Low-Level Design

on

Non-Profit QuizBot

Course Name: GenAI

Institution Name: Medicaps University – Datagami Skill Based Course

Submitted by:

Sr no	Student Name	Enrolment Number
1.	Ananya Subramanya Rao	EN22CS301120
2.	Anirudh Vyas	EN22CS301131
3.	Anushka Kashyap	EN22CS301180
4.	Atharv Chaturvedi	EN22CS301228
5.	Harshit Panchal	EN23CS3L1010

Group Name: Group 03D2

Project Number: GAI-15

Industry Mentor Name: Rishabh SIr

University Mentor Name: Dr. Hemlata Patel

Academic Year: 2026

1. Introduction

1.1 Purpose and Scope

This Low-Level Design (LLD) document provides a detailed technical specification for QuizBot — Non-Profit Quiz/Tutor Bot. It translates the high-level architecture into concrete implementation details including module design, API contracts, algorithm specifications, database schemas, and security patterns. This document serves as the primary technical reference for development, code review, and future maintenance of the QuizBot system.

The scope of this document covers all components of the QuizBot v1.0 local web application. It includes the React 19 frontend, FastAPI Python backend, Google Gemini 2.5 Flash LLM integration, Sentence Transformers embedding pipeline, ChromaDB vector store, Firebase Authentication and Realtime Database, and the SQLite local database.

1.2 System Overview

QuizBot is an AI-powered educational assessment platform designed for professionals and volunteers in the non-profit sector. The system accepts donor email content as input, uses a Large Language Model to generate contextual multiple-choice quiz questions, evaluates user responses with detailed AI-generated explanations, and tracks learning progress over time through a secure and responsive web application.

The system is built on a 3-tier client-server architecture with a clear separation of concerns between the Presentation Layer (React 19), Business Logic Layer (FastAPI), and Data Layer (SQLite, Firebase, ChromaDB). A dedicated AI Layer integrates with external services (Gemini 2.5 Flash) and in-process ML models (Sentence Transformers).

1.3 Design Assumptions and Constraints

- The application runs as a local web application (v1.0). Cloud deployment is planned for future versions.
- All users authenticate using email/password through Firebase Authentication.
- Gemini 2.5 Flash API access requires a valid GOOGLE_API_KEY stored in the .env file.
- ChromaDB operates as an in-process client with persistent storage. No dedicated ChromaDB server is required.
- SQLite is used for local structured storage. PostgreSQL migration is planned for future production deployment.
- The frontend runs on localhost:5173 (Vite dev server). CORS is restricted to this origin only.
- JWT tokens expire after 1 hour. The Axios interceptor automatically handles 401 responses and attempts token refresh via Firebase.
- All API keys and credentials must be stored in the .env file and must never be committed to version control.
- The system supports single-user sessions. Multi-user team accounts are planned for future versions.

2. System Architecture Overview

2.1 Layered Architecture

QuizBot implements a 4-layer architecture where each layer has a clearly defined responsibility and communicates only with adjacent layers. This separation ensures maintainability, testability, and scalability.

