

PM Portal - Complete Technical & Functional Documentation

Version: 1.0.0

Last Updated: 2025

Project: PM Portal Bot (Sharuk_Proj)

Table of Contents

1. [Overview](#1-overview)
2. [System Architecture](#2-system-architecture)
3. [Technology Stack](#3-technology-stack)
4. [Database Schema](#4-database-schema)
5. [API Endpoints](#5-api-endpoints)
6. [Authentication & Authorization](#6-authentication--authorization)
7. [Core Features](#7-core-features)
8. [Deployment](#8-deployment)
9. [Troubleshooting](#9-troubleshooting)

1. Overview

1.1 What is PM Portal?

PM Portal is a full-stack web application designed to assist project managers with AI-powered chatbot functionality. The system provides intelligent document processing, project management capabilities, and conversational AI to help users manage projects, answer questions about documents, and organize work efficiently.

1.2 Problem It Solves

Before PM Portal:

- Project managers had to manually search through multiple documents to find information

- No centralized system to manage project conversations and files
- Difficult to get quick answers about project documentation
- No way to organize project-related chats and files together

****With PM Portal:****

- AI-powered chatbot that understands your documents and answers questions instantly
- Centralized project management with conversations and files organized by project
- Intelligent document search using vector embeddings (Pinecone)
- Easy file upload and processing (PDF, DOCX, XLSX)
- Mandatory template files that can be used across projects
- Real-time chat history and session management

1.3 Project Flow in Simple Words

****Step 1: User Login****

- User visits the landing page
- Chooses to login with Google OAuth or Email
- System creates/updates user account in database
- User session is created and stored

****Step 2: Main Interface (Chatbot Landing Page)****

- User sees the chatbot interface immediately after login
- Chatbot is the main landing page (no separate home page)
- User can start chatting, upload files, or create projects

****Step 3: Using the Chatbot****

- User types a question in the chat
- System searches through uploaded files and mandatory files using vector search (Pinecone)
- Relevant document chunks are retrieved
- AI (Google Gemini) generates an answer based on the context
- Answer is displayed to the user
- Conversation is saved to database

****Step 4: File Management****

- User can upload files (PDF, DOCX, XLSX)
- Files are processed: text is extracted
- Text is split into chunks (400 characters each)
- Each chunk is converted to an embedding (vector)
- Embeddings are stored in Pinecone vector database
- File metadata is saved in PostgreSQL database

****Step 5: Project Management****

- User can create projects to organize work
- Each project can have multiple conversations
- Files can be linked to projects
- Mandatory template files can be marked for project use
- All project data is stored and retrieved from database

1.4 High-Level Summary

Frontend (React.js)

- **Location:** `frontend/src/`
- **Main Component:** `HomePage.js` - The chatbot interface
- **Purpose:** User interface for interacting with the chatbot, managing projects, and uploading files
- **Key Features:**
 - Real-time chat interface
 - File upload with drag-and-drop
 - Project creation and management
 - Chat history sidebar
 - Mandatory files management

Backend (FastAPI)

- **Location:** `backend/main.py`
- **Purpose:** API server that handles all business logic
- **Key Responsibilities:**
 - Authentication (Google OAuth, Email login)
 - Chat message processing
 - File upload and processing
 - Vector search (Pinecone integration)
 - AI integration (Google Gemini)
 - Database operations

