

Project – High 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

Table of Contents

1. **Introduction**
 - 1.1. Scope of the document
 - 1.2. Intended Audience
 - 1.3. System overview
2. **System Design**
 - 2.1. Application Design
 - 2.2. Process Flow
 - 2.3. Information Flow
 - 2.4. Components Design
 - 2.5. Key Design Considerations
 - 2.6. API Catalogue.
3. **Data Design**
 - 3.1. Data Model
 - 3.2. Data Access Mechanism
 - 3.3. Data Retention Policies
 - 3.4. Data Migration
4. **Interfaces**
5. **State and Session Management**
6. **Caching**
7. **Non-Functional Requirements**
 - 7.1. Security Aspects
 - 7.2. Performance Aspects
8. **References**

1. Introduction

The non-profit sector relies heavily on donor relationships, fundraising strategies, and community engagement. Professionals and volunteers working in this space need to continuously update their knowledge of best practices — from donor stewardship to impact reporting and ethical fundraising.

However, traditional learning methods lack personalization, contextual relevance, and immediate feedback. There is a clear gap between the information available in day-to-day donor communications and the ability to convert that information into structured learning experiences.

QuizBot bridges this gap by transforming real-world donor emails into intelligent, AI-powered assessments that test comprehension, reinforce learning, and track progress over time. **QuizBot** is an AI-powered web application built specifically for the **non-profit sector**. It takes donor emails as input and automatically converts them into intelligent multiple-choice quizzes using Google Gemini AI. Users take the quiz, get evaluated, receive detailed explanations, and track their learning progress over time.

Problem Statement

Develop an interactive AI-driven educational bot for the Non-Profit domain that engages users in targeted assessments based on donor emails, evaluates their responses using a Large Language Model, and provides deep contextual explanations for incorrect answers — creating a personalized learning loop that bridges knowledge gaps and reinforces industry-specific concepts.

Project Objectives

- Accept donor email content as input and generate contextual MCQ-based quizzes automatically
- Use Google Gemini LLM to generate questions, evaluate answers, and produce personalized feedback
- Store donor email embeddings in a Vector Database (ChromaDB/FAISS) for semantic search and retrieval
- Provide detailed AI-generated explanations for each correct and incorrect answer
- Track user quiz history, scores, and learning progress through Firebase Realtime Database
- Display analytics dashboards showing performance trends, accuracy rates, and achievements
- Implement secure user authentication via Firebase Authentication

1.1 Scope of the document

This High-Level Design (HLD) document describes the architectural design and system structure of **QuizBot** — an AI-driven educational assessment platform for the non-profit domain. The document covers:

- Overall system architecture and component interactions
- Technology stack and design decisions
- Data flow between frontend, backend, and third-party services
- Integration design for AI (Google Gemini), Vector Database (ChromaDB/FAISS), and Firebase
- Security, authentication, and API design at a high level

This document does not cover low-level implementation details such as individual function logic, database query optimization, or UI component internals — those are addressed in the LLD document.

1.2 Intended Audience

Audience	Purpose
Software Architects	To review and validate the overall system design
Backend Developers	To understand service boundaries, API contracts, and data flow
Frontend Developers	To understand integration points with backend APIs and Firebase
Project Managers	To understand system scope, components, and dependencies
QA Engineers	To understand system boundaries for testing strategy
Academic Evaluators	To assess technical depth and design quality of the project

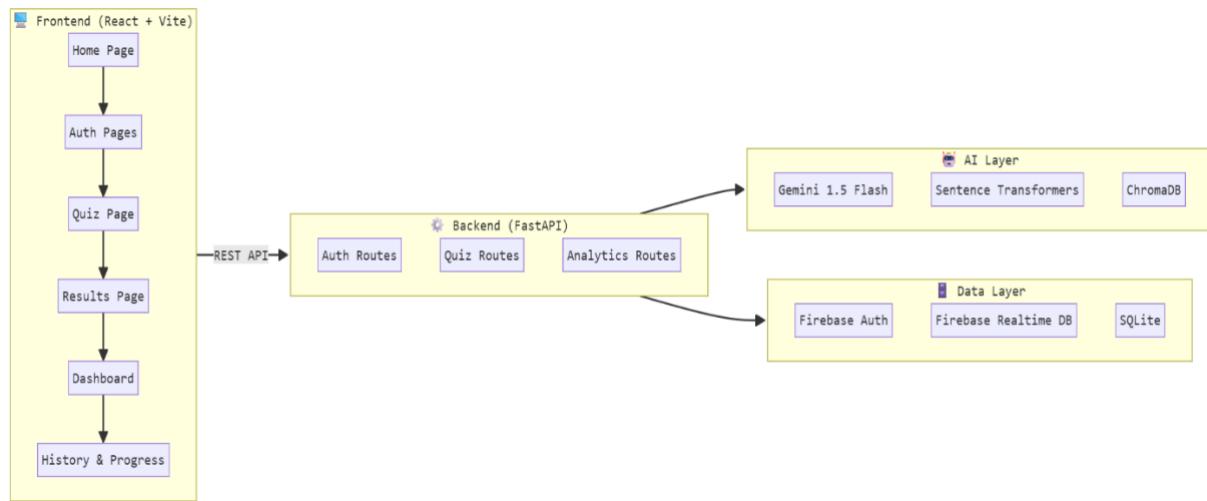
1.3 System overview

QuizBot is a full-stack, AI-powered web application designed for non-profit sector education. It allows users to paste donor emails and automatically generates contextual multiple-choice quizzes using a Large Language Model. The system evaluates responses, provides detailed AI-generated explanations, and tracks user learning progress over time.

Core capabilities:

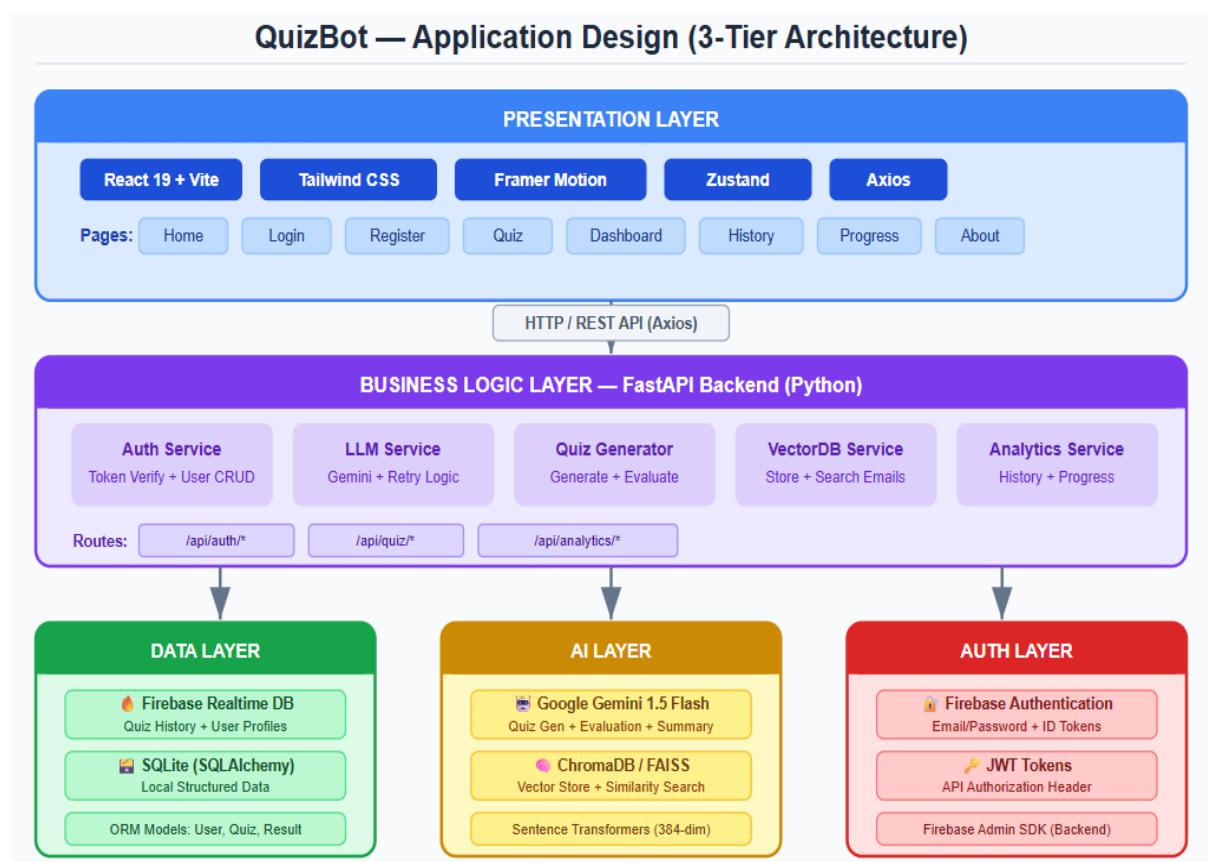
- **AI Quiz Generation** — Google Gemini API processes donor email content and generates 3–10 contextual MCQs
- **Vector Semantic Search** — ChromaDB stores email embeddings using Sentence Transformers for similarity matching
- **Smart Evaluation** — LLM evaluates each answer and generates personalized explanations
- **User Authentication** — Firebase Authentication handles secure email/password login and registration
- **Progress Tracking** — Firebase Realtime Database stores quiz history, scores, and analytics per user
- **Analytics Dashboard** — Users see performance trends, accuracy rates, and achievements

