EduAI Project Documentation

Technical, Functional, and Handover Guide

Version 1.0 - Prepared on February 14, 2026

Contents

1 Confidentiality Notice

This document is prepared for project handover, implementation review, and support transition. It contains architecture details, Firestore schema, security model, and operational guidance for the EduAI application. Distribute this document only to authorized stakeholders.

2 Table of Contents

3 1. Document Control

3.1 1.1 Document metadata

- Document title: `EduAI Project Documentation`
- Document type: `Technical + Functional + Operational`
- Project: `EduAI`
- Platform: `Flutter (Android / iOS / Web / Desktop targets in project tree)`
- Backend: `Firebase Authentication + Cloud Firestore`
- AI integration: `Gemini API via direct REST calls`
- Optional file host: `Cloudinary (for PDF storage URL workflow)`

3.2 1.2 Intended audience

- Product owner / client reviewer
- Developers taking over maintenance
- QA engineer for regression cycles
- DevOps / support engineer responsible for Firebase setup and monitoring
- Academic evaluator (if used as a college project submission artifact)

3.3 1.3 Document purpose

This document is intended to be a complete handover artifact with:

- Clear architecture explanation
- Exact behavior of each student and admin screen
- Data model and security logic
- Deployment, testing, and support guidance
- Practical extension roadmap

3.4 1.4 Versioning and maintenance

- Version 1.0 is based on the current codebase state as available in the project repository.
- Update this document when:
  - Firestore schema changes
  - Security rules change
  - New user roles/features are introduced
  - AI integration strategy changes (e.g., server-side proxy)

4 2. Executive Summary

EduAI is a Flutter-based learning application with two primary personas:

- Student: consumes study materials, attempts quizzes, tracks progress, logs wellbeing data, and uses AI chat support.
- Admin: manages classes, subjects, materials, quiz bank, student class assignment, system usage counters, and Gemini API configuration.

The application uses Firebase Authentication for identity and Cloud Firestore as the primary database. No dedicated custom backend server is used in the current architecture. The app performs direct Gemini API requests from client-side code after loading API configuration from Firestore (`app_config/gemini`).

The solution is intentionally lightweight and practical for academic demonstration while still implementing:

- Role-based access control (student/admin)
- Multi-collection domain model
- Real-time streams on key datasets
- Basic operational safety controls
- Config-driven AI key update via admin UI

Current design strengths:

- Clear modular separation by page and service
- Strong CRUD coverage for academic use case
- Firestore security rules enforce role and ownership checks
- Smooth UI with a consistent light blue modern theme
- Support for text + PDF-based study context in AI chat

Current design limitations:

- AI API key usage is client-side (suitable for demo; not ideal for production security)
- No server-side analytics aggregation
- No deep offline-first caching strategy
- Limited automated test coverage

5 3. Product Scope and Objectives

5.1 3.1 Problem statement

Students often need a single system that combines:

- Structured study resources
- Quick self-assessment through quizzes
- Basic academic analytics
- Reflective wellbeing support
- On-demand tutoring assistance

EduAI addresses this by integrating learning content, quiz evaluation, and AI tutoring in one app.

5.2 3.2 Primary objectives

1. Provide role-specific experiences for students and admins.
2. Keep data model simple and maintainable for a college-level system.
3. Enable content and quiz management by admins without code changes.
4. Offer AI tutoring mode that can be restricted to selected study materials.
5. Capture student progress and wellbeing entries for basic review.