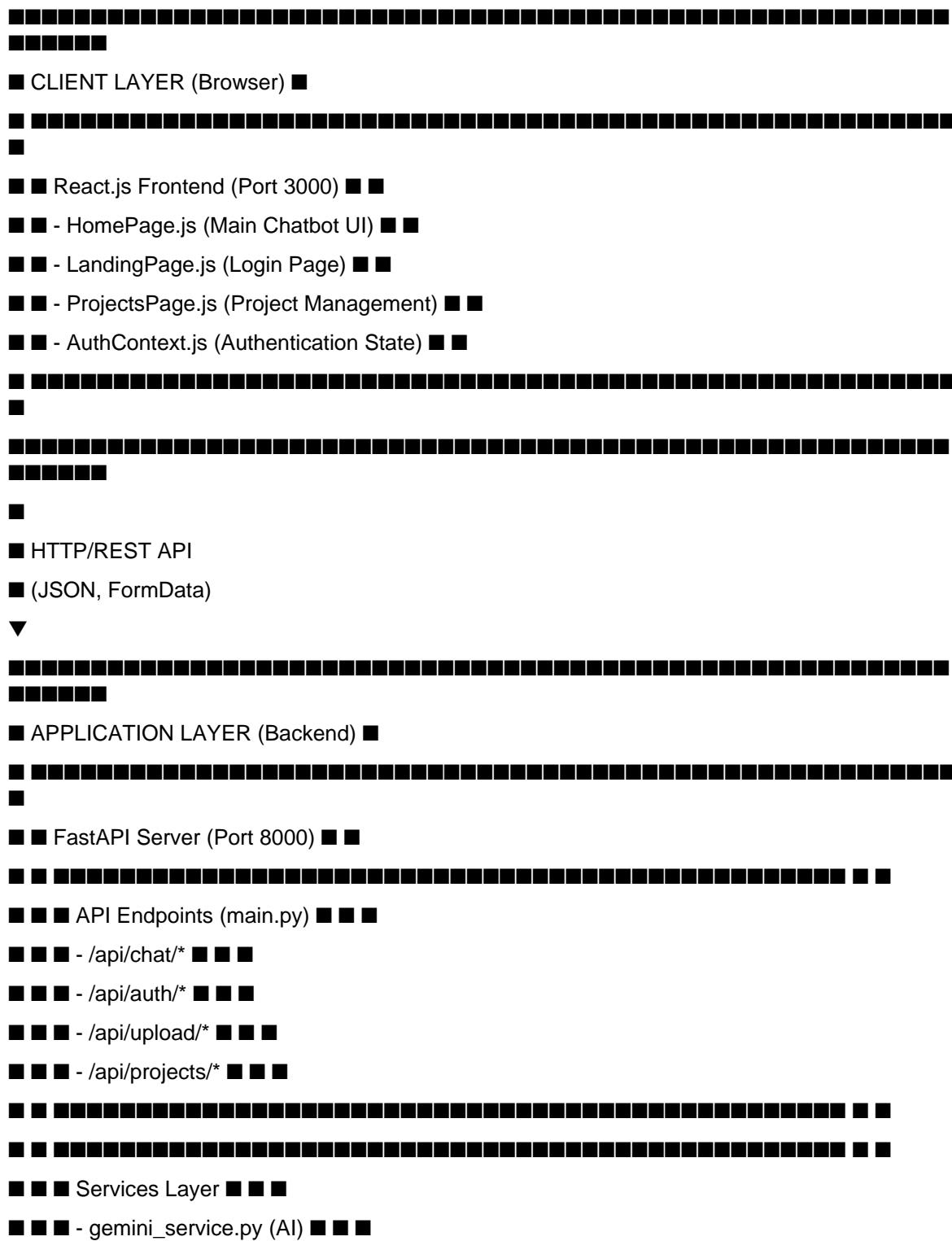
Database (PostgreSQL)

- **Purpose:** Stores all application data
- **Key Tables:**
 - Users and sessions
 - Projects and conversations
 - Chat messages
 - Uploaded files
 - Mandatory files
 - Project knowledge base files

2. System Architecture

2.1 Full Architecture Explanation

PM Portal follows a **three-tier architecture**:





2.2 Frontend Structure

Component Hierarchy

App.js (Root)

■■■ AuthProvider (Context)

■ ■ ■ Routes

■ ■■■ LandingPage (Login)

■ ■ ■ ■ GoogleCallback (OAuth Handler)

- ■■■ EmailLoginCallback (Email Login Handler)
- ■■■ HomePage (Main Chatbot - Landing Page)
- ■■■ ProjectsPage (Project Management)

Key Frontend Components

1. HomePage.js ('frontend/src/components/HomePage.js`)

- **Purpose:** Main chatbot interface (the landing page)
- **Key Features:**
 - Chat input and message display
 - File upload functionality
 - Project creation and management
 - Chat history sidebar
 - Mandatory files dropdown
- **State Management:**
 - `chatMessages`: Array of chat messages
 - `chatId`: Current chat session ID
 - `projects`: List of user projects
 - `uploadedFileIds`: Files attached to current chat
 - `playbookFileIds`: Mandatory files selected for project

2. LandingPage.js ('frontend/src/components/LandingPage.js`)

- **Purpose:** User login page
- **Features:**
 - Google OAuth login button
 - Email login option
- Redirects to `/home` after successful login

3. AuthContext.js ('frontend/src/context/AuthContext.js`)

- **Purpose:** Manages authentication state across the app
- **Provides:**
 - `user`: Current user object
 - `login()`: Login function
 - `logout()`: Logout function
 - `isAuthenticated`: Boolean flag

