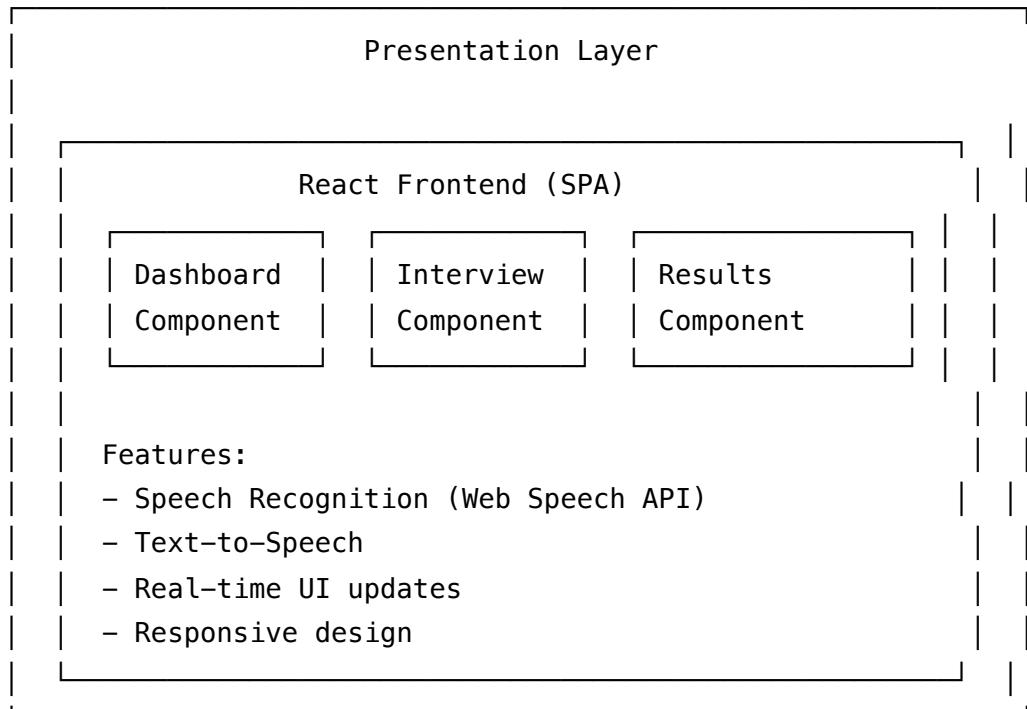


# **System Architecture**

## **Overview**

The AI Mock Interview Platform follows a modern three-tier architecture with clear separation of concerns between presentation, business logic, and data layers.