5.3 3.3 Out-of-scope (current version)

- Parent portal
- Institution-level multi-tenancy
- Billing/subscription
- Full LMS integrations
- Proctored exams
- Secure server-side AI proxy with rate-limiting
- Advanced recommendation engine

6 4. Technology Stack and Dependencies

6.1 4.1 Runtime and framework

- Flutter SDK (Dart 3.9.x as declared in `pubspec.yaml`)
- Material 3 UI components

6.2 4.2 Core backend services

- Firebase Core
- Firebase Authentication (email/password)
- Cloud Firestore (all persistent app data)

6.3 4.3 External integrations

- Gemini API (`generateContent` endpoint)
- Cloudinary (optional, for PDF uploads from admin module)

6.4 4.4 Key package dependencies

- `firebase_core`
- `firebase_auth`
- `cloud_firestore`
- `http`
- `file_picker`
- `intl`
- `url_launcher`
- `path_provider`
- `syncfusion_flutter_pdfviewer`
- `syncfusion_flutter_pdf`

6.5 4.5 Why this stack fits the project

- Fast iteration for UI + backend integration
- Real-time stream support via Firestore snapshots
- Minimal infrastructure burden for academic delivery
- Cross-platform potential from one codebase
- Easy onboarding for maintainers familiar with Flutter/Firebase

7 5. System Architecture

7.1 5.1 High-level architecture

EduAI follows a client-centric architecture:

- Flutter app handles UI, feature workflows, and integration calls.
- Firebase Authentication handles sign-in/sign-up and auth state.
- Firestore stores users, classes, subjects, materials, quiz data, mood/reflection entries, chats, and app config.
- Gemini API receives prompts from app through HTTPS REST calls.
- Cloudinary optionally stores uploaded PDFs and returns secure URLs.

7.2 5.2 Layered structure inside app

1. Presentation layer: `lib/pages/**`
2. Domain model layer: `lib/models/**`
3. Service layer:
   - `AuthService`
   - `FirestoreService`
   - `ChatApiService`
   - `CloudinaryService`
   - `WellbeingService`
4. Configuration layer:
   - `app_config.dart`
   - `app_theme.dart`
   - Firebase options files

7.3 5.3 App startup and navigation flow

1. `main.dart` initializes Firebase.
2. `EduAiApp` boots `MaterialApp` with global theme.
3. `_RootGate` listens to Firebase auth stream.
4. If not authenticated: show `AuthPage`.
5. If authenticated:
   - ensure profile exists (`ensureProfile`)
   - stream user profile from Firestore
   - route to `AdminShellPage` or `StudentShellPage` based on role.

7.4 5.4 Role-based shell pattern

Both admin and student experiences use:

- Top app bar with dynamic title and logout
- Bottom `NavigationBar`
- `IndexedStack` body for tab persistence

This gives stable in-memory state while switching tabs and avoids route churn.

8 6. Source Code Structure

8.1 6.1 Core folders

- `lib/models` -> typed domain objects and Firestore mapping
- `lib/services` -> data access, external APIs, utility logic
- `lib/pages/auth` -> login/signup

- `lib/pages/student` -> student-facing modules
- `lib/pages/admin` -> admin-facing modules
- `lib/pages/shared` -> shared components/screens like loading
- `lib/config` -> app config and theming

8.2 6.2 Notable entry files

- `lib/main.dart`: Firebase initialization
- `lib/app.dart`: root app widget and role routing
- `lib/config/app_theme.dart`: global visual theme
- `firestore.rules`: backend access security
- `pubspec.yaml`: dependency and environment declaration

8.3 6.3 Maintainability characteristics

- Services are centralized and singleton-style (`instance` static pattern).
- Models isolate Firestore map parsing and timestamp conversion.
- UI pages remain mostly feature-oriented and readable.
- Validation messages are explicit and user-facing.

9 7. Authentication and Authorization Design

9.1 7.1 Authentication

Implemented in `AuthService`:

- `signIn(email, password)`
- `signUp(name, email, password)`
- `signOut()`
- `authStateChanges()`

Signup flow:

1. Create Firebase user (email/password).
2. Set display name.
3. Determine role:
   - if email in `AppConfig.bootstrapAdminEmails` -> admin
   - else -> student
4. Create Firestore profile in `users/{uid}`.

9.2 7.2 Profile fallback

`ensureProfile(User firebaseUser)` ensures each authenticated user has a Firestore profile. If missing, it creates a fallback student profile.