2. System Design



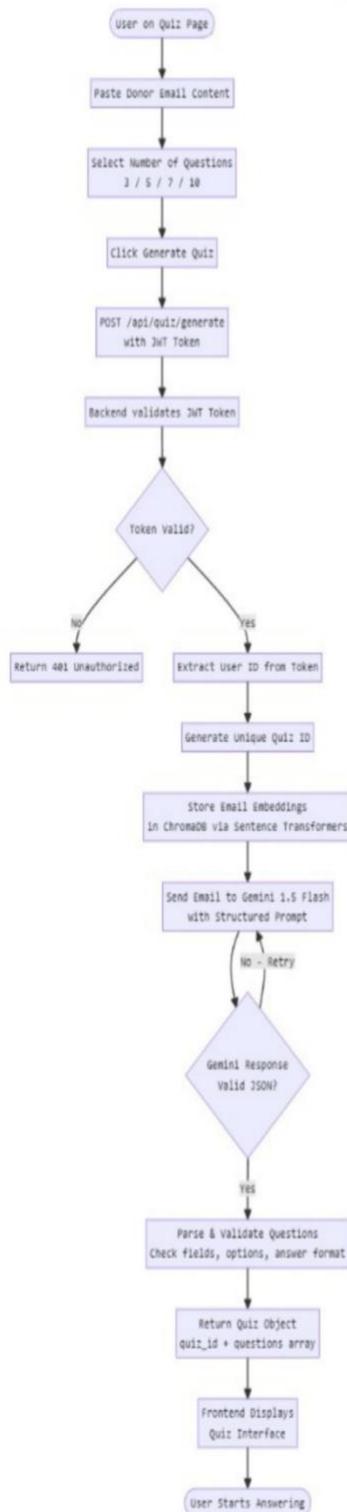
2.1 Application Design

QuizBot follows a 3-Tier Client-Server Architecture with clear separation between Presentation, Business Logic, and Data layers.