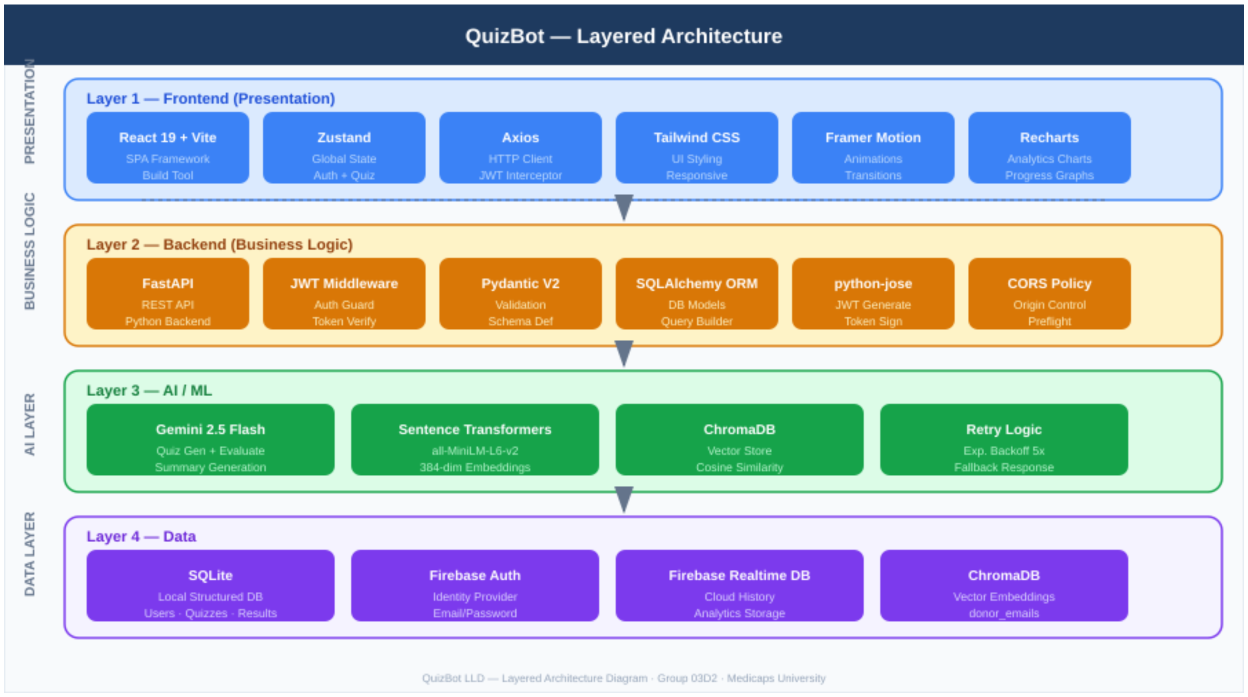


Figure 1: QuizBot Layered Architecture — Frontend, Backend, AI, and Data Layers

- Layer 1 — Frontend (Presentation): Handles all user interactions, state management, and UI rendering. Built with React 19, Vite, Tailwind CSS, Zustand, Axios, Framer Motion, and Recharts.
- Layer 2 — Backend (Business Logic): Manages REST API routing, authentication enforcement, quiz generation orchestration, evaluation logic, and database operations. Built with FastAPI, SQLAlchemy ORM, python-jose, and Pydantic V2.
- Layer 3 — AI Layer: Handles all AI and machine learning operations. Google Gemini 2.5 Flash handles quiz generation, answer evaluation, and summary generation. Sentence Transformers handle email embedding. ChromaDB provides vector storage and semantic search.
- Layer 4 — Data Layer: Manages persistent data storage. SQLite stores structured relational data locally. Firebase Realtime Database stores quiz history and analytics in the cloud. ChromaDB stores vector embeddings.

2.2 Technology Stack Mapping

Layer	Technology	Responsibility
Frontend	React 19 + Vite	SPA framework, build tooling, component rendering
	Zustand	Global auth state + quiz state management
	Axios	HTTP client with JWT interceptor for all API calls

	Tailwind CSS + Framer	Responsive UI styling + page animations
Backend	Python + FastAPI	REST API server, request routing, middleware
	SQLAlchemy ORM	Database models, parameterised queries, FK relationships
	python-jose	JWT token generation, signing (HS256), and validation
	Pydantic V2	Request/response schema validation, 422 error handling
AI Layer	Gemini 2.5 Flash	MCQ generation, per-answer evaluation, summary generation
	Sentence Transformers	384-dim email embedding (all-MiniLM-L6-v2)
	ChromaDB	Vector store, cosine similarity search, HNSW index
Data Layer	SQLite	Users, quizzes, questions, results, progress — local
	Firebase Auth	Identity provider, email/password, ID Token issuance
	Firebase Realtime DB	Cloud quiz history, analytics, hierarchical JSON

2.3 Sequence Diagram

The sequence diagram below illustrates the complete end-to-end flow of the QuizBot system across all four major stages: Authentication, Quiz Generation, Evaluation, and Progress tracking.