9.3 7.3 Authorization model

Authorization is enforced at Firestore rules level:

- User-specific ownership checks via `isSelf(uid)`.
- Admin role checks via `isAdmin()` using profile document role.
- Collection-specific permissions prevent unauthorized writes.

This means UI restrictions alone are not the only protection layer.

9.4 7.4 Session handling

Auth session is reactive; page routing updates automatically when auth state changes. No separate token management is manually implemented in app code.

10 8. Firestore Data Model

10.1 8.1 Collections used

- `users`
- `classes`
- `subjects`
- `app_config`
- `materials`
- `quiz_questions`
- `quiz_attempts`
- `mood_entries`
- `reflections`
- `chats`

10.2 8.2 Core relationship model

- A student (`users` with role student) may have one `classId`.
- A class (`classes`) contains multiple subject IDs (`subjectIds` array).
- A material links class + subject + chapter.
- A quiz question links class + subject + chapter.
- A quiz attempt links student + class + subject + chapter.
- Mood/reflection/chat entries each link to `studentId`.

10.3 8.3 Timestamp handling

All models parse Firestore `Timestamp` fields to Dart `DateTime`. Write operations use `Timestamp.now()` for operational fields like `createdAt` and `updatedAt`.

10.4 8.4 Data normalization approach

The schema is semi-normalized:

- IDs are stored across collections for join-by-query behavior.
- Denormalized names are not heavily duplicated.
- Aggregations are computed at read-time in UI/service logic.

11 9. Security Rules and Access Matrix

11.1 9.1 Rule utility functions

- `signedIn()`: authenticated check
- `isSelf(uid)`: user owns resource
- `isAdmin()`: role lookup from `users/{auth.uid}`

11.2 9.2 Collection rule highlights

11.2.1 users

- Read: self or admin
- Create: self only
- Update/Delete: admin only

11.2.2 classes, subjects, materials, quiz_questions, app_config

- Read: any signed-in user
- Write: admin only

11.2.3 quiz_attempts

- Read: admin or owner student
- Create: authenticated student can create only for own `studentId`
- Update/Delete: admin only

11.2.4 mood_entries and reflections

- Read: admin or owner
- Create: owner only
- Update/Delete: denied

11.2.5 chats

- Read: admin or owner
- Create: owner only
- Update: denied
- Delete: admin or owner

11.3 9.3 Security posture summary

For a client-driven app, current rules are well-structured for:

- role separation
- ownership enforcement
- admin governance over institutional datasets

Main production caveat remains AI key exposure risk in client architecture.

12 10. Application Configuration

12.1 10.1 Static app config (`app_config.dart`)

- `cloudinaryCloudName`
- `cloudinaryUnsignedPreset`
- `bootstrapAdminEmails`
- `moodCategories`

12.2 10.2 Dynamic AI config (`app_config/gemini`)

`ChatApiService` loads runtime values from Firestore:

- `apiKey` (or aliases `api_key`, `apikey`, `geminiApiKey`)
- `model` (or alias `geminiModel`)

Admin can edit this via `AdminApiSettingsPage`.

12.3 10.3 Firebase platform config

`firebase_options.dart` provides per-platform Firebase app options generated by FlutterFire CLI. These are client app identifiers, not administrator secrets.

12.4 10.4 Environment strategy recommendation

For production hardening:

1. Move Gemini access to server-side proxy.
2. Keep keys in secure secret manager.
3. Restrict client from direct external AI endpoint usage.

13 11. Student Module - Functional Documentation

13.1 11.1 Student shell

Tabs:

- Dashboard
- Materials
- Quiz
- Wellbeing
- AI Chat

The student shell uses gradient background and `IndexedStack` tab persistence.

13.2 11.2 Dashboard page

Purpose:

- Display assigned class
- Show quiz progress summary
- Show subject-wise performance

Data inputs:

- classes stream
- student attempts stream
- subjects stream

Outputs:

- total attempts
- average score
- latest score and timestamp
- per-subject average + attempt count

13.3 11.3 Materials page

Purpose:

- Show subjects assigned to student class
- Navigate to subject detail content

Behavior:

- If no class assigned: show contact-admin message.
- If class record missing: show error prompt.