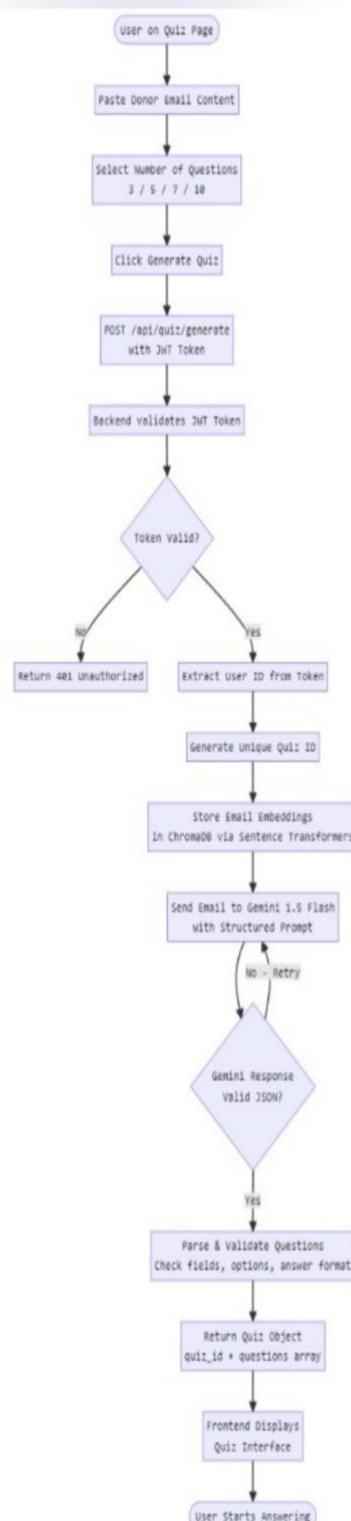


2.2 Process Flow

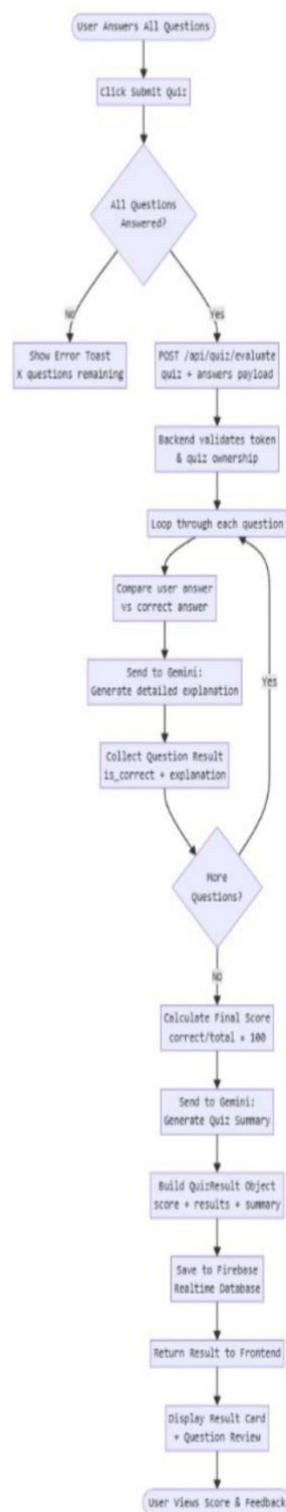
2.2.1 User Registration & Login Flow



2.2.2 Quiz Generation Flow

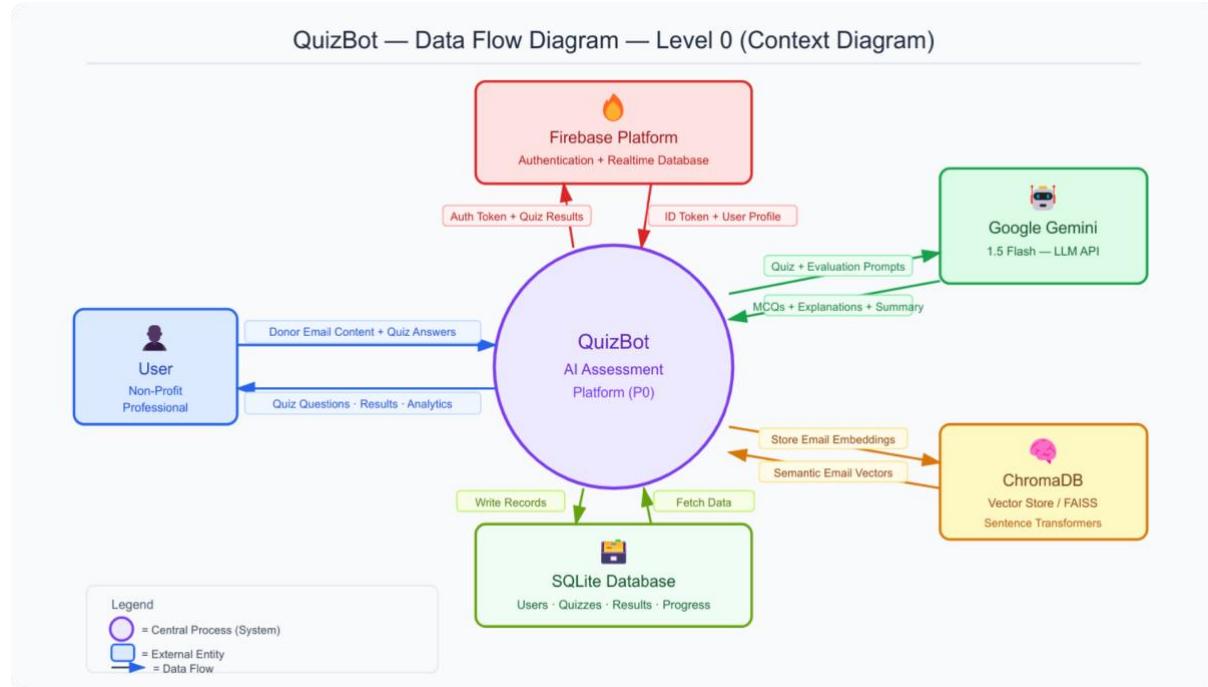


2.2.3 Quiz Submission & Evaluation Flow

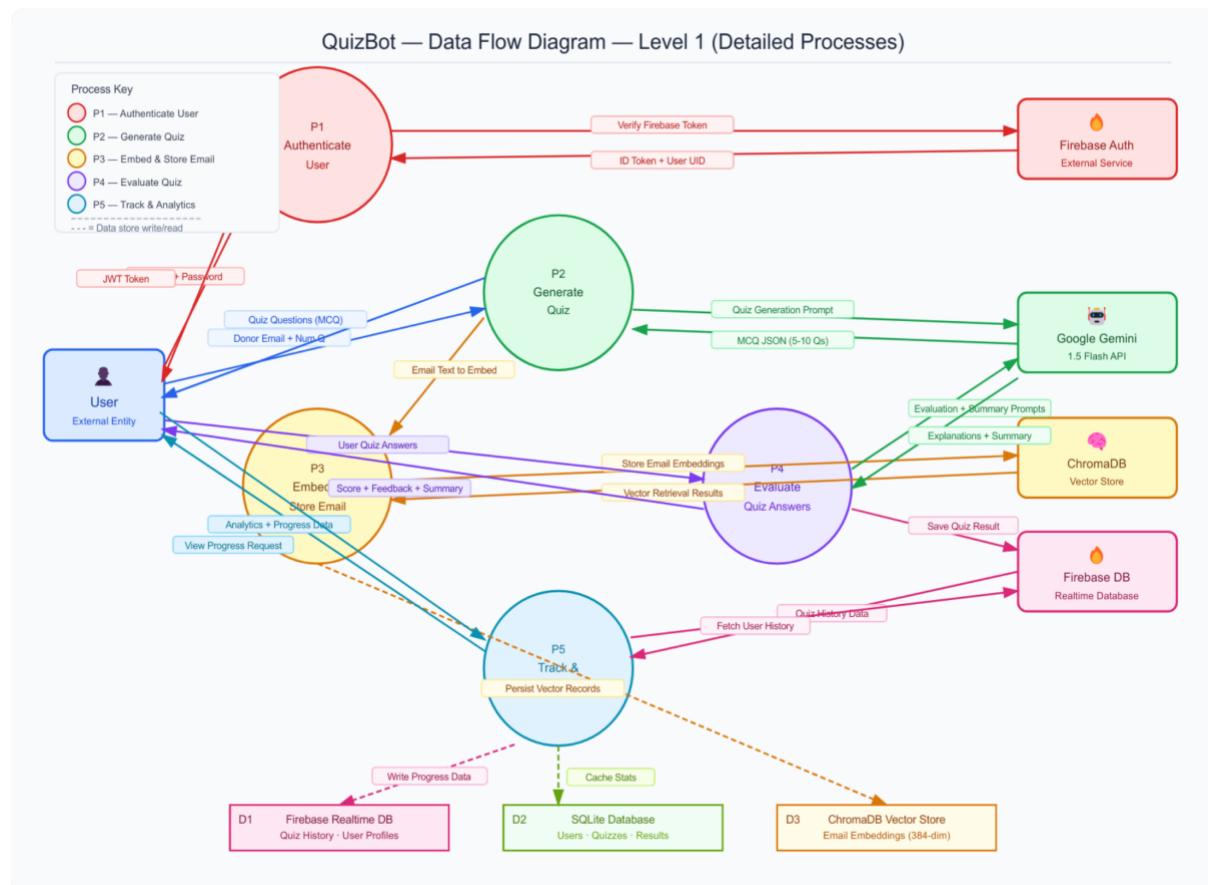


2.3 Information Flow

2.3.1 Data Flow Diagram — Level 0 (Context)



2.3.2 Data Flow Diagram — Level 1 (Detailed)

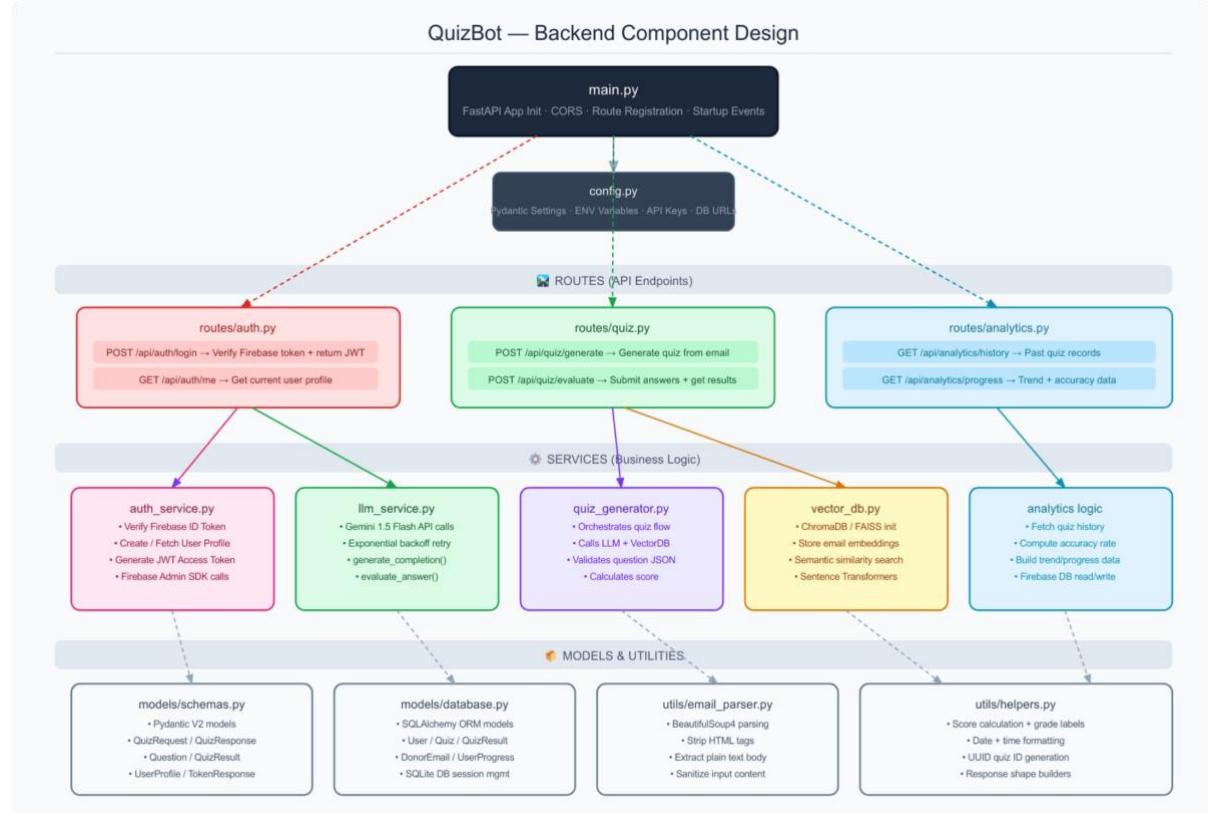


2.3.3 Information Flow Summary Table

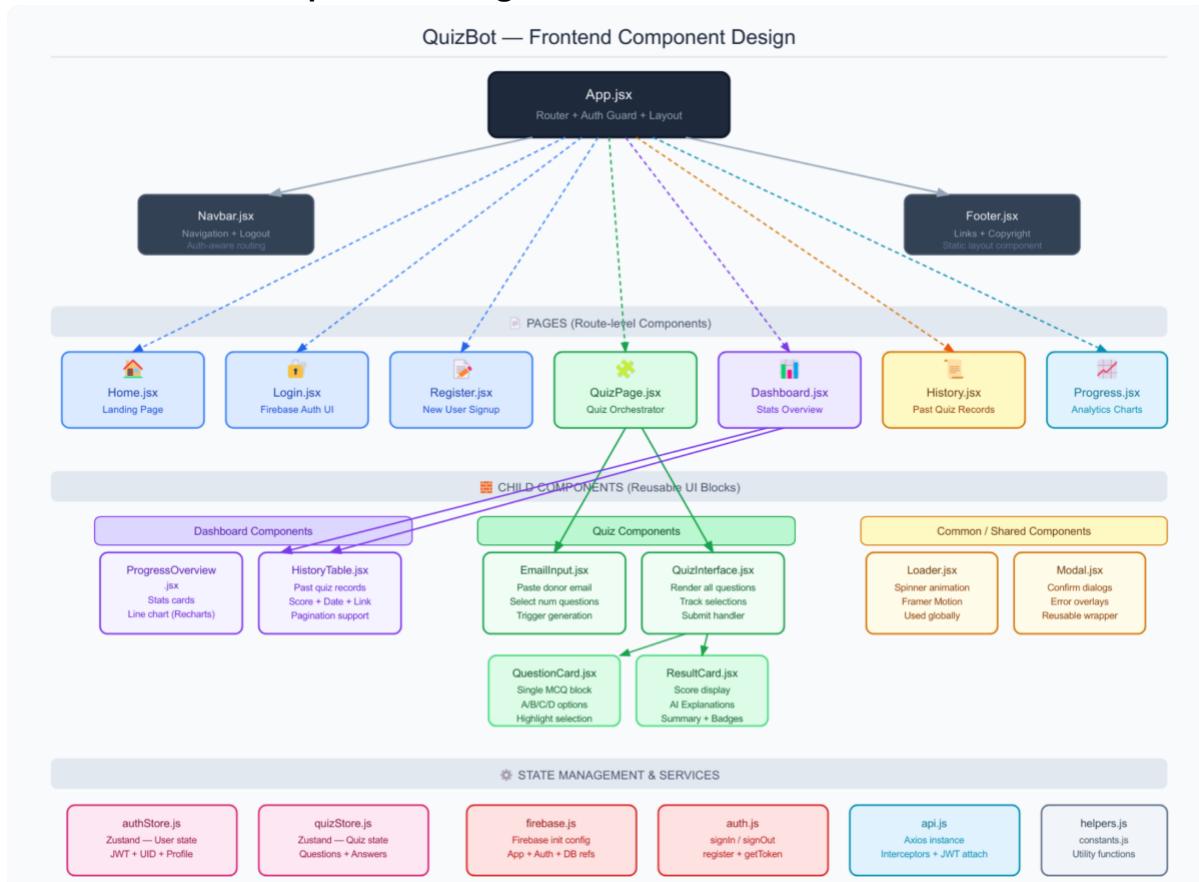
Data	Source	Destination	Format	When
Donor Email Text	User Input	Backend → ChromaDB + Gemini	Plain Text	Quiz Generation
Email Embeddings	Sentence Transformers	ChromaDB	384-dim Vector	Quiz Generation
Quiz Questions	Google Gemini	Frontend	JSON Array	After Generation
User Answers	Frontend	Backend	JSON Array	Quiz Submission
Answer Explanations	Google Gemini	Frontend	Plain Text	After Submission
Quiz Results	Backend	Firebase Realtime DB	JSON Object	After Evaluation
Auth Token	Firebase Auth	Frontend → Backend	JWT String	Login/Register
User Profile	Firebase Realtime DB	Backend → Frontend	JSON Object	Dashboard Load
Progress Analytics	Firebase Realtime DB	Backend → Frontend	JSON Object	Progress Page

2.4 Components Design

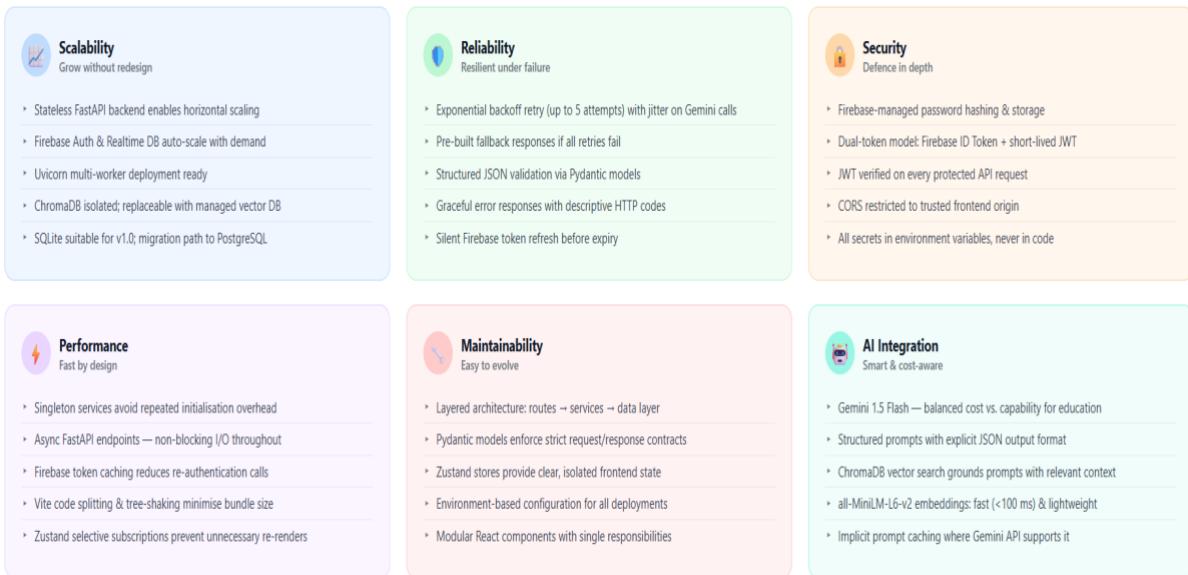