2.3 Backend Structure

File Organization

backend/

■■■ main.py # FastAPI app + all API endpoints

```
■■■■■ models.py # SQLAlchemy database models  
■■■■■ schemas.py # Pydantic request/response schemas  
■■■■■ db_migrations.py # Database migration utilities  
■■■■■ requirements.txt # Python dependencies  
■■■■■ services/ # Business logic services  
■ ■■■■ auth_service.py  
■ ■■■■ gemini_service.py  
■ ■■■■ pinecone_service.py  
■ ■■■■ embedding_service.py  
■ ■■■■ chunking_service.py  
■ ■■■■ pdf_service.py  
■ ■■■■ docx_extraction_helper.py  
■■■■■ uploads/ # User-uploaded files storage
```

Request Flow Example: User Asks a Question

1. User types question in frontend

↓

2. Frontend sends POST /api/chat

{

"user_message": "What is the project timeline?",

"chat_id": "abc-123",

"user_email": "user@example.com",

"uploaded_file_ids": [1, 2],

"playbook_file_ids": [5, 6]

}

↓

3. Backend receives request in main.py

↓

4. Backend processes:

- Gets uploaded files from database
- Gets mandatory files from database
- Searches Pinecone for relevant chunks
- Calls Gemini API with context
- Saves message to database

↓

5. Backend returns response

{

"response": "Based on the documents...",

"success": true

}

↓

6. Frontend displays response in chat

2.4 File Processing Flow

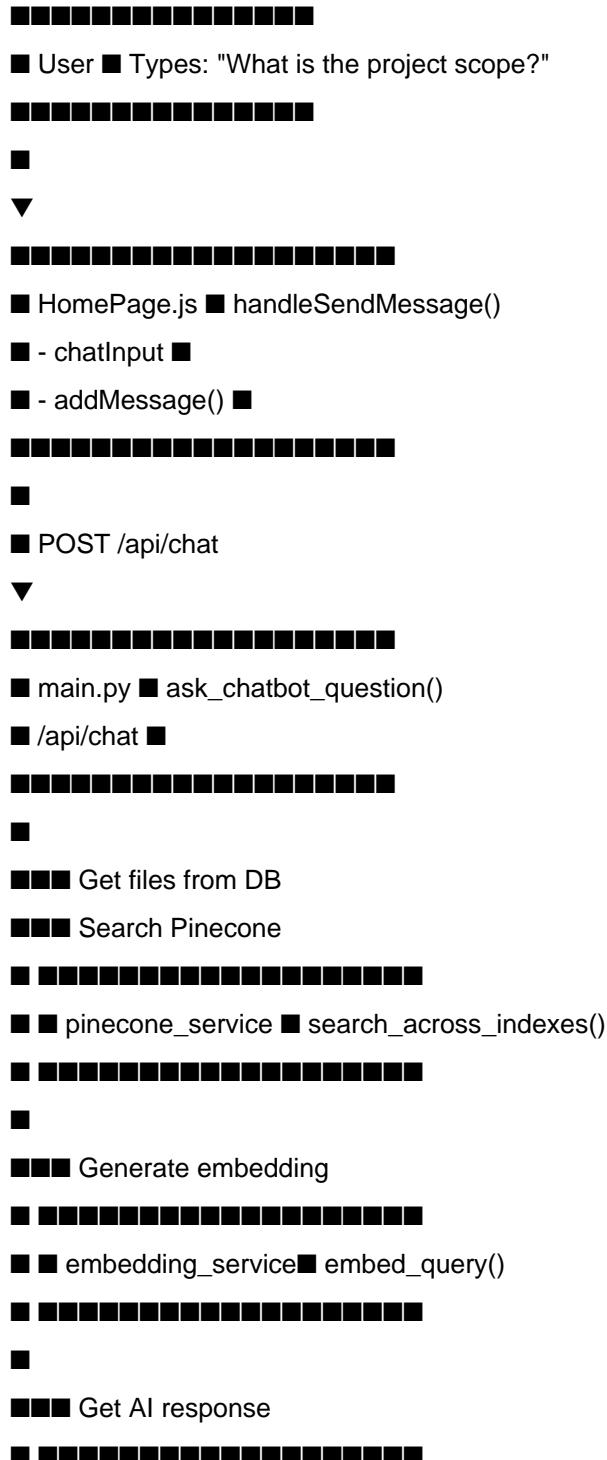
When a user uploads a file:

1. User selects file in frontend
↓
2. Frontend uploads to POST /api/upload-file
(FormData with file + user_email)
↓
3. Backend receives file
↓
4. Backend processes file:
 - a. Saves file to uploads/ directory
 - b. Extracts text:
 - PDF → pdf_service.py → pdfplumber
 - DOCX → docx_extraction_helper.py → python-docx
 - XLSX → xlsx library
 - c. Saves extracted text to database (UploadedFile table)
↓
5. Backend creates chunks:
 - Uses chunking_service.py
 - Splits text into 400-character chunks
 - 100-character overlap between chunks
↓
6. Backend generates embeddings:
 - Uses embedding_service.py
 - Converts each chunk to 384-dimensional vector
 - Uses "all-MiniLM-L6-v2" model
↓
7. Backend indexes in Pinecone:
 - Creates index: kb-file-{file_id}-{filename}
 - Stores vectors with metadata
 - Each vector has: file_id, file_name, chunk_index, text
↓
8. Backend updates database:
 - Sets indexing_status = "indexed"
↓
9. Backend returns success to frontend

```
{  
  "success": true,  
  "file_id": 123,  
  "message": "File uploaded and indexed"  
}
```

2.5 Component Interaction

****Chatbot Question Flow:****



```
■ ■ gemini_service ■ chat()
■ ██████████████████
■
■■■ Save to database
███████████████████
■ Conversation ■ Save message
■ ChatMessage ■
███████████████████
■
▼
███████████████████
■ Response JSON ■ { "response": "...", "success": true }
███████████████████
■
▼
███████████████████
■ HomePage.js ■ Updates chatMessages state
■ - Display ■ Shows response in UI
███████████████████
---
```

3. Technology Stack

3.1 Frontend Technologies

React.js 18.2.0

- **Why:** Modern, component-based UI library
- **Usage:** Building interactive user interface
- **Key Features Used:**
 - Hooks (useState, useEffect, useCallback, useMemo)
 - Context API (AuthContext)
 - Component lifecycle management

React Router DOM 6.20.1

- **Why:** Client-side routing for single-page application
- **Usage:** Navigation between pages (Landing, Home, Projects)
- **Routes:**
 - `'/` → LandingPage

- `/home` → HomePage (Chatbot)
- `/projects` → ProjectsPage
- `/callback` → GoogleCallback
- `/email-login` → EmailLoginCallback

Axios 1.6.2

- **Why:** HTTP client for API calls
- **Usage:** Sending requests to backend API
- **Example:**
```javascript  
axios.post(`/\${API\_URL}/api/chat`, formData)  
```

XLSX 0.18.5

- **Why:** Excel file processing in browser
- **Usage:** Reading Excel files for file upload feature

3.2 Backend Technologies

FastAPI 0.104.1

- **Why:** Modern, fast Python web framework
- **Usage:** REST API server
- **Features:**
 - Automatic API documentation (Swagger UI)
 - Type validation with Pydantic
 - Async support for better performance

Uvicorn 0.24.0

- **Why:** ASGI server to run FastAPI
- **Usage:** Production server
- **Command:** `uvicorn main:app --host 0.0.0.0 --port 8000`

SQLAlchemy 2.0.23

- **Why:** Python ORM (Object-Relational Mapping)
- **Usage:** Database operations without writing SQL
- **Benefits:**
 - Type-safe database queries
 - Automatic relationship management
 - Database-agnostic (works with PostgreSQL, MySQL, etc.)

PostgreSQL (via psycopg2-binary 2.9.9)

- **Why:** Robust, open-source relational database
- **Usage:** Storing all application data
- **Why PostgreSQL:**

- ACID compliance
- JSON support
- Excellent performance
- Free and open-source

3.3 AI & Machine Learning

Google Gemini API (gemini-2.0-flash)

- **Why:** Advanced AI model for generating responses
- **Usage:** Chatbot responses based on document context
- **Features:**
 - Multi-key fallback support (3 API keys)
 - Automatic rate limit handling
 - Context-aware responses

Pinecone (Vector Database)

- **Why:** Fast similarity search for document chunks
- **Usage:** Finding relevant document sections for questions
- **How it works:**
 1. Documents are split into chunks
 2. Each chunk is converted to a vector (embedding)
 3. Vectors are stored in Pinecone
 4. When user asks a question, question is also converted to vector
 5. Pinecone finds most similar document chunks
 6. These chunks are sent to Gemini as context

Sentence Transformers (all-MiniLM-L6-v2)