- If no subjects mapped: show empty-state guidance.

13.4 11.4 Subject details page

Purpose:

- Show materials for selected subject/class
- Support PDF opening

Behavior:

- Renders title, chapter, note content, and optional PDF open button.
- Opens `StudentPdfViewerPage` for attached PDF URL.

13.5 11.5 Quiz page

Purpose:

- Launch random quiz based on class + subject + chapter
- Evaluate locally
- Persist attempts to Firestore
- Show score history

Controls:

- subject dropdown
- chapter dropdown (`All Chapters` included)
- question count dropdown (5/10/15)

Quiz workflow:

1. Student selects options.
2. App fetches random questions from service.
3. Student answers in-session.
4. App computes score and stores attempt.
5. Last score and history update through Firestore streams.

Validation:

- Ensures class exists and has subjects.
- Ensures question bank is not empty for selected context.

13.6 11.6 Wellbeing page

Sections:

- Daily mood input (from config mood categories)
- Daily reflection journal
- Mood history list
- Reflection history list

Mood save flow:

- Suggestion generated by `WellbeingService.suggestionForMood`.
- Entry written to `mood_entries`.

Reflection flow:

- Non-empty validation.
- Entry written to `reflections`.

13.7 11.7 AI Chat page (student-side)

Major capabilities:

- Chat with AI assistant
- Optional strict mode: answer only from selected study materials
- Material subset selector
- PDF text extraction and prompt context assembly
- Chat clear action

Data sources:

- `materials` (for class)
- `chats` stream
- `app_config/gemini` for API key/model

Safety and constraints:

- strict mode enforces context-based tutoring prompt
- text extraction has size limits and truncation logic
- friendly API/network error messages are surfaced to user

14 12. Admin Module - Functional Documentation

14.1 12.1 Admin shell

Tabs:

- Classes
- Materials
- Quiz
- Students
- Usage

Includes logout action and tab-wise screen persistence.

14.2 12.2 Classes and subjects management

Features:

- Create class
- Create subject
- Assign subjects to a class (dialog with checkboxes)
- Delete class
- Delete subject

Operational notes:

- Class assignment model uses subject ID arrays.
- Subject and class streams drive real-time updates.

14.3 12.3 Materials management

Features:

- Add/edit material (class, subject, chapter, title, text)
- Optional PDF URL
- Optional Cloudinary PDF upload
- View existing materials by selected class/subject
- Edit and delete material records
- Open attached PDF in viewer

Validation:

- class + subject required
- chapter/title/content required

14.4 12.4 Quiz bank management

Features:

- Add/edit quiz questions
- Select class, subject, chapter
- Enter question text + 4 options
- Set correct option index
- View existing question bank
- Delete questions

Validation:

- class + subject required
- chapter and question required
- all four options required

14.5 12.5 Students management

Features:

- View student list
- Assign class to each student
- View per-student attempt metrics:
  - total attempts
  - average score
  - last attempt time

Special handling:

- Class dropdown guards against invalid selected values if class is missing.
- Class display resolves class names via ID mapping.

14.6 12.6 Usage dashboard

Features:

- View collection-based counters:
  - students
  - quiz attempts
  - chats
  - materials
- Refresh counts on demand
- Quick entry to API settings

14.7 12.7 API settings screen

Features:

- Load current Gemini API key from Firestore
- Show/hide masked key
- Edit and replace API key
- Save key and refresh `ChatApiService` config
- Manual reload action

This enables runtime key rotation without app rebuild.