2.4.1 Backend Components Design



2.4.2 Frontend Components Design



2.5 Key Design Considerations



2.6 API Catalogue

Base URL: <http://localhost:8000>

Authentication APIs - /api/auth

Method	Endpoint	Description	Auth Required
POST	/api/auth/login	Verify Firebase token, return JWT + user profile	No
GET	/api/auth/me	Get current authenticated user profile	Yes

Quiz APIs — /api/quiz

Method	Endpoint	Description	Auth Required
POST	/api/quiz/generate	Generate quiz from donor email	Yes
POST	/api/quiz/evaluate	Submit answers and get results	Yes

Analytics APIs — /api/analytics

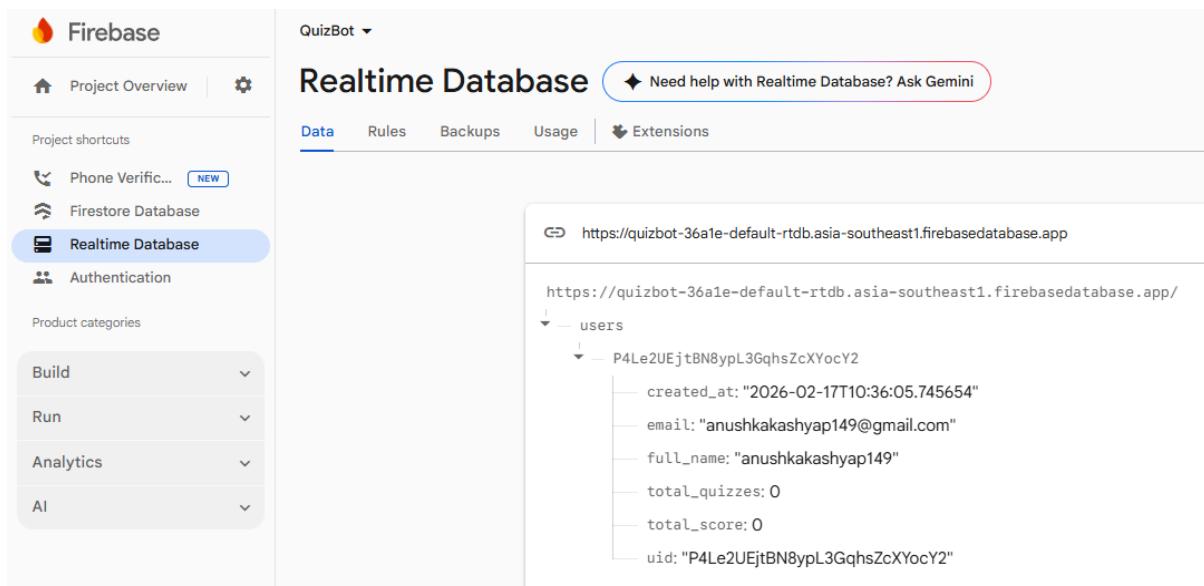
Method	Endpoint	Description	Auth Required
GET	/api/analytics/history	Get all past quiz records for user	Yes
GET	/api/analytics/progress	Get progress analytics and trend data	Yes
GET	/api/analytics/stats	Get combined user stats + analytics	Yes

3. Data Design

3.1 Data Model

QuizBot uses a **multi-database strategy** - Firebase Realtime Database for cloud-synced user and quiz data, SQLite for local structured storage, and ChromaDB for vector embeddings. Each database is chosen for its specific strengths.

3.1.1 Firebase Realtime Database Schema Firebase stores data as a hierarchical JSON tree. QuizBot uses two root nodes - users and quiz_history.

