitive interface.

1.1 Communication

Microservices communicate through **Google Pub/Sub**, using structured messages.

1.2 Frontend

The frontend (built with Flutter) interacts with the main backend API. It provides:

- user account management,
- document upload,
- visualization of document analysis results.

Chapter 2

Microservice NestBackend

2.1 Overview

NestBackend 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

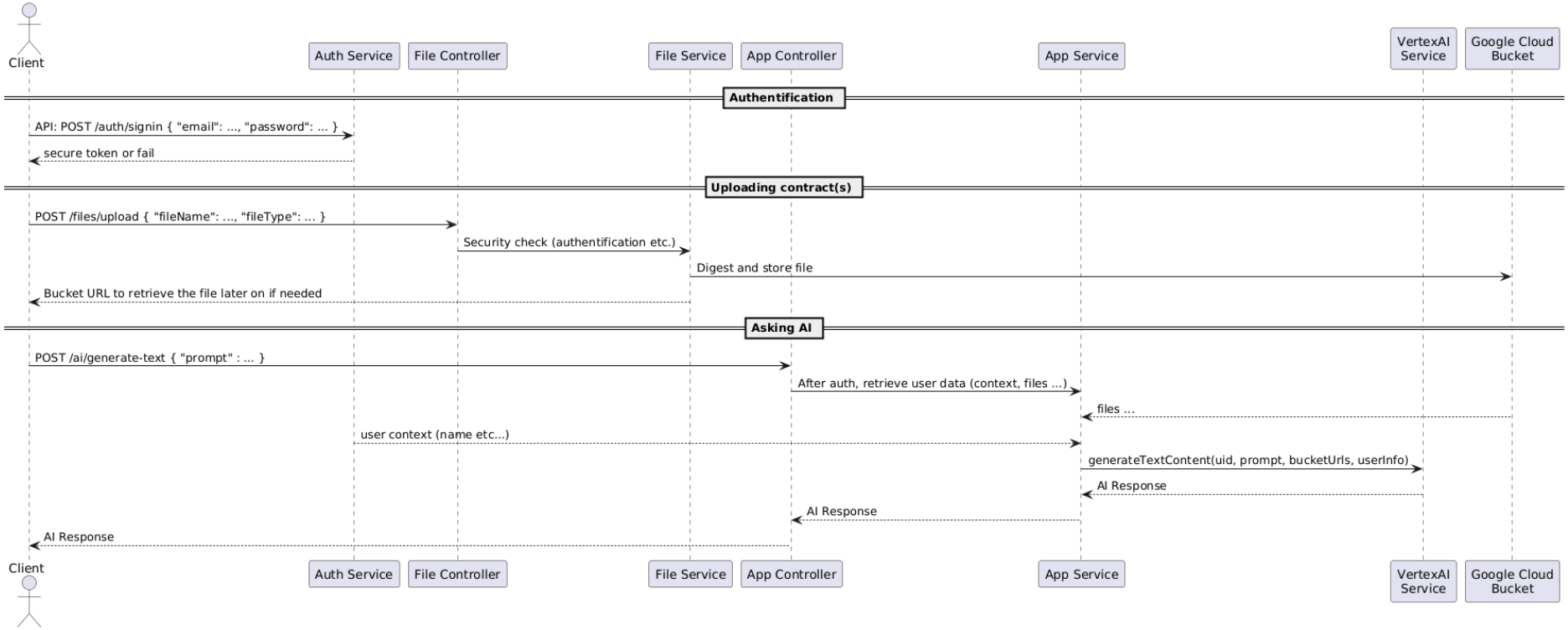

Figure 2.1: Sequence diagram illustrating the AI processing flow

Chapter 3

Microservice DocAnalysis

3.1 Purpose

This service processes documents received via Pub/Sub, vectorizes them, and enables semantic search.

3.2 Components

- **ChromaDB**: vector database
- **LangChain** (optional): NLP processing pipeline
- **PDF parsing, OCR, NLP tools**