15 13. AI Chat Subsystem - Detailed Design

15.1 13.1 Configuration loading

`ChatApiService.loadConfig()` retrieves `app_config/gemini` from Firestore server source. It normalizes field name variations and stores:

- `_apiKey`
- `_model`
- debug/error context for UI display

15.2 13.2 Request construction

`ask(prompt, ...)` builds a Gemini `generateContent` request with:

- optional chat history as user/model turns
- strict or normal system instruction
- optional injected study material context
- generationConfig (temperature and token limits vary by mode)

15.3 13.3 Strict mode

When strict mode is enabled in Student Chat:

- Model is instructed to answer from provided material context only.
- If answer not found, model is instructed to return exact fallback message.
- History can include model responses for better continuity.

15.4 13.4 Context assembly from materials

Pipeline:

1. Collect note text from material content.
2. Download and parse PDF text for selected materials.
3. Extract potential module headings.
4. Normalize and trim text.
5. Build bounded context buffer with per-material and total char limits.

This reduces prompt overload and keeps requests within practical payload size.

15.5 13.5 Error normalization

HTTP failure handling maps upstream errors to user-friendly messages:

- quota/resource exhausted -> daily quota guidance
- auth/permission/API key issues -> key update guidance
- 5xx -> temporary service unavailability message
- network failures -> connectivity message

15.6 13.6 Chat persistence model

After successful reply:

- user message + AI reply stored in `chats`
- chat stream drives UI rendering in reverse chronological order
- recent history can be fetched for contextual follow-up support

16 14. PDF and Study Material Pipeline

16.1 14.1 Admin-side PDF upload

Via `CloudinaryService`:

- File picker accepts PDF only.
- Upload targets Cloudinary `raw/upload` endpoint.

- Returns `secure_url`.
- URL is saved in material record.

Fallback supported:

- Admin can manually paste PDF URL even without Cloudinary setup.

16.2 14.2 Student-side PDF viewing and saving

`StudentPdfViewerPage`:

- Loads PDF from network URL.
- Offers local save action.
- Sanitizes file names.
- Stores files under app document directory (`saved_notes`).
- Handles duplicate file names with timestamp suffix.

16.3 14.3 PDF text extraction for AI

In chat module:

- PDF binary downloaded by URL.
- Text extracted using Syncfusion `PdfTextExtractor`.
- Cached per material ID.
- Empty/invalid/oversized docs return safe empty context.

16.4 14.4 Constraints

- Image-only scanned PDFs may produce little/no extractable text.
- 15 MB cap applied in extraction routine.
- Timeout and parse failures are tolerated without app crash.

17 15. UI and Theme System

17.1 15.1 Theming strategy

Global theme is centralized in `AppTheme.lightTheme()` with:

- blue/white palette
- Material 3 widgets
- card, button, input, navigation theming
- consistent radius, elevation, and spacing behavior

17.2 15.2 Visual consistency

Applied patterns:

- gradient page backgrounds in shells and key screens
- standardized card styling
- standardized form input decoration
- standardized snackbar behavior

17.3 15.3 UX behavior patterns

- asynchronous actions disable controls while running
- user feedback through snackbar messages
- empty states and loading states explicitly rendered
- list and form sections visually separated by cards/spacers

17.4 15.4 Accessibility and readability notes

Current state:

- clear labels and direct text prompts
- moderate contrast in light theme
- icon + text pairing in action buttons

Recommended enhancements:

- semantic labels for all icon-only actions
- larger adaptive text support validation
- localization strategy for multilingual use

18 16. Error Handling and Reliability

18.1 16.1 UI-level safety patterns

- `mounted` checks before `setState` after async operations

- try/catch around external calls
- snackbars for operation outcomes
- null and empty data guards before rendering feature sections

18.2 16.2 Firestore operation resilience

- Stream builders display loading states until data is available.
- Null doc handling in model `fromDoc` methods avoids runtime parsing crashes.
- Write operations generally use trimmed values and explicit required checks.