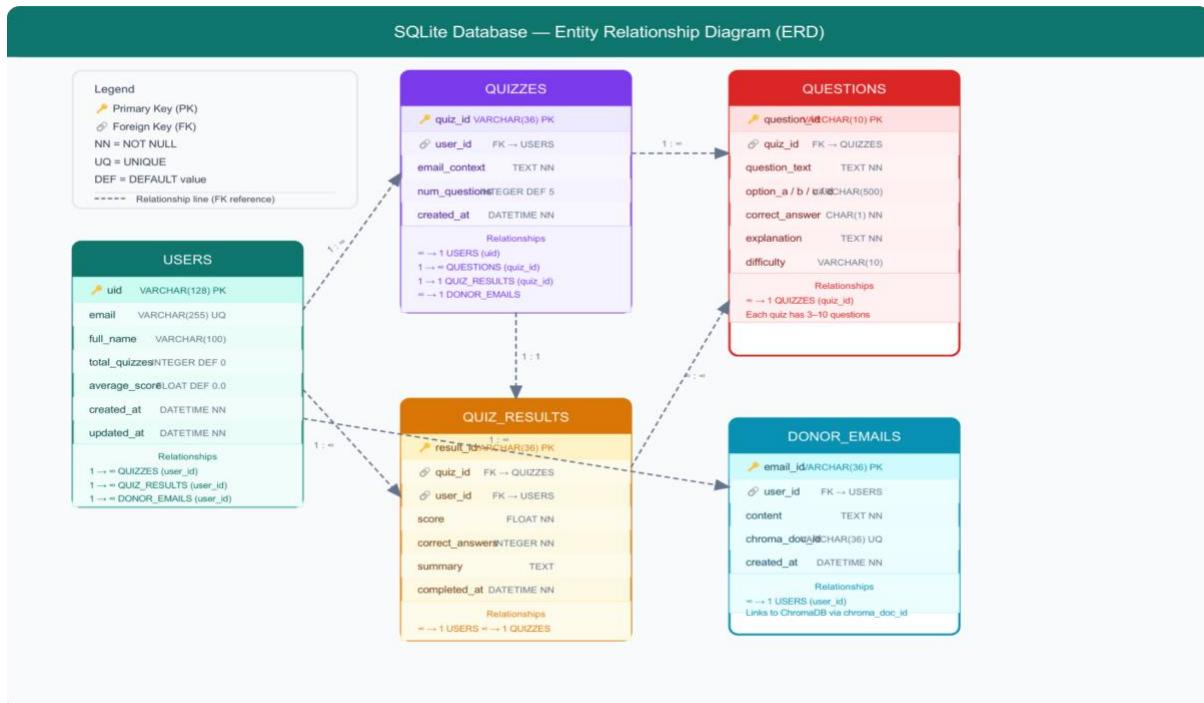


The screenshot shows the Firebase Realtime Database interface. On the left, the sidebar has 'Project Overview' and 'Realtime Database' selected. The main area shows the 'Realtime Database' tab with a URL: <https://quizbot-36a1e-default-rtdb.firebaseio.com/>. Below it, the 'Data' tab is selected, showing the 'users' node structure:

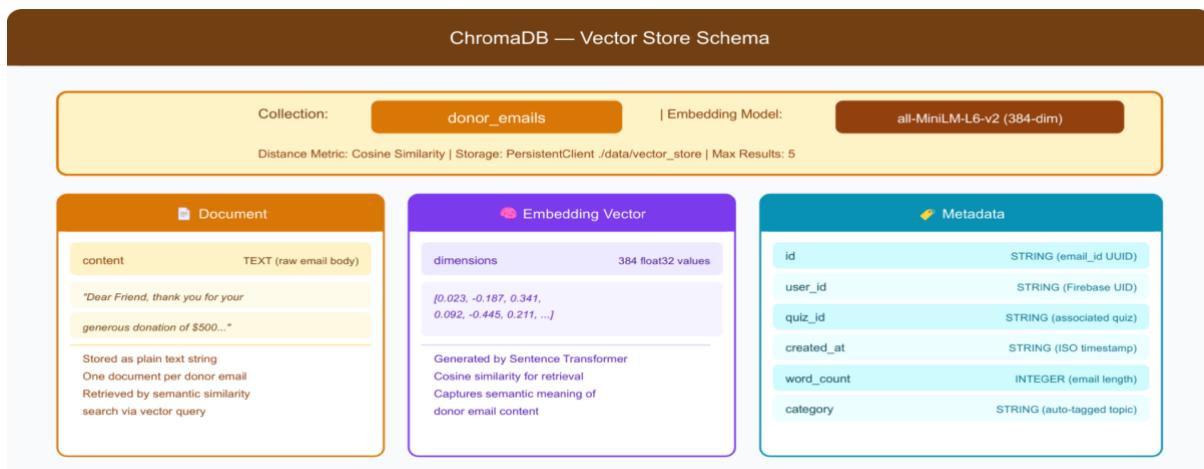
```

https://quizbot-36a1e-default-rtdb.firebaseio.com/
  users
    P4Le2UEjtBN8ypL3GqhsZcXYocY2
      created_at: "2026-02-17T10:36:05.745654"
      email: "anushkakashyap149@gmail.com"
      full_name: "anushkakashyap149"
      total_quizzes: 0
      total_score: 0
      uid: "P4Le2UEjtBN8ypL3GqhsZcXYocY2"
  