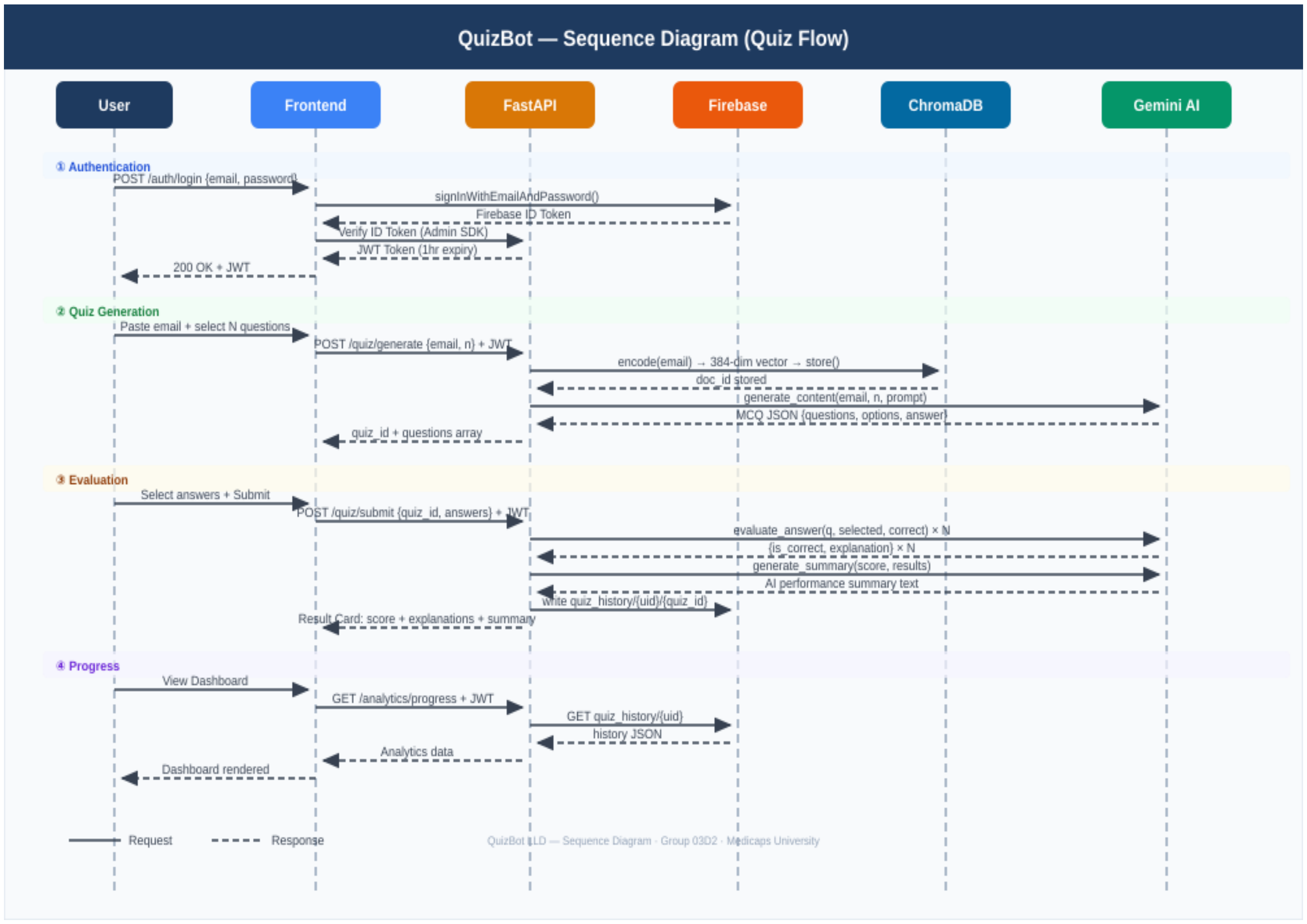

QuizBot — Sequence Diagram (Quiz Flow)
User
Frontend
FastAPI
Firebase
ChromaDB
Gemini AI
① Authentication
POST /auth/login {email, password}
signInWithEmailAndPassword()
Firebase ID Token
Verify ID Token (Admin SDK)
JWT Token (1hr expiry)
200 OK + JWT
② Quiz Generation
Paste email + select N questions
POST /quiz/generate {email, n} + JWT
encode(email) → 384-dim vector → store()
doc_id stored
generate_content(email, n, prompt)
MCQ JSON {questions, options, answer}
quiz_id + questions array
③ Evaluation
Select answers + Submit
POST /quiz/submit {quiz_id, answers} + JWT
evaluate_answer(q, selected, correct) × N
{is_correct, explanation} × N
generate_summary(score, results)
AI performance summary text
write quiz_history/{uid}/{quiz_id}
Result Card: score + explanations + summary
④ Progress
View Dashboard
GET /analytics/progress + JWT
GET quiz_history/{uid}
history JSON
Analytics data
Dashboard rendered
Request
Response
QuizBot LLD — Sequence Diagram · Group 03D2 · Medicaps University

Figure 2: QuizBot Sequence Diagram — Complete Request/Response Flow

3. Module Interaction Design

QuizBot is composed of six primary modules. Each module encapsulates a specific domain of functionality and communicates with other modules through well-defined interfaces.

3.1 Auth Module

- Components: auth_service.py (backend), AuthStore (Zustand), LoginPage / RegisterPage (React)
- Responsibilities: User registration, Firebase ID Token verification via Admin SDK, JWT issuance (HS256, 1hr expiry), JWT refresh via Axios interceptor on 401, session termination (Firebase signOut + localStorage clear)
- Interfaces: POST /api/auth/register, POST /api/auth/login, POST /api/auth/refresh
- Dependencies: Firebase Authentication, Firebase Admin SDK, python-jose, SQLite users table

3.2 Quiz Generation Module

- Components: quiz_generator.py (backend), llm_service.py, vector_db.py, EmailInputPage (React)
- Responsibilities: Accept donor email + question count, encode email to 384-dim vector, store in ChromaDB, send to Gemini 2.5 Flash with structured prompt, parse MCQ JSON response, save quiz to SQLite
- Interfaces: POST /api/quiz/generate
- Dependencies: Gemini 2.5 Flash API, Sentence Transformers, ChromaDB, SQLite quizzes + questions tables
- Retry Logic: Exponential backoff — delays of 1s, 2s, 4s, 8s, 16s before final fallback response

3.3 Evaluation Module

- Components: llm_service.py evaluate_answer(), quiz_result_service.py (backend), QuizInterface / ResultCard (React)
- Responsibilities: Loop through each submitted answer, send (question, selected, correct) to Gemini for evaluation, collect is_correct flag and explanation, generate overall AI performance summary, save results to SQLite and Firebase Realtime DB
- Interfaces: POST /api/quiz/submit
- Dependencies: Gemini 2.5 Flash API, SQLite quiz_results + question_results tables, Firebase Realtime DB

3.4 Analytics Module

- Components: analytics_service.py (backend), Dashboard / ProgressPage / HistoryPage (React)
- Responsibilities: Read quiz history from Firebase Realtime DB, calculate accuracy rate and score trends, return structured analytics data to frontend for Recharts visualization
- Interfaces: GET /api/analytics/history, GET /api/analytics/progress
- Dependencies: Firebase Realtime DB, SQLite user_progress table