# Architecture Diagram



HTTPS/REST API  
JSON Payloads  
JWT Authentication

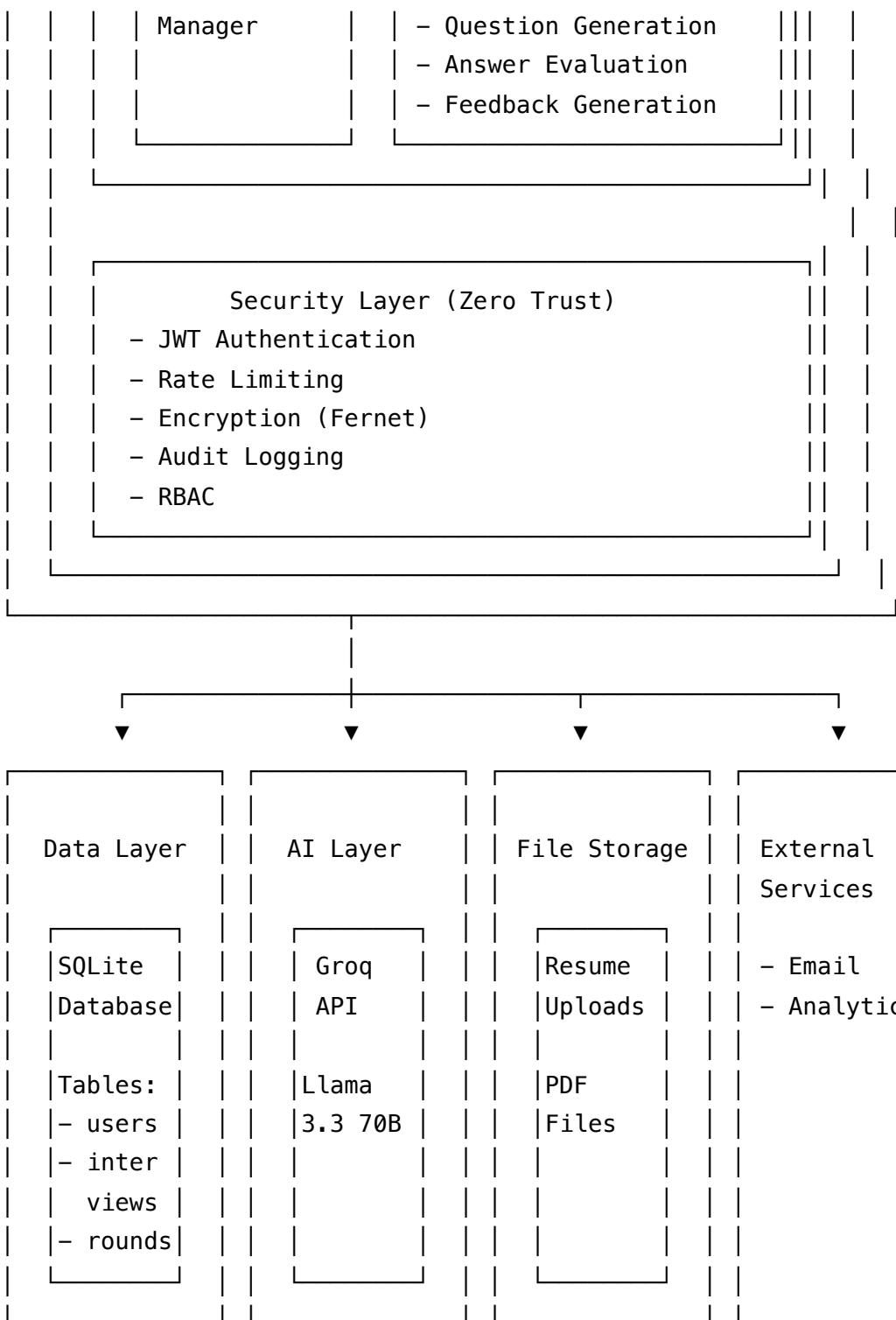
## Application Layer

### Flask Backend Server

- API Endpoints (REST)
- Authentication (/api/login, /api/register)
  - Interviews (/api/start-role-interview)
  - Multi-Round (/api/suggest-rounds)
  - Feedback (/api/personalized-feedback)

### Business Logic Layer

Interview      Evaluation Engine



## Component Details

### 1. Frontend Layer (React)

**Technology:** React 18, React Router, Web Speech API

## **Key Components:**

- **Dashboard:** Main interface for interview management
- **Login/Register:** Authentication flows
- **RoleSelection:** Role-based interview configuration
- **Interview:** Real-time interview interface
- **Results:** Score display and feedback visualization

## **State Management:**

- React Hooks (useState, useEffect, useRef)
- Local state for UI components
- Session storage for authentication tokens

## **Communication:**

- REST API calls via fetch()
- JSON request/response format
- JWT token in Authorization header

## **2. Backend Layer (Flask)**

**Technology:** Flask 2.3, Python 3.8+

### **Modules:**

#### **API Layer**

- RESTful endpoints
- Request validation
- Response formatting
- Error handling

#### **Business Logic**

- **Interview Manager:** Orchestrates interview flow
- **Evaluation Engine:** AI-powered scoring
- **Question Generator:** LLM-based question creation
- **Feedback Generator:** Personalized learning paths

## Security Layer

- **Authentication:** JWT with refresh tokens
- **Authorization:** Role-based access control
- **Rate Limiting:** Prevents abuse
- **Encryption:** Fernet for sensitive data
- **Audit Logging:** Complete activity tracking

## 3. Data Layer

**Database:** SQLite with encryption

**Schema:**

```
users (id, email, password_hash, name, role, created_at)
interviews (id, user_id, job_role, score, status, created_at)
interview_questions (id, interview_id, round_id, question, answer, score, ...)
interview_rounds (id, interview_id, round_name, round_type, status, score, ...)
learning_paths (id, interview_id, strengths, weaknesses, roadmap, resources)
audit_logs (id, user_id, action, resource, timestamp, success)
```

## 4. AI Layer

**Provider:** Groq Cloud API

**Model:** Llama 3.3 70B Versatile

**Use Cases:**

1. **Question Generation:** Role-specific interview questions
2. **Answer Evaluation:** Multi-dimensional scoring
3. **Follow-up Generation:** Contextual probing questions
4. **Feedback Creation:** Personalized improvement plans
5. **Round Suggestion:** Interview round recommendations

**Prompt Engineering:**

- Fairness-aware prompts
- Role-specific templates
- Structured JSON outputs
- Temperature tuning (0.7-0.8)

# Data Flow

## Interview Flow

### 1. User Login

Frontend → POST /api/login → Backend  
Backend → Validate credentials → Generate JWT  
Backend → Response with token → Frontend stores token