```

3.1.2 SQLite Database Schema

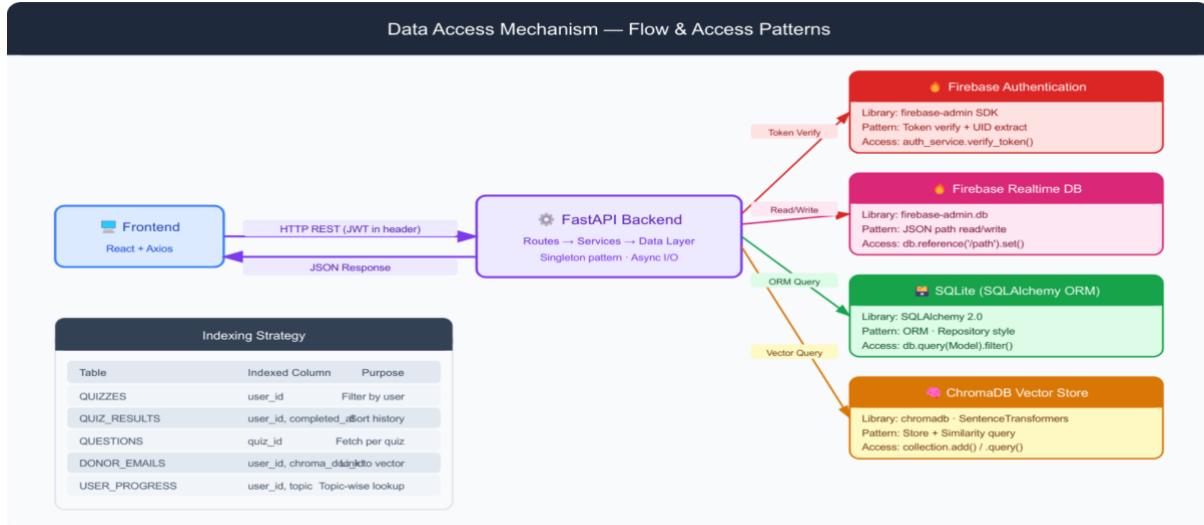


3.1.3 ChromaDB Vector Store Schema

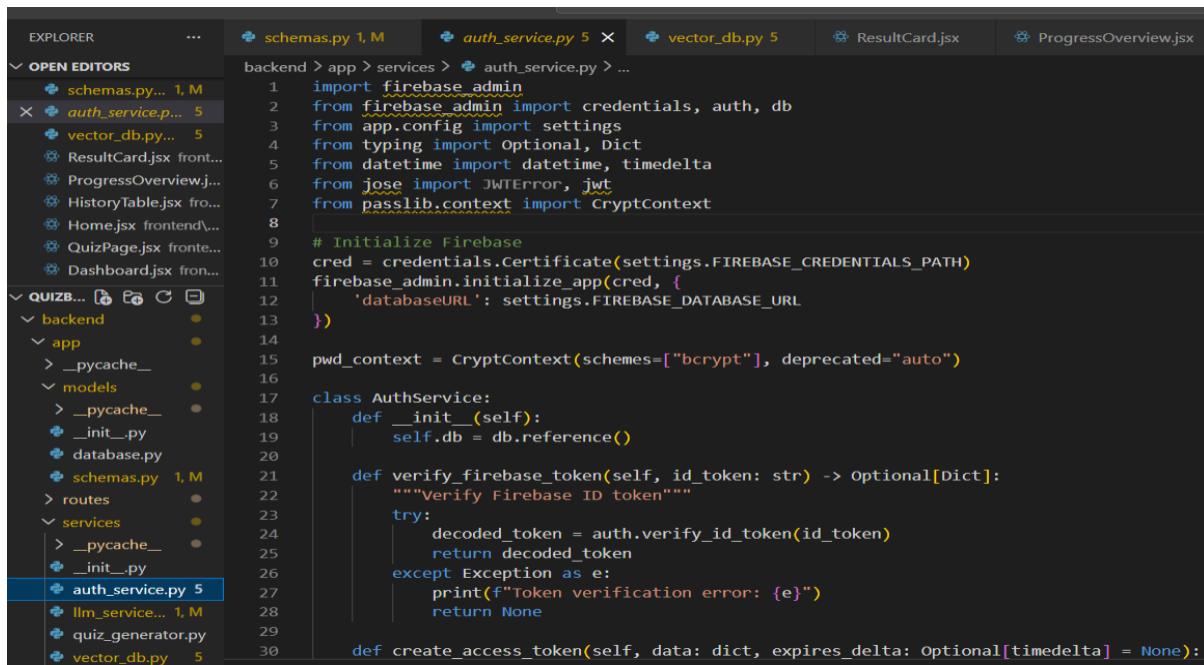


3.2 Data Access Mechanism

QuizBot accesses data through distinct mechanisms per database, all orchestrated by the FastAPI backend services.



3.2.1 Firebase Realtime Database Access



```

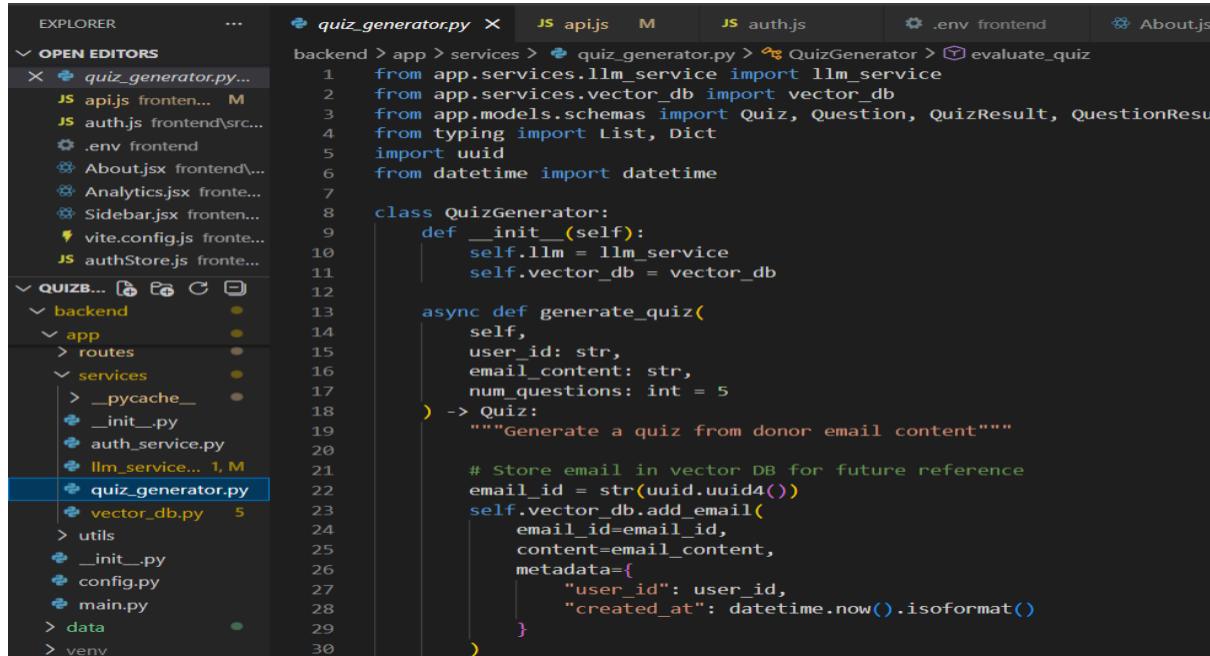
EXPLORER ... schemas.py 1, M auth_service.py 5 vector_db.py 5 ResultCard.jsx ProgressOverview.jsx
OPEN EDITORS
schemas.py... 1, M auth_service.p... 5
auth_service.p... 5
vector_db.py... 5
ResultCard.jsx front...
ProgressOverview.j...
HistoryTable.jsx fro...
Home.jsx frontend\...
QuizPage.jsx front...
Dashboard.jsx front...
QUIZ... 🔍 🔍 🔍 🔍
backend
app
__pycache__
models
__pycache__
__init__.py
database.py
schemas.py 1, M
routes
services
__pycache__
__init__.py
auth_service.py 5
llm_service... 1, M
quiz_generator.py
vector_db.py 5
backend > app > services > auth_service.py > ...
1 import firebase_admin
2 from firebase_admin import credentials, auth, db
3 from app.config import settings
4 from typing import Optional, Dict
5 from datetime import datetime, timedelta
6 from jose import JWTError, jwt
7 from passlib.context import CryptContext
8
# Initialize Firebase
9 cred = credentials.Certificate(settings.FIREBASE_CREDENTIALS_PATH)
10 firebase_admin.initialize_app(cred, {
11     'databaseURL': settings.FIREBASE_DATABASE_URL
12 })
13
14 pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
15
16 class AuthService:
17     def __init__(self):
18         self.db = db.reference()
19
20     def verify_firebase_token(self, id_token: str) -> Optional[Dict]:
21         """Verify Firebase ID token"""
22         try:
23             decoded_token = auth.verify_id_token(id_token)
24             return decoded_token
25         except Exception as e:
26             print(f"Token verification error: {e}")
27             return None
28
29     def create_access_token(self, data: dict, expires_delta: Optional[timedelta] = None):
30

```

Access Rules:

- All reads/writes are server-side only via Firebase Admin SDK
- Frontend never directly reads/writes the Realtime DB — always goes through FastAPI
- Firebase Security Rules set to deny all direct client access

3.2.2 ChromaDB Vector Access



```

EXPLORER      ...
OPEN EDITORS  ...
QUIZB...      ...
backend > app > services > quiz_generator.py > QuizGenerator > evaluate_quiz
1   from app.services.llm_service import llm_service
2   from app.services.vector_db import vector_db
3   from app.models.schemas import Quiz, Question, QuizResult, QuestionResu
4   from typing import List, Dict
5   import uuid
6   from datetime import datetime
7
8   class QuizGenerator:
9       def __init__(self):
10           self.llm = llm_service
11           self.vector_db = vector_db
12
13       async def generate_quiz(
14           self,
15           user_id: str,
16           email_content: str,
17           num_questions: int = 5
18       ) -> Quiz:
19           """Generate a quiz from donor email content"""
20
21           # Store email in vector DB for future reference
22           email_id = str(uuid.uuid4())
23           self.vector_db.add_email(
24               email_id=email_id,
25               content=email_content,
26               metadata={
27                   "user_id": user_id,
28                   "created_at": datetime.now().isoformat()
29               }
30       )

```

3.3 Data Retention Policies

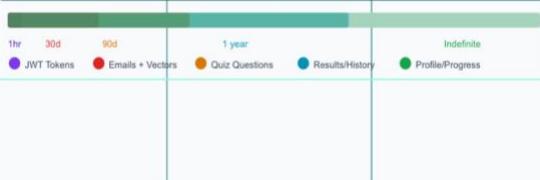
3.4 Data Migration

Data Retention Policies & Migration Strategy

3.3 Data Retention Policies

Data Type	Retention	Deletion Trigger
Firebase — User Profile	Indefinite	Account deletion (manual)
Firebase — Quiz History	1 Year	Auto-purge / user request
SQLite — Quiz Questions	90 Days	Auto-purge after 90 days
SQLite — Quiz Results	1 Year	User request / account delete
SQLite — Donor Emails (raw)	30 Days	Auto-purge (privacy policy)
ChromaDB — Embeddings	30 Days	Synced with donor email purge
JWT Tokens	1 Hour	Auto-expiry by Firebase
SQLite — User Progress	Indefinite	Account deletion only

Data Lifecycle Timeline



3.4 Migration Strategy

Migration Path 1: SQLite → PostgreSQL

- 1 Trigger Condition
Concurrent users > 50 · DB file > 1GB
- 2 Freeze Writes
Enable maintenance mode · pause requests
- 3 Export SQLite Dump
sqlite3 quizbot.db .dump > dump.sql
- 4 Create PostgreSQL Schema
Alembic migrations · psycopg2 driver
- 5 Update .env & Validate
DATABASE_URL → postgres:// · Run tests



Migration Path 2: ChromaDB → Pinecone / Weaviate

- 1 Export ChromaDB docs + embeddings
- 2 Serialize all 384-dim vectors + metadata to JSON
- 3 Create Pinecone index (dim=384, cosine)
- 4 Batch-upload vectors · update vector_db.py service

Migration Path 3: Firebase Realtime DB → Firestore

Trigger: Complex queries needed · Dataset exceeds RT DB limits
Change: db.reference('/path') → firestore.collection('name')
Effort: Medium — SDK switch + JSON tree → document model reshape

4. Interfaces

4.1 User Interfaces (UI)

QuizBot's frontend is a Single Page Application (SPA) built with React 19 and Vite. All interfaces are responsive, accessible, and transition-animated via Framer Motion.