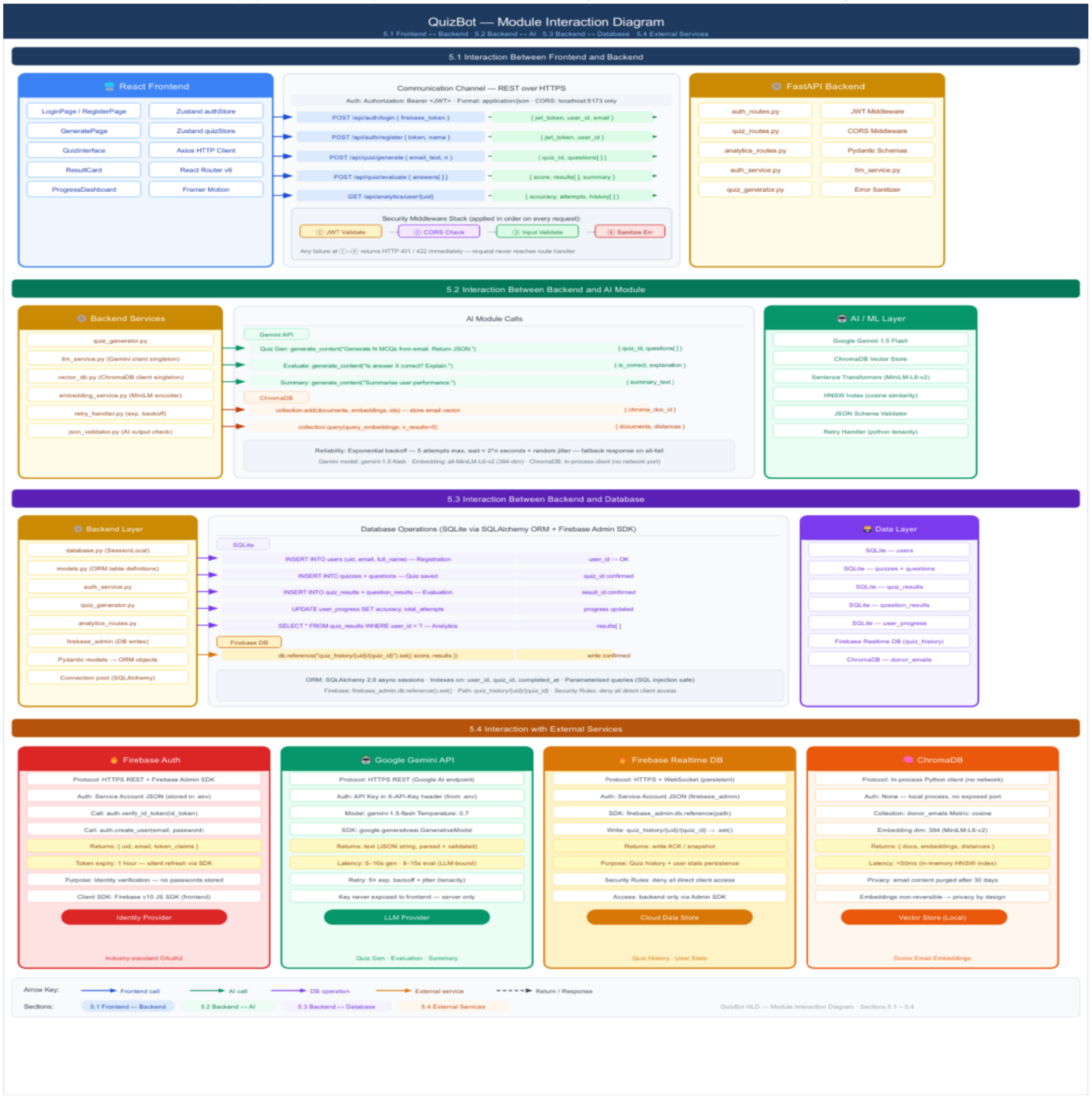
3.5 Vector Store Module

- Components: vector_db.py (backend)
- Responsibilities: Initialize ChromaDB persistent client, create donor_emails collection on first run, encode email text to 384-dim vector using Sentence Transformers, upsert into ChromaDB with

quiz_id as document ID, retrieve similar embeddings by cosine similarity for context-aware quiz generation
- Interfaces: Internal module — called by quiz_generator.py only
- Dependencies: ChromaDB, Sentence Transformers (all-MiniLM-L6-v2)

3.6 Security Module

- Components: auth_middleware.py (JWT middleware), CORS configuration in main.py
- Responsibilities: Extract Bearer token from Authorization header, verify JWT signature and expiry, inject user context into request state, enforce CORS policy, validate all request payloads using Pydantic V2 models
- Interfaces: Applied globally to all non-public FastAPI endpoints via dependency injection
- Public Routes (bypass JWT): POST /api/auth/register, POST /api/auth/login