### 2. Start Interview

Frontend → POST /api/start-role-interview → Backend  
Backend → Generate questions via LLM → Store in DB  
Backend → Response with questions → Frontend displays

### 3. Submit Answer

Frontend → POST /api/submit-answer-enhanced → Backend  
Backend → Evaluate via LLM → Calculate scores  
Backend → Generate follow-up (if needed) → Store results  
Backend → Response with scores → Frontend updates UI

### 4. Complete Interview

Frontend → POST /api/complete-interview → Backend  
Backend → Calculate final scores → Generate feedback via LLM  
Backend → Store learning path → Send email notification  
Backend → Response with results → Frontend shows feedback

### 5. View Results

Frontend → GET /api/personalized-feedback/<id> → Backend  
Backend → Retrieve from DB → Response with feedback  
Frontend → Display strengths, weaknesses, roadmap, resources

# Multi-Round Flow

## 1. Suggest Rounds

Frontend → POST /api/suggest-rounds → Backend

Backend → LLM analyzes role → Suggests rounds

Backend → Response with suggestions → Frontend displays cards

## 2. Start Multi-Round

Frontend → POST /api/start-multi-round-interview → Backend

Backend → Create interview + rounds → Store in DB

Backend → Response with round IDs → Frontend starts first round

## 3. Start Round

Frontend → POST /api/start-round/<id> → Backend

Backend → Generate round-specific questions → Store

Backend → Response with questions → Frontend displays

## 4. Complete Round

Frontend → POST /api/complete-round/<id> → Backend

Backend → Calculate round score → Check for next round

Backend → Response with score + next round → Frontend advances

## 5. All Rounds Complete

Backend → Generate comprehensive feedback → Store

Frontend → Display overall results with round breakdown

# Security Architecture

## Zero Trust Principles

### 1. Never Trust, Always Verify

- Every request authenticated
- JWT validation on all endpoints
- Token expiration (15 min access, 7 day refresh)

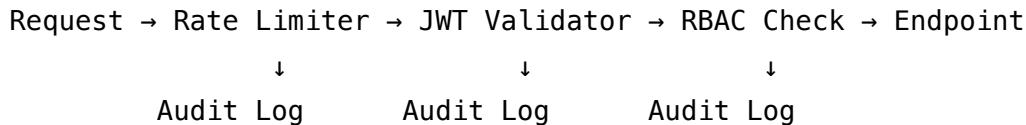
### 2. Least Privilege Access

- Role-based permissions
- Resource-level authorization
- Minimal data exposure

### 3. Assume Breach

- Encrypted data at rest (Fernet)
- Audit logging for forensics
- Rate limiting for DoS protection

## Security Layers



## Scalability Considerations

### Current Architecture

- Single server deployment
- SQLite database
- Synchronous processing

### Scaling Strategy

#### Horizontal Scaling:

- Stateless API design (ready for load balancing)
- JWT tokens (no server-side sessions)
- Database connection pooling

#### Vertical Scaling:

- Async LLM calls (non-blocking)
- Caching layer (Redis) for common questions
- CDN for static assets

#### Database Migration:

- SQLite → PostgreSQL for production
- Read replicas for analytics
- Sharding by user\_id

# Deployment Architecture

## Development

```
Local Machine
├── Backend (localhost:5000)
├── Frontend (localhost:3000)
└── Database (local SQLite file)
```

## Production (Recommended)

```
Cloud Platform (AWS/GCP/Azure)
├── Frontend (Vercel/Netlify)
│   └── CDN for static assets
├── Backend (EC2/Cloud Run/App Service)
│   ├── Load Balancer
│   ├── Auto-scaling group
│   └── Health checks
├── Database (RDS/Cloud SQL)
│   ├── Primary instance
│   └── Read replicas
└── File Storage (S3/Cloud Storage)
    └── Resume uploads
```

# Monitoring & Observability

## Metrics

- Request latency
- Error rates
- LLM API response times
- Database query performance

## Logging

- Application logs (INFO, ERROR)
- Audit logs (security events)
- Access logs (API requests)

## Alerts

- High error rates
- Slow response times
- Security violations
- API quota limits

## Technology Choices Rationale

### Flask vs Django

**Choice:** Flask

**Reason:** Lightweight, flexible, faster development for MVP

### SQLite vs PostgreSQL

**Choice:** SQLite (dev), PostgreSQL (prod)

**Reason:** Zero-config for development, easy migration path

### Groq vs OpenAI

**Choice:** Groq

**Reason:** 10x faster inference, cost-effective, high quality

### React vs Vue

**Choice:** React

**Reason:** Larger ecosystem, better documentation, team familiarity

### JWT vs Sessions

**Choice:** JWT

**Reason:** Stateless, scalable, mobile-friendly