18.3 16.3 Chat reliability measures

- Config preload with retry option
- API timeout handling
- Friendly error mapping by status code category
- context-size limiting to reduce request failures

18.4 16.4 Data integrity considerations

Strong points:

- ownership constraints in rules
- admin-only write boundaries for institutional entities
- immutable mood/reflection updates blocked by rules

Current gap:

- no schema validation framework beyond runtime parsing and simple rule logic

19 17. Performance and Scalability Considerations

19.1 17.1 Current scalability posture

Appropriate for small-to-medium institutional demo datasets with:

- moderate concurrent users
- manageable collection sizes
- lightweight query patterns

19.2 17.2 Potential bottlenecks

1. Chat context size and PDF parsing overhead on client
2. Counter queries loading whole collection snapshots for usage stats
3. Nested stream builders increasing widget rebuild work
4. Random quiz sampling done client-side after fetching matching set

19.3 17.3 Practical optimization options

- Introduce backend aggregation docs for usage counters
- Move heavy AI context extraction to cloud functions
- Add query pagination where lists can grow long
- Add local caching for class/subject lookup sets
- Precompute chapter indexes for large quiz banks

19.4 17.4 Monitoring recommendations

- Track Firestore read/write quotas
- Track chat error rate and response latency
- Track upload success/failure for Cloudinary

- Add simple telemetry hooks for screen-level failures

20 18. Deployment and Environment Setup

20.1 18.1 Prerequisites

- Flutter SDK compatible with project
- Firebase project with:
  - Authentication enabled (Email/Password)
  - Firestore enabled
- Optional Cloudinary account for PDF upload

20.2 18.2 Setup checklist

1. Clone project and run `flutter pub get`.
2. Configure Firebase project options in app.
3. Deploy Firestore rules (`firestore.rules`) and indexes.
4. Ensure admin bootstrap email list is configured as needed.
5. Create `app_config/gemini` document with `apiKey` and optional model.
6. Run application (`flutter run`) and validate role flows.

20.3 18.3 Firestore config bootstrap

Minimum docs to create manually:

- admin user profile (if not bootstrap via signup)
- `app_config/gemini` with valid API key
- classes and subjects

20.4 18.4 Build and release notes

- Flutter supports multi-platform output in this project tree.
- Validate platform Firebase options before release per target.
- Keep generated plugin files in sync with dependency changes.

21 19. Testing Strategy and Validation Checklist

21.1 19.1 Current test posture

The codebase is primarily validated through manual and runtime checks. Automated coverage is limited, so production-grade rollout should add unit/widget/integration suites.

21.2 19.2 Recommended automated test layers

1. Model parsing tests (`fromDoc` null and type variants)
2. Service method behavior tests (mock Firestore/HTTP)
3. Widget tests for auth flows and critical forms
4. Integration smoke tests for role routing and CRUD lifecycle

21.3 19.3 Manual regression checklist

21.3.1 Authentication

- signup creates correct user role
- login works for existing users
- logout clears session route

21.3.2 Admin

- class CRUD
- subject CRUD
- subject assignment to class
- material add/edit/delete
- quiz question add/edit/delete
- student class assignment
- usage counters refresh
- API key update save/reload

21.3.3 Student

- dashboard metrics visible

- material navigation and subject details open
- PDF open/save flow
- quiz generation and score history persistence
- mood and reflection save/history
- chat send/reply/store/clear with both strict and non-strict modes

21.3.4 Security

- student cannot write admin-only collections
- user ownership rules hold for chats/mood/reflection/attempts
- unauthenticated access blocked

22 20. Operations Runbook and Support Guide

22.1 20.1 Daily operational checks

- Verify Firestore connectivity
- Verify auth login for both admin and student accounts
- Verify chat config availability (`app_config/gemini`)
- Check if recent chats and quiz attempts are writing correctly

22.2 20.2 Incident handling scenarios

22.2.1 Scenario A: Chat stopped working

Checklist:

1. Open admin API settings and verify API key exists.
2. Retry config load in student chat.
3. Check quota exhaustion messages.
4. Validate internet connectivity.
5. Confirm Firestore rule access to `app_config` for signed-in users.

22.2.2 Scenario B: Student cannot see materials

Checklist:

1. Confirm student has `classId`.
2. Confirm class exists.
3. Confirm class has subject IDs assigned.
4. Confirm materials exist for class+subject.