QuizBot — Module Interaction Diagram
5.1 Frontend ↔ Backend · 5.2 Backend ↔ AI · 5.3 Backend ↔ Database · 5.4 External Services
5.1 Interaction Between Frontend and Backend
React Frontend
LoginPage / RegisterPage
Zustand authStore
GeneratePage
Zustand quizStore
QuizInterface
Axios HTTP Client
ResultCard
React Router v6
ProgressDashboard
Framer Motion
Communication Channel — REST over HTTPS
Auth: Authorization: Bearer <JWT> · Format: application/json · CORS: localhost:5173 only
POST /api/auth/login { firebase_token }
{ jwt_token, user_id, email }
POST /api/auth/register { token, name }
{ jwt_token, user_id }
POST /api/quiz/generate { email_text, n }
{ quiz_id, questions[ ] }
POST /api/quiz/evaluate { answers[ ] }
{ score, results[ ], summary }
GET /api/analytics/user/{uid}
{ accuracy, attempts, history[ ] }
Security Middleware Stack (applied in order on every request):
① JWT Validate
② CORS Check
③ Input Validate
④ Sanitize Err
Any failure at ①–④ returns HTTP 401 / 422 immediately — request never reaches route handler
FastAPI Backend
auth_routes.py
JWT Middleware
quiz_routes.py
CORS Middleware
analytics_routes.py
Pydantic Schemas
auth_service.py
llm_service.py
quiz_generator.py
Error Sanitizer
5.2 Interaction Between Backend and AI Module
Backend Services
quiz_generator.py
llm_service.py (Gemini client singleton)
vector_db.py (ChromaDB client singleton)
embedding_service.py (MiniLM encoder)
retry_handler.py (exp. backoff)
json_validator.py (AI output check)
AI Module Calls
Gemini API
Quiz Gen: generate_content("Generate N MCQs from email. Return JSON.")
{ quiz_id, questions[ ] }
Evaluate: generate_content("Is answer X correct? Explain.")
{ is_correct, explanation }
Summary: generate_content("Summarise user performance.")
{ summary_text }
ChromaDB
collection.add(documents, embeddings, ids) — store email vector
{ chroma_doc_id }
collection.query(query_embeddings, n_results=5)
{ documents, distances }
Reliability: Exponential backoff — 5 attempts max, wait = 2^n seconds + random jitter — fallback response on all-fail
Gemini model: gemini-1.5-flash · Embedding: all-MiniLM-L6-v2 (384-dim) · ChromaDB: in-process client (no network port)
AI / ML Layer
Google Gemini 1.5 Flash
ChromaDB Vector Store
Sentence Transformers (MiniLM-L6-v2)
HNSW Index (cosine similarity)
JSON Schema Validator
Retry Handler (python tenacity)
5.3 Interaction Between Backend and Database
Backend Layer
database.py (SessionLocal)
models.py (ORM table definitions)
auth_service.py
quiz_generator.py
analytics_routes.py
firebase_admin (DB writes)
Pydantic models → ORM objects
Connection pool (SQLAlchemy)
Database Operations (SQLite via SQLAlchemy ORM + Firebase Admin SDK)
SQLite
INSERT INTO users (uid, email, full_name) — Registration
user_id → OK
INSERT INTO quizzes + questions — Quiz saved
quiz_id confirmed
INSERT INTO quiz_results + question_results — Evaluation
result_id confirmed
UPDATE user_progress SET accuracy, total_attempts
progress updated
SELECT * FROM quiz_results WHERE user_id = ? — Analytics
results[ ]
Firebase DB
db.reference("quiz_history/{uid}/{quiz_id}").set({ score, results })
write confirmed
ORM: SQLAlchemy 2.0 async sessions · Indexes on: user_id, quiz_id, completed_at · Parameterised queries (SQL injection safe)
Firebase: firebase_admin.db.reference().set() · Path: quiz_history/{uid}/{quiz_id} · Security Rules: deny all direct client access
Data Layer
SQLite — users
SQLite — quizzes + questions
SQLite — quiz_results
SQLite — question_results
SQLite — user_progress
Firebase Realtime DB (quiz_history)
ChromaDB — donor_emails
5.4 Interaction with External Services
Firebase Auth
Protocol: HTTPS REST + Firebase Admin SDK
Auth: Service Account JSON (stored in .env)
Call: auth.verify_id_token(id_token)
Call: auth.create_user(email, password)
Returns: { uid, email, token_claims }
Token expiry: 1 hour — silent refresh via SDK
Purpose: Identity verification — no passwords stored
Client SDK: Firebase v10 JS SDK (frontend)
Identity Provider
Industry-standard OAuth2
Google Gemini API
Protocol: HTTPS REST (Google AI endpoint)
Auth: API Key in X-API-Key header (from .env)
Model: gemini-1.5-flash Temperature: 0.7
SDK: google.generativeai.GenerativeModel
Returns: text (JSON string, parsed + validated)
Latency: 5–10s gen · 8–15s eval (LLM-bound)
Retry: 5× exp. backoff + jitter (tenacity)
Key never exposed to frontend — server only
LLM Provider
Quiz Gen · Evaluation · Summary
Firebase Realtime DB
Protocol: HTTPS + WebSocket (persistent)
Auth: Service Account JSON (firebase_admin)
SDK: firebase_admin.db.reference(path)
Write: quiz_history/{uid}/{quiz_id} → .set()
Returns: write ACK / snapshot
Purpose: Quiz history + user stats persistence
Security Rules: deny all direct client access
Access: backend only via Admin SDK
Cloud Data Store
Quiz History · User Stats
ChromaDB
Protocol: In-process Python client (no network)
Auth: None — local process, no exposed port
Collection: donor_emails Metric: cosine
Embedding dim: 384 (MiniLM-L6-v2)
Returns: { docs, embeddings, distances }
Latency: <50ms (in-memory HNSW index)
Privacy: email content purged after 30 days
Embeddings non-reversible → privacy by design
Vector Store (Local)
Donor Email Embeddings
Arrow Key:
Frontend call
AI call
DB operation
External service
Return / Response
Sections:
5.1 Frontend ↔ Backend
5.2 Backend ↔ AI
5.3 Backend ↔ Database
5.4 External Services
QuizBot HLD — Module Interaction Diagram · Sections 5.1 – 5.4

4. API Design Specifications

All API endpoints use JSON request and response bodies. All protected endpoints require a valid JWT Bearer token in the Authorization header. Base URL: http://localhost:8000/api

4.1 Authentication Endpoints

Method	Endpoint	Auth	Description
POST	`/auth/register`	None	Register new user. Accepts {email, password, full_name}. Creates Firebase user and SQLite record. Returns JWT.
POST	`/auth/login`	None	Login with email/password. Verifies Firebase ID Token. Returns JWT and user profile.
POST	`/auth/refresh`	JWT	Refresh expired JWT using Firebase refreshToken. Returns new JWT.
POST	`/auth/logout`	JWT	Invalidate session. Clears server-side state. Client clears localStorage.