Interface Inventory

Interface	Route	Access	Purpose
Home Page	/home	Public	Landing page, product overview, CTA
Login	/login	Public	Firebase email/password login
Register	/register	Public	New user account creation
Quiz Page	/quiz	Protected	Email input + quiz generation + taking
Dashboard	/dashboard	Protected	Stats overview + recent quizzes
History	/history	Protected	All past quiz records
Progress	/progress	Protected	Analytics charts + achievements

UI Component Interfaces

EmailInput Component

- Input: Large textarea for donor email paste
- Config: Dropdown to select number of questions (3 / 5 / 7 / 10)
- Action: "Generate Quiz" button → triggers POST /api/quiz/generate
- Output: Passes quiz object to QuizInterface

QuizInterface Component

- Input: Quiz object (quiz_id + questions array)
- Renders: Each question via QuestionCard
- Tracks: Selected answers in local state
- Action: "Submit Quiz" button → validates all answered → triggers POST /api/quiz/evaluate

QuestionCard Component

- Input: Single question object (text + 4 options + difficulty)
- Output: Selected option (A/B/C/D) emitted to parent QuizInterface
- Visual: Highlights selected option, shows difficulty badge

ResultCard Component

- Input: QuizResult object (score + results array + summary)
- Renders: Score percentage, AI summary, per-question explanation accordion
- Action: "Try Another Quiz" → resets quizStore and navigates to /quiz

Dashboard Components

- ProgressOverview: Displays total quizzes, average score, accuracy rate as stat cards + Recharts line graph
- HistoryTable: Paginated table of past quizzes with score, date, and view link

4.2 API Interface (Backend ↔ Frontend)

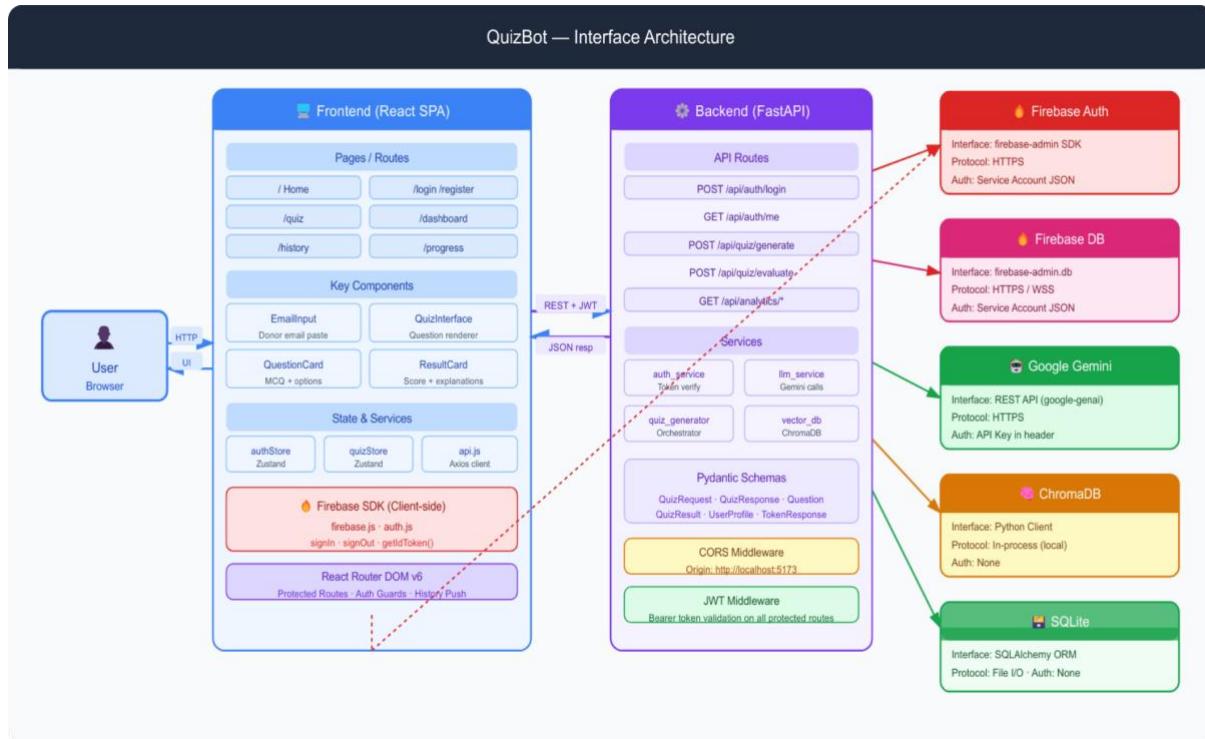
All API communication follows REST principles. Full catalogue covered in Section 2.6.
 Summary:

Category	Base Path	Methods	Auth
Authentication	/api/auth	POST, GET	Partial
Quiz	/api/quiz	POST	Required
Analytics	/api/analytics	GET	Required

4.3 External Service Interfaces

Service	Interface Type	Protocol	Auth Method
Google Gemini AI	REST API	HTTPS	API Key in header
Firebase Authentication	SDK + REST	HTTPS	Service Account JSON
Firebase Realtime DB	SDK	HTTPS/WSS	Service Account JSON
ChromaDB	Python Client	In-process	None (local)
SQLite	SQLAlchemy ORM	File I/O	None (local file)

4.4 Interface Flow Diagram



5. State and Session Management

5.1 Frontend State Management

QuizBot uses **Zustand** for global client-side state — lightweight, boilerplate-free, and React-hook based.

```

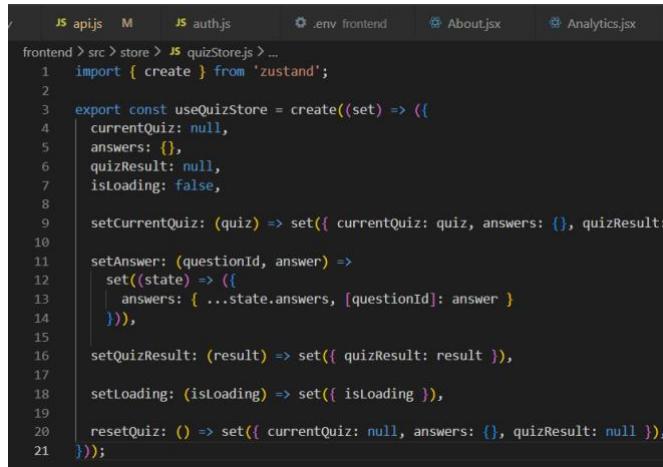
quizz_generatorory
  js api.js M js auth.js
  frontend > src > store > js authStore.js > ...
  1 import { create } from 'zustand';
  2 import { persist } from 'zustand/middleware';
  3
  4 export const useAuthStore = create(
  5   persist(
  6     set((...args) => {
  7       user: null,
  8       isAuthenticated: false,
  9       isLoading: true,
  10
  11       setUser: (user) => set({ user, isAuthenticated: !!user,
  12
  13       logout: () => set({ user: null, isAuthenticated: false
  14
  15       setLoading: (isLoading) => set({ isLoading }),
  16
  17       name: 'auth-storage',
  18
  19     });
  20
  21   );
  
```

authStore.js

Manages the authenticated user session on the frontend.

State Field	Type	Description
user	Object	Firebase user object (uid, email, displayName)
token	String	JWT access token from backend
isAuthenticated	Boolean	True when logged in
isLoading	Boolean	Auth check in progress

- **Actions:** setUser(), setToken(), logout(), checkAuth()
- **Persistence:** JWT stored in localStorage. On app load, checkAuth() reads the stored token and re-validates with Firebase before restoring the session.



```

JS api.js M JS auth.js .env frontend About.jsx Analytics.jsx
frontend > src > store > JS quizStore.js > ...
1 import { create } from 'zustand';
2
3 export const useQuizStore = create((set) => ({
4   currentQuiz: null,
5   answers: {},
6   quizResult: null,
7   isLoading: false,
8
9   setCurrentQuiz: (quiz) => set({ currentQuiz: quiz, answers: {} }, quizResult: null),
10
11   setAnswer: (questionId, answer) =>
12     set((state) => ({
13       answers: { ...state.answers, [questionId]: answer }
14     })),
15
16   setQuizResult: (result) => set({ quizResult: result }),
17
18   setLoading: (isLoading) => set({ isLoading }),
19
20   resetQuiz: () => set({ currentQuiz: null, answers: {}, quizResult: null })
21 }));

```

quizStore.js

Manages the active quiz session on the frontend.

