That's a comprehensive healthcare data analysis project! Let me break down the key components and considerations for your speech-to-query assistant:

**Core Architecture Components**

**Authentication & Authorization**

* User authentication system (likely SSO integration for healthcare environments)
* Role-based access control (RBAC) to validate dataset permissions
* Integration with Databricks workspace permissions

**Dataset Selection Interface**

* Dynamic catalog of available datasets from Databricks environment
* Metadata display (schema, descriptions, data lineage)
* Permission validation before dataset access

**Speech-to-Text Processing**

* Audio input handling (microphone capture, file upload)
* Speech recognition service (Azure Speech, AWS Transcribe, or Google Speech-to-Text)
* Fallback text input option

**Natural Language to Query Translation**

* NLP processing to understand user intent
* Query generation engine (SQL/PySpark)
* Context awareness for healthcare-specific terminology

**Critical Healthcare Compliance Features**

**PHI Data Protection**

* Automatic PHI detection and masking in query results
* Dynamic data masking based on user permissions
* Encryption of data in transit and at rest

**Audit Logging**

* Comprehensive logging of all user actions
* Query execution tracking with timestamps
* Data access audit trails for compliance (HIPAA, etc.)
* User session monitoring

**Technical Implementation Considerations**

**Databricks Integration**

* Databricks SQL Connector for query execution
* Unity Catalog integration for data governance
* Cluster management for query processing

**Query Generation**

* Template-based query building
* Parameterized queries to prevent injection
* Query validation and optimization
* Support for both batch and streaming data

**Security Measures**

* Row-level and column-level security
* Data lineage tracking
* Automated PHI scanning
* Secure API endpoints

Would you like me to dive deeper into any specific component, such as the NLP-to-SQL translation logic, the PHI masking implementation, or the audit logging architecture? I can also help you design the system architecture or create implementation examples for specific parts.

Excellent tech stack choice for a healthcare speech-to-query system! That combination gives you a robust, scalable architecture. Let me outline how these technologies would work together:

**Architecture Overview**

**Frontend: React**

* User authentication UI
* Dataset selection interface
* Audio recording/upload components
* Query results visualization
* Real-time feedback and loading states

**Backend: FastAPI**

* RESTful API endpoints
* Authentication middleware
* Databricks integration
* Query execution management
* Audit logging service

**AI/ML: NLP + Python**

* Speech-to-text processing
* Natural language understanding
* SQL/PySpark query generation
* PHI detection and masking

**Cloud: Azure Services**

* Azure Speech Services for audio processing
* Azure OpenAI/Cognitive Services for NLP
* Azure Key Vault for secrets management
* Azure Monitor for logging and analytics

**Key Implementation Components**

**React Frontend Structure:**

src/

├── components/

│ ├── Auth/LoginComponent

│ ├── DatasetSelector/

│ ├── AudioRecorder/

│ ├── QueryBuilder/

│ └── ResultsViewer/

├── services/

│ ├── authService.js

│ ├── databricksService.js

│ └── audioService.js

└── hooks/

├── useAuth.js

├── useAudioRecording.js

└── useQueryExecution.js

**FastAPI Backend Structure:**

app/

├── routers/

│ ├── auth.py

│ ├── datasets.py

│ ├── queries.py

│ └── audit.py

├── services/

│ ├── databricks\_service.py

│ ├── nlp\_service.py

│ ├── phi\_masking.py

│ └── speech\_service.py

├── models/

└── middleware/

├── auth\_middleware.py

└── audit\_middleware.py

**Integration Points**

**Azure Speech Services:**

* Real-time speech-to-text conversion
* Custom healthcare vocabulary models
* Multiple language support

**Databricks Connectivity:**

* Databricks SQL Connector
* Unity Catalog API integration
* Cluster management via REST API

**NLP Pipeline:**

* Intent classification (aggregate, filter, join, etc.)
* Entity extraction (columns, tables, conditions)
* Query template matching
* Healthcare terminology processing

**Security & Compliance Implementation**

**Authentication Flow:**

* JWT tokens with Azure AD integration
* Role-based permissions validation
* Session management with refresh tokens

**PHI Protection:**

* Pre-query PHI scanning
* Result set masking based on user roles
* Regex + ML-based PHI detection

**Audit Logging:**

* Structured logging with correlation IDs
* Azure Monitor integration
* Compliance reporting dashboards

Would you like me to create a detailed implementation plan for any specific component, or would you prefer to see code examples for the core integrations (like the FastAPI-Databricks connection or the React audio recording component)?

Perfect! Let's start with Step 1 - Environment Setup without Docker.