4.2 Quiz Endpoints

Method	Endpoint	Auth	Description
POST	`/quiz/generate`	JWT	Generate quiz from donor email. Body: {email_content, num_questions}. Returns quiz_id + questions array.
POST	`/quiz/submit`	JWT	Submit quiz answers. Body: {quiz_id, answers[]}. Returns score, explanations, AI summary.
GET	`/quiz/{quiz_id}`	JWT	Retrieve a specific quiz by ID with all questions.
GET	`/quiz/history`	JWT	List all quizzes for the authenticated user.

4.3 Analytics Endpoints

Method	Endpoint	Auth	Description
GET	`/analytics/history`	JWT	Get quiz history from Firebase Realtime DB. Returns list of quiz attempts with scores and dates.
GET	`/analytics/progress`	JWT	Get aggregated progress metrics — total quizzes, average score, accuracy rate, trend data.

5. Algorithm Design

5.1 Quiz Question Generation Algorithm

The quiz generation algorithm follows a structured pipeline that combines vector embedding, ChromaDB storage, and LLM prompt engineering to produce contextually accurate MCQ questions from donor email content.

Step	Operation
1	Receive email_content (string) and num_questions (int) from validated Pydantic request model.
2	Encode email_content using SentenceTransformer.encode() → produces 384-dimensional Float32 vector.
3	Upsert 384-dim vector into ChromaDB donor_emails collection with quiz_id as document ID.
4	Build structured Gemini prompt: "Generate {n} MCQ questions from this donor email. Return JSON array with fields: question, option_a, option_b, option_c, option_d, correct_option (A/B/C/D)."
5	Call Gemini 2.5 Flash generate_content() with exponential backoff retry logic (max 5 attempts).
6	Parse JSON response. Validate that all required fields (question, options, correct_option) are present for each question.
7	If JSON parsing fails, trigger fallback: return a pre-defined "service unavailable" quiz with generic questions.
8	Save quiz record to SQLite quizzes table. Save each question to questions table with order_index.
9	Return quiz_id and questions array to frontend.

5.2 Answer Evaluation Logic

Each submitted answer is evaluated individually by Gemini 2.5 Flash. The evaluation prompt includes the original question, all four options, the correct answer, and the user's selected option to ensure contextual explanation quality.

- Input: {question_text, option_a, option_b, option_c, option_d, correct_option, user_selected}
- Gemini Prompt: "Evaluate this quiz answer. Correct: {correct_option}. User selected: {user_selected}. Return JSON: {is_correct: bool, explanation: string}. Explanation must reference the donor email context."
- Output: {is_correct: bool, explanation: string} — validated and stored per question_result record
- After all N answers are evaluated, a second Gemini call generates the overall performance summary using the complete results array.
- Summary Prompt: "Based on {score}/{total} with these results, generate a 2-paragraph performance summary. Score >= 80 → congratulate. Score < 80 → encourage improvement. Focus on non-profit concepts."

5.3 Vector Similarity Search

ChromaDB uses HNSW (Hierarchical Navigable Small World) graph indexing for approximate nearest-neighbor search. The all-MiniLM-L6-v2 model produces 384-dimensional vectors optimized for semantic sentence similarity.

- Embedding Model: SentenceTransformer("all-MiniLM-L6-v2") — loads once at application startup for performance.
- Vector Dimensions: 384 (Float32). Each donor email produces exactly one 384-element vector.

- Similarity Metric: Cosine similarity. Range: [-1, 1]. Higher values indicate greater semantic similarity.
- HNSW Index: Enables sub-linear search time O(log N) vs linear O(N) for brute-force. Suitable for local deployment.
- Query Flow: New email encoded → cosine similarity compared against all stored vectors → top-K similar emails retrieved → context provided to Gemini for better question alignment.
- Purge Policy: Embeddings older than 30 days are automatically deleted to manage local storage and user privacy.

5.4 Caching Strategy

QuizBot v1.0 does not implement a dedicated caching layer. The following optimizations are applied as lightweight caching alternatives:

- Sentence Transformer Model: Loaded once at FastAPI startup (lifespan event) and held in memory for the duration of the server process. This eliminates repeated model loading overhead (~2-3 seconds per request).
- ChromaDB Client: Initialized once at startup with a persistent directory. Collection creation is idempotent — if the collection already exists, the existing collection is returned.
- SQLAlchemy Session: A single database session is created per HTTP request and disposed of after the response is returned.
- Future Enhancement: Redis caching planned for frequently accessed analytics data and repeated quiz generation requests on identical email content.

6. Database Design

QuizBot uses a multi-database strategy. SQLite manages all structured relational data locally. Firebase Realtime Database provides cloud-based history storage. ChromaDB stores vector embeddings for semantic retrieval.