22.2.3 Scenario C: Admin cannot edit data

Checklist:

1. Confirm admin profile role is `admin`.
2. Confirm `users/{uid}` exists.
3. Validate rule deployment state.

22.3 20.3 Backup and data export suggestions

- Schedule periodic Firestore exports
- Keep release-tagged backups before major schema changes
- Track index and rules files in version control with release notes

23 21. Known Constraints and Risk Register

23.1 21.1 Key constraints

1. Client-side AI calls expose integration pattern not ideal for production.
2. Usage counters perform collection scans and may become expensive at scale.
3. No advanced auditing trail for admin destructive actions.
4. No soft-delete strategy for classes/subjects/materials/questions.
5. No granular per-school tenant partitioning.

23.2 21.2 Risks and mitigations

23.2.1 Risk: API key misuse

- Impact: high
- Current mitigation: admin-only edit path in Firestore
- Recommended mitigation: move to server proxy + secret manager

23.2.2 Risk: accidental admin deletions

- Impact: medium
- Current mitigation: confirmations for some destructive flows
- Recommended mitigation: soft deletes and restore UI

23.2.3 Risk: inconsistent data references

- Impact: medium
- Current mitigation: null checks and fallback labels
- Recommended mitigation: referential cleanup jobs and validation scripts

23.2.4 Risk: quota exhaustion (Firestore/Gemini)

- Impact: medium
- Current mitigation: friendly error messaging
- Recommended mitigation: usage dashboards + alerting + caching

24 22. Suggested Enhancements and Roadmap

24.1 22.1 Short-term enhancements

1. Add unit and widget tests for services and key screens.
2. Add admin search/filter on student list and materials/questions.
3. Add score trend chart widgets on dashboard.
4. Add retry/backoff wrappers for network-heavy operations.
5. Improve confirmation dialogs for all destructive actions.

24.2 22.2 Medium-term enhancements

1. Introduce Cloud Functions layer for AI proxy and governance.
2. Add centralized analytics collection (events + operational metrics).
3. Add role-permission extensibility (teacher, counselor, supervisor).
4. Add chapter tagging and richer material metadata.
5. Add in-app notifications for assignment updates.

24.3 22.3 Long-term enhancements

1. Multi-school tenancy with institution isolation.
2. Adaptive learning recommendations based on attempt history.
3. Parent/guardian dashboard.
4. Offline mode with sync queue.
5. Enterprise-grade audit and compliance controls.

25 23. Appendix A - Firestore Collection Reference

25.1 23.1 users

Purpose: Profile and role metadata.

Typical fields:

- `name: string`
- `email: string`
- `role: "student" | "admin"`
- `classId: string?`
- `createdAt: timestamp?`
- `updatedAt: timestamp?` (for class assignment updates)

25.2 23.2 classes

- `name: string`
- `subjectIds: string[]`
- `createdAt: timestamp?`
- `updatedAt: timestamp?`

25.3 23.3 subjects

- name: string
- createdAt: timestamp?
- updatedAt: timestamp?

25.4 23.4 materials

- classId: string
- subjectId: string
- chapter: string
- title: string
- content: string
- pdfUrl: string?
- createdBy: string
- createdAt: timestamp?
- updatedAt: timestamp?

25.5 23.5 quiz_questions

- classId: string
- subjectId: string
- chapter: string
- question: string
- options: string[]
- correctIndex: number
- createdBy: string
- createdAt: timestamp?
- updatedAt: timestamp?

25.6 23.6 quiz_attempts

- studentId: string
- classId: string
- subjectId: string
- chapter: string
- totalQuestions: number
- correctAnswers: number
- scorePercent: number
- questionIds: string[]
- attemptedAt: timestamp?

25.7 23.7 mood_entries

- studentId: string
- dateLabel: string (YYYY-MM-DD)
- mood: string
- suggestion: string
- createdAt: timestamp?