**Step 1: Environment Setup**

**1.1 Prerequisites Check**

First, let's make sure you have the required software:

# Check Python version (need 3.9+)

python --version

# or

python3 --version

# Check Node.js version (need 18+)

node --version

# Check npm

npm --version

If you don't have these installed:

* **Python**: Download from [python.org](https://python.org/) (3.9 or higher)
* **Node.js**: Download from [nodejs.org](https://nodejs.org/) (18 or higher)

**1.2 Create Project Directory Structure**

# Create main project directory

mkdir healthcare-speech-assistant

cd healthcare-speech-assistant

# Create backend and frontend directories

mkdir backend

mkdir frontend

**1.3 Backend Setup (FastAPI)**

# Navigate to backend directory

cd backend

# Create Python virtual environment

python -m venv venv

# Activate virtual environment

# On Windows:

venv\Scripts\activate

# On macOS/Linux:

source venv/bin/activate

# Install core dependencies

pip install fastapi uvicorn python-multipart python-jose[cryptography] passlib[bcrypt] python-dotenv requests

# Create requirements.txt

pip freeze > requirements.txt

**1.4 Frontend Setup (React with TypeScript)**

# Navigate back to main directory

cd ..

cd frontend

# Create React app with TypeScript

npx create-react-app . --template typescript

# Install additional dependencies we'll need

npm install axios react-router-dom @types/react-router-dom

npm install @types/node

# Install UI components (optional but helpful)

npm install @mui/material @emotion/react @emotion/styled

**1.5 Create Basic Project Structure**

Let's create the initial file structure:

**Backend Structure:**

cd ../backend

# Create app directory structure

mkdir app

mkdir app/routers

mkdir app/services

mkdir app/models

mkdir app/middleware

mkdir app/utils

# Create main files

touch main.py

touch app/\_\_init\_\_.py

touch app/routers/\_\_init\_\_.py

touch app/services/\_\_init\_\_.py

touch app/models/\_\_init\_\_.py

touch app/middleware/\_\_init\_\_.py

touch app/utils/\_\_init\_\_.py

touch .env

**1.6 Create Basic FastAPI Application**

Let's create a simple FastAPI app to test everything works:### 1.7 Create Environment File### 1.8 Create Basic React App Structure### 1.9 Update React App Styling### 1.10 Test the Setup

Now let's test everything works:

**Terminal 1 (Backend):**

cd backend

# Make sure virtual environment is activated

source venv/bin/activate # macOS/Linux

# or

venv\Scripts\activate # Windows

# Start FastAPI server

uvicorn main:app --reload --port 8000

**Terminal 2 (Frontend):**

cd frontend

# Start React development server

npm start

**1.11 Verify Everything Works**

1. **Backend**: Visit http://localhost:8000 - you should see the API message
2. **API Docs**: Visit http://localhost:8000/docs - FastAPI auto-generated docs
3. **Frontend**: Visit http://localhost:3000 - React app should show "Connected" status

**1.12 Create .gitignore Files## ✅ Step 1 Complete!**

You now have:

* ✅ Python virtual environment with FastAPI
* ✅ React TypeScript application
* ✅ Basic API endpoints working
* ✅ CORS configured for frontend-backend communication
* ✅ Environment variables setup
* ✅ Project structure ready for expansion

**Next Steps:**

* Test both servers are running
* Verify the React app shows "✅ Connected" status
* Check FastAPI docs at http://localhost:8000/docs

**Ready for Step 2?** We'll set up the authentication system with JWT tokens and user management. Let me know when you've got everything running and we'll move forward!

from pydantic import BaseModel, EmailStr

from typing import Optional, List

from datetime import datetime

from enum import Enum

class UserRole(str, Enum):

ADMIN = "admin"

DATA\_ANALYST = "data\_analyst"

HEALTHCARE\_PROFESSIONAL = "healthcare\_professional"

RESEARCHER = "researcher"

VIEWER = "viewer"

class UserBase(BaseModel):

email: EmailStr

full\_name: str

role: UserRole

department: Optional[str] = None

is\_active: bool = True

class UserCreate(UserBase):

password: str

class UserLogin(BaseModel):

email: EmailStr

password: str

class UserResponse(UserBase):

id: int

created\_at: datetime

last\_login: Optional[datetime] = None

class Config:

from\_attributes = True

class Token(BaseModel):

access\_token: str

token\_type: str

expires\_in: int

user: UserResponse

class TokenData(BaseModel):

email: Optional[str] = None

user\_id: Optional[int] = None

role: Optional[UserRole] = None

# Dataset permissions model

class DatasetPermission(BaseModel):

dataset\_id: str

dataset\_name: str

permission\_level: str # "read", "write", "admin"

granted\_by: str

granted\_at: datetime