State Field	Type	Description
currentQuiz	Object	Full quiz object (quiz_id + questions)
selectedAnswers	Object	Map of question_id → selected option
quizResult	Object	Evaluation result after submission
isGenerating	Boolean	Quiz generation in progress
isSubmitting	Boolean	Evaluation in progress

Actions: setQuiz(), setAnswer(), setResult(), resetQuiz()

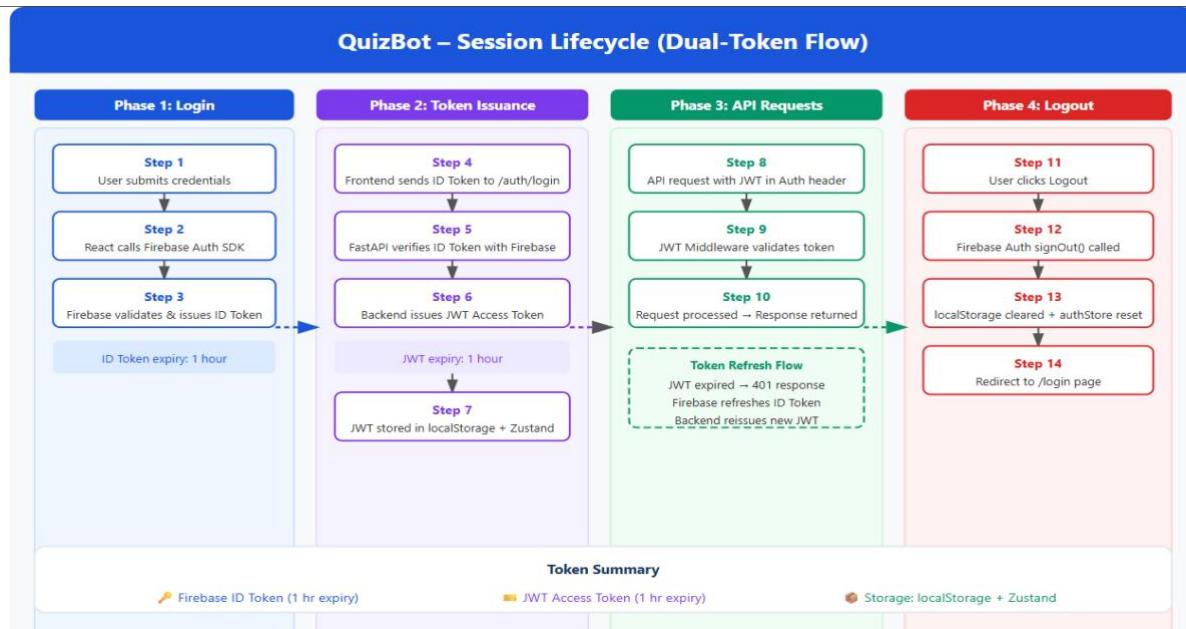
5.2 Session Management

QuizBot uses a **dual-token session model** — Firebase ID Token for authentication and a backend JWT for API authorization.

Session Security Rules

- JWT tokens expire after **1 hour** — matching Firebase ID token lifetime
- Tokens are **never stored in cookies** — localStorage only (no CSRF risk)
- Backend verifies **every request** — no session state held server-side (stateless)
- Frontend **Axios interceptor** automatically attaches the latest valid token to every outgoing request
- If token refresh fails — user is silently logged out and redirected to /login

Session Lifecycle



6. Caching

QuizBot is a **v1.0 application** with no dedicated caching layer implemented yet. However, the following caching mechanisms exist implicitly and are documented for future planning.

6.1 Current Implicit Caching

Cache Type	Location	What Is Cached	Duration
JWT Token	localStorage	Auth token	1 hour (token lifetime)
Quiz State	Zustand (memory)	Current quiz + answers	Session duration
Auth User	Zustand (memory)	User profile + UID	Session duration
Vite Build Cache	Browser cache	Static JS/CSS bundles	Until new deploy
Firebase SDK Cache	Browser memory	Firebase app instance	Session duration
Singleton Services	Backend memory	llm_service, vector_db	App lifetime

6.2 Backend Singleton Caching

FastAPI initialises the following services **once at startup** and reuses them across all requests - acting as in-memory singletons:

Singleton	Initialized In	What It Holds
llm_service	main.py startup	Gemini client + config
vector_db	main.py startup	ChromaDB client + collection reference
auth_service	main.py startup	Firebase Admin app instance
SessionLocal	database.py	SQLAlchemy connection pool

7. Non-Functional Requirements

7.1 Security Aspects

7.1.1 Authentication Security

Requirement	Implementation	Status
Secure login	Firebase Authentication (email + password)	Implemented
Token-based auth	JWT via python-jose (1hr expiry)	Implemented
Server-side verification	Firebase Admin SDK token validation on every request	Implemented
Token refresh	Frontend auto-refresh via getIdToken(true)	Implemented
Logout cleanup	localStorage cleared + Zustand reset	Implemented
Password hashing	Managed by Firebase Auth (bcrypt internally)	Implemented

7.1.2 API Security

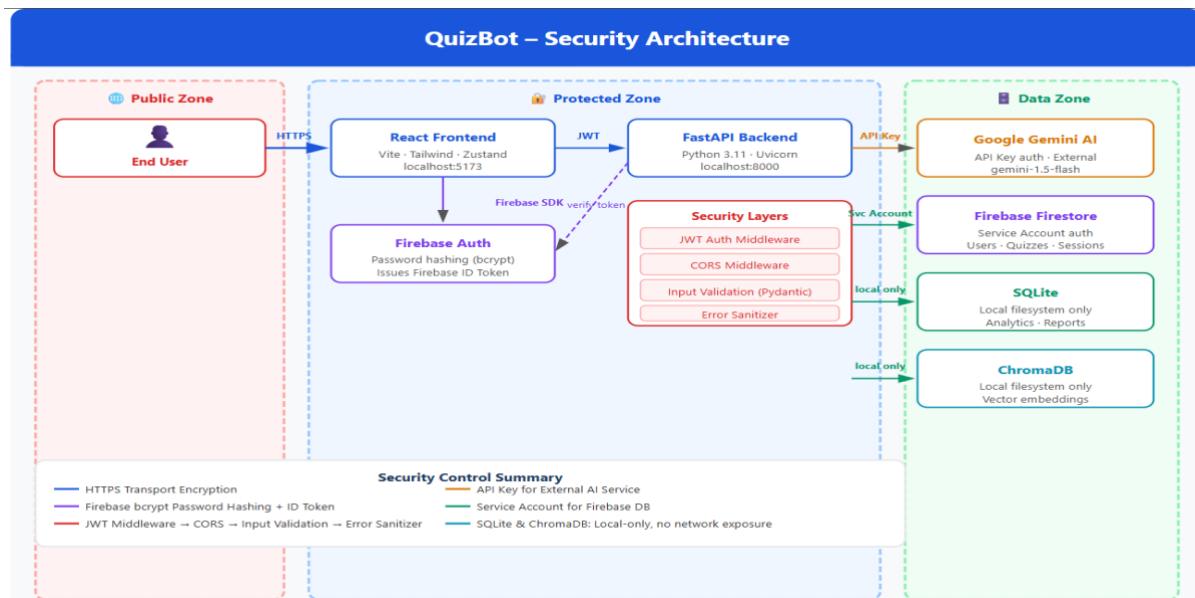
Requirement	Implementation	Status
CORS policy	Backend allows only http://localhost:5173	Implemented
Protected routes	JWT middleware on all non-public endpoints	Implemented
Input validation	Pydantic V2 schema validation on all request bodies	Implemented
Rate limiting	Not yet implemented	Planned
HTTPS enforcement	Required in production deployment	Production

7.1.3 Data Security

Requirement	Implementation
API key protection	All keys stored in .env files, never hardcoded

Firebase credentials	Service account JSON stored server-side only
Email content privacy	Donor emails retained for 30 days then purged
No PII in vectors	ChromaDB stores only non-reversible embedding vectors
DB access control	SQLite and ChromaDB are local — no external exposure
Firebase DB rules	Configured to deny all direct client access (server-only)

7.1.4 Security Architecture Diagram



7.2 Performance Aspects