25.8 23.8 reflections

- studentId: string
- dateLabel: string (YYYY-MM-DD)
- text: string
- createdAt: timestamp?

25.9 23.9 chats

- studentId: string
- userMessage: string
- aiReply: string
- createdAt: timestamp?

25.10 23.10 app_config/gemini

- apiKey: string
- model: string?
- updatedAt: timestamp?

26 24. Appendix B - Service Method Reference

26.1 24.1 AuthService

- authStateChanges()
- signIn(email, password)
- signUp(name, email, password)
- signOut()
- profileStream(uid)
- getProfile(uid)
- ensureProfile(firebaseUser)

26.2 24.2 FirestoreService (selected)

26.2.1 Class/subject

- streamClasses()
- saveClass(...)
- updateClassSubjects(...)
- deleteClass(classId)
- streamSubjects()

- saveSubject(...)
- deleteSubject(subjectId)
- assignClassToStudent(studentId, classId)

26.2.2 App config

- getGeminiApiKey()
- saveGeminiApiKey(apiKey)

26.2.3 Materials

- streamMaterials(classId, subjectId)
- streamClassMaterials(classId)
- saveMaterial(...)
- deleteMaterial(materialId)

26.2.4 Quiz

- streamQuizQuestions(...)
- saveQuizQuestion(...)
- deleteQuizQuestion(questionId)
- getRandomQuestions(...)
- saveQuizAttempt(attempt)
- streamStudentAttempts(studentId)
- streamAllAttempts()

26.2.5 Wellbeing and reflection

- streamMoodEntries(studentId)
- saveMood(...)
- streamReflections(studentId)
- saveReflection(...)

26.2.6 Chat

- streamChats(studentId)
- getRecentChats(studentId, limit)
- saveChat(studentId, userMessage, aiReply)
- clearChats(studentId)

26.2.7 Admin usage

- streamStudents()
- getUsageCounts()

26.3 24.3 ChatApiService

- loadConfig(forceRefresh)
- ask(prompt, history, studyMaterialContext, answerOnlyFromMaterials, includeModelHistory)

Internal responsibilities:

- config field normalization
- strict/normal system instruction strategy
- request payload generation
- API response extraction
- user-friendly error normalization

26.4 24.4 CloudinaryService

- isConfigured
- pickAndUploadPdf()
- uploadPdf(fileName, filePath, bytes)

26.5 24.5 WellbeingService

- suggestionForMood(mood)

27 25. Appendix C - Handover Checklist

27.1 25.1 Configuration handover

☐ Firebase project access transferred
☐ Firestore rules deployed and verified
☐ Firestore indexes deployed
☐ Admin bootstrap emails reviewed
☐ Gemini API key set in app_config/gemini
☐ Optional Cloudinary credentials verified

27.2 25.2 Access handover

☐ At least one admin account validated
☐ Test student account validated
☐ Permission boundaries tested (admin vs student)

27.3 25.3 Functional handover

- ☐ Class/subject creation and assignment confirmed
- ☐ Material creation (with/without PDF URL) confirmed
- ☐ Quiz creation and student attempt recording confirmed
- ☐ Mood and reflection logging confirmed
- ☐ AI chat strict/non-strict modes confirmed
- ☐ Usage dashboard counters confirmed

27.4 25.4 Operational handover

- ☐ Support runbook shared
- ☐ Incident owner(s) identified
- ☐ Backup/export process documented
- ☐ Release tagging and rollback process documented

27.5 25.5 Recommended immediate post-handover actions

1. Implement automated smoke tests for critical workflows.
2. Introduce staging and production Firebase project separation if not already done.
3. Plan server-side AI proxy migration for better key protection.
4. Add event-level analytics for system and pedagogical insights.

28 Closing Note

EduAI in its current implementation provides a practical and coherent educational platform suitable for demonstration and controlled rollout. Its structure is clean enough for extension and stable handover, with clear separation between student and admin capabilities. The project is best described as a strong foundation that can evolve into a production-grade solution through targeted hardening in security, testing, observability, and backend governance.