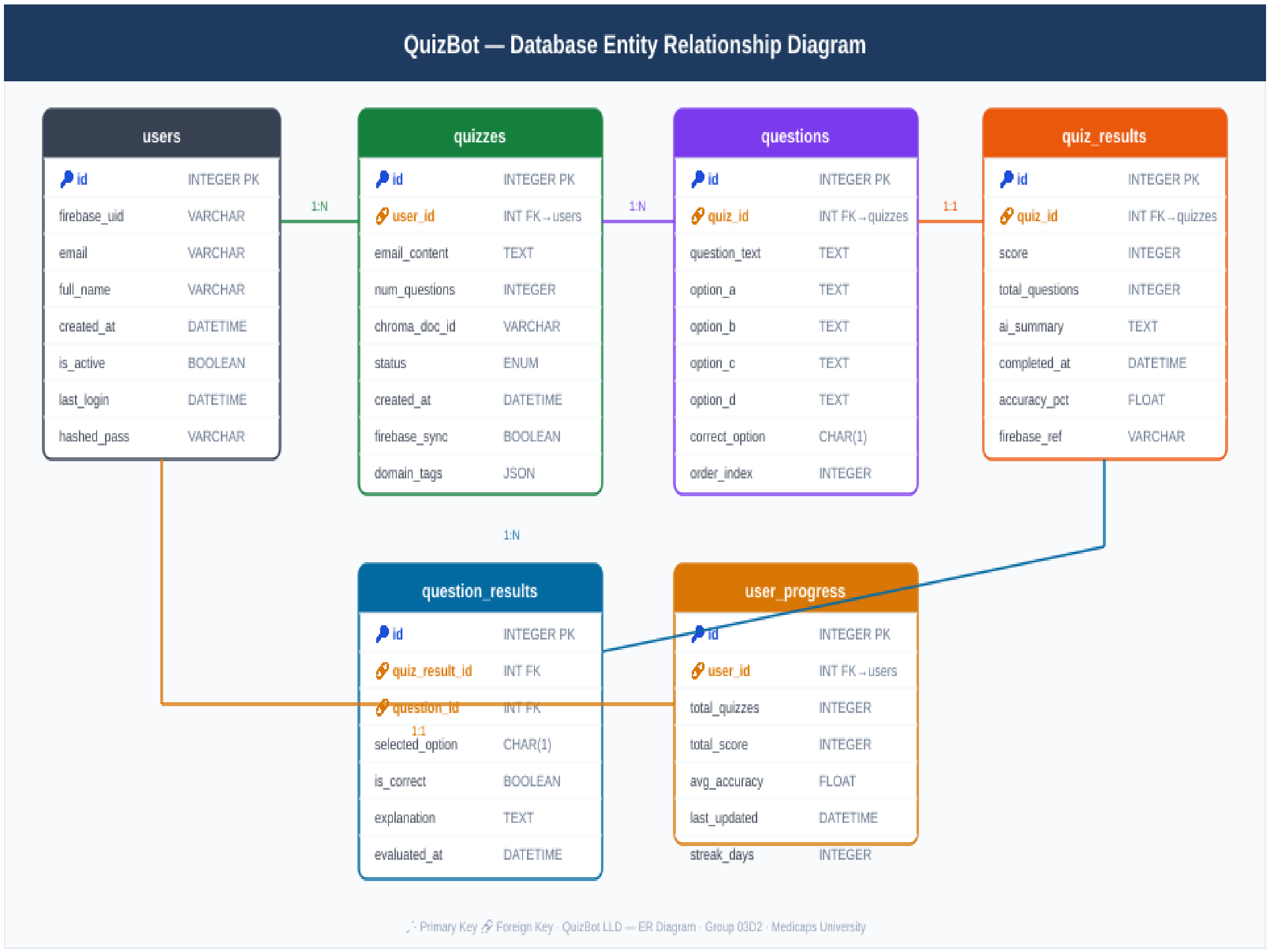


Figure 3: QuizBot Entity Relationship Diagram — SQLite Schema

6.1 SQLite Schema

- users — Stores registered user accounts. firebase_uid links to Firebase Authentication. All other tables reference users.id via FK.
- quizzes — Stores each quiz attempt. email_content holds the donor email text. chroma_doc_id links to the ChromaDB embedding. domain_tags (JSON) store Gemini-extracted non-profit theme tags.
- questions — Stores individual MCQ questions for each quiz. option_a through option_d store the four choices. correct_option stores the letter A, B, C, or D.
- quiz_results — Stores the overall result for each completed quiz. accuracy_pct is computed as (score / total_questions) × 100. firebase_ref stores the path written to Firebase Realtime DB.
- question_results — Stores per-question evaluation results. explanation holds the AI-generated feedback text. Links to both quiz_results and questions via FK.
- user_progress — Aggregated progress metrics per user. Updated after every quiz completion. Stores total_quizzes, total_score, avg_accuracy, streak_days.

6.2 Firebase Realtime Database Structure

Path	Contents
`quiz_history/{uid}/{quiz_id}`	score, total_questions, accuracy_pct, ai_summary, completed_at, results[]
`users/{uid}`	email, full_name, total_quizzes, total_score, last_active

6.3 ChromaDB Collection Design

Property	Value
Collection Name	`donor_emails`
Embedding Dimensions	`384 (Float32)`
Distance Metric	`cosine`
Index Type	`HNSW (default)`
Document ID	`quiz_id (string)`
Metadata	`{user_id, created_at, num_questions, domain_tags}`
Purge Policy	`Delete embeddings older than 30 days`

7. Security Design

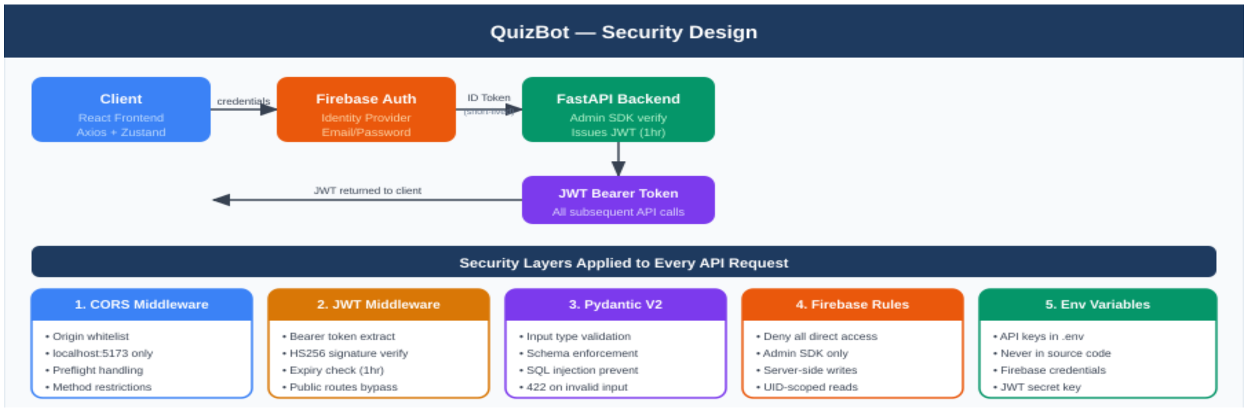


Figure 4: QuizBot Security Design — Dual-Token Auth Model and Security Layers

7.1 Dual-Token Authentication Model

- Step 1 — Client Login: User submits email/password to Firebase Authentication via signInWithEmailAndPassword(). Firebase verifies credentials and returns a short-lived Firebase ID Token.
- Step 2 — Backend Verification: The ID Token is sent to FastAPI POST /auth/login. The backend verifies it using Firebase Admin SDK verify_id_token(). This confirms the user is who they claim to be.
- Step 3 — JWT Issuance: After successful verification, the backend generates a backend JWT using python-jose (HS256 algorithm, 1-hour expiry). This JWT contains the user's uid and email as claims.
- Step 4 — API Authorization: All subsequent requests include the JWT in the Authorization: Bearer {token} header. The JWT Middleware extracts, verifies the signature, and checks expiry before any route handler executes.
- Step 5 — Token Refresh: When the JWT expires (401 response), the Axios interceptor automatically calls Firebase getIdToken(true) to refresh the Firebase ID Token, then calls POST /auth/refresh to obtain a new backend JWT.

7.2 Security Controls Summary

Control	Implementation
CORS	FastAPI CORSMiddleware — allows only localhost:5173. Blocks all other origins. Preflight handled automatically.
JWT Middleware	Applied to all routes except /auth/register and /auth/login. Signature verified with HS256. Expiry strictly enforced.
Input Validation	Pydantic V2 models on all request bodies. Invalid input returns HTTP 422 Unprocessable Entity before any business logic runs.
SQL Injection	All database queries use SQLAlchemy ORM parameterised queries. Raw SQL is never used.
Firebase Rules	Firebase Security Rules set to deny all direct client reads/writes. All Firebase access goes through the FastAPI backend via Admin SDK.
Session Termination	Logout calls Firebase signOut(), clears JWT from localStorage, and resets Zustand auth store. All API calls after logout fail with 401.

8. Summary

This Low-Level Design document provides a complete technical specification for QuizBot — Non-Profit Quiz/Tutor Bot. The design demonstrates how modern web technologies, cloud services, and AI capabilities are integrated into a coherent, secure, and maintainable educational assessment platform.

The 4-layer architecture cleanly separates concerns across the Presentation, Business Logic, AI, and Data layers. The dual-token authentication model (Firebase ID Token + backend JWT) provides robust, stateless security. The multi-database strategy (SQLite + Firebase Realtime DB + ChromaDB) ensures the right storage technology is applied to each type of data — structured relational data, cloud analytics, and vector embeddings respectively.

The AI integration architecture — combining Google Gemini 2.5 Flash for language understanding and generation with Sentence Transformers for semantic embedding — represents a production-quality approach to building AI-powered applications. The exponential backoff retry logic ensures system reliability under external API constraints.

Key Design Decisions

- ChromaDB over FAISS: ChromaDB was selected for its persistent storage, built-in metadata filtering, and production-ready API compared to FAISS which requires manual index persistence management.
- SQLite over PostgreSQL (v1.0): SQLite provides zero-configuration local deployment suitable for the current local application scope. PostgreSQL migration is designed into the architecture via SQLAlchemy ORM abstraction.
- Dual-Token over Single Token: The combination of Firebase ID Token and backend JWT allows the system to leverage Firebase's proven authentication infrastructure while maintaining full control over API authorization logic.
- Pydantic V2 over manual validation: Pydantic V2 provides type-safe, automatic request validation with zero-boilerplate error handling (HTTP 422) across all endpoints.

Future Enhancements Considered in Design

- Redis caching layer for analytics data and frequent Gemini prompts.
- PostgreSQL migration via SQLAlchemy ORM swap (no query changes required).
- Docker + Docker Compose for containerised deployment (Dockerfile and compose.yml planned).
- Cloud deployment on Google Cloud Run (FastAPI) and Vercel (React frontend).
- WebSocket support for real-time quiz sessions and collaborative features.