- **Why:** Converts text to numerical vectors (embeddings)
- **Usage:** Creating embeddings for document chunks and queries
- **Output:** 384-dimensional vectors
- **Why this model:**
 - Fast and efficient
 - Good quality embeddings
 - Small model size

3.4 File Processing Libraries

pdfplumber 0.10.3

- **Why:** Extract text from PDF files
- **Usage:** Processing uploaded PDF documents

python-docx
- **Why:** Extract text from DOCX files
- **Usage:** Processing uploaded Word documents

PyPDF2 3.0.1
- **Why:** Additional PDF manipulation
- **Usage:** PDF metadata extraction

3.5 Authentication

Google OAuth 2.0
- **Why:** Secure, industry-standard authentication
- **Usage:** User login with Google account
- **Flow:**
1. User clicks "Login with Google"
2. Redirected to Google login page
3. User authorizes application
4. Google redirects back with authorization code
5. Backend exchanges code for access token
6. Backend gets user info from Google
7. User is logged in

Email Login (Custom)
- **Why:** Alternative login method for internal users
- **Usage:** Direct email-based authentication
- **Restriction:** Only @forsysinc.com emails allowed

3.6 How Technologies Work Together

Example: User asks "What is the project timeline?"

1. React Frontend (HomePage.js)
 - User types question
 - Calls addMessageWithBotResponse()
 - Sends POST request to backend
2. FastAPI Backend (main.py)
 - Receives request at /api/chat
 - Extracts question and file IDs

3. Database (PostgreSQL)

- Backend queries UploadedFile table
- Gets file metadata and extracted text

4. Pinecone Service

- Backend calls embedding_service.embed_query(question)
- Converts question to 384-dim vector
- Searches Pinecone for similar chunks
- Returns top 5 most relevant chunks

5. Gemini Service

- Backend calls gemini_service.chat()
- Sends question + context chunks to Gemini
- Gemini generates answer
- Returns response to backend

6. Database (PostgreSQL)

- Backend saves question and answer to ChatMessage table
- Updates Conversation table

7. FastAPI Backend

- Returns JSON response to frontend

8. React Frontend

- Receives response
- Updates chatMessages state
- Displays answer in UI

3.7 Development Tools

Node.js 16+

- **Why:** JavaScript runtime for frontend development
- **Usage:** Running React development server
- **Command:** `npm start` (runs on port 3000)

Python 3.8+

- **Why:** Backend programming language
- **Usage:** FastAPI server and all backend logic

Git

- **Why:** Version control
- **Usage:** Tracking code changes

3.8 Environment Variables

Backend (.env):

```
DATABASE_URL=postgresql://user:pass@localhost:5432/dbname
SECRET_KEY=your-secret-key
CORS_ORIGINS=http://localhost:3000
GEMINI_API_KEY_1=your-key-1
GEMINI_API_KEY_2=your-key-2
GEMINI_API_KEY_3=your-key-3
PINECONE_API_KEY=your-pinecone-key
GOOGLE_CLIENT_ID=your-google-client-id
GOOGLE_CLIENT_SECRET=your-google-client-secret
GOOGLE_REDIRECT_URI=http://localhost:3000/callback
```

Frontend (.env):

```
REACT_APP_API_URL=http://localhost:8000
```

4. Database Schema

4.1 Overview

The PM Portal uses PostgreSQL as its relational database. All data is stored in tables defined using SQLAlchemy ORM models. The database stores user information, authentication sessions, projects, conversations, chat messages, uploaded files, and mandatory template files.

4.2 Database Tables

4.2.1 Users Table (`users`)

Purpose: Stores user account information

Fields:

- `id` (Integer, Primary Key): Unique user identifier
- `email` (String, Unique, Indexed): User's email address (used for login)
- `name` (String): User's display name
- `google_id` (String, Unique, Indexed): Google OAuth user ID (if logged in via Google)
- `created_at` (DateTime): Account creation timestamp
- `updated_at` (DateTime): Last update timestamp

****Example Data:****

id: 1
email: "john.doe@forsysinc.com"
name: "John Doe"
google_id: "123456789012345678901"
created_at: "2025-01-15 10:30:00"
updated_at: "2025-01-15 10:30:00"

****Relationships:****

- One user can have many sessions
- One user can have many projects
- One user can have many conversations
- One user can upload many files

4.2.2 Sessions Table (`sessions`)

****Purpose:**** Manages user authentication sessions

****Fields:****

- `id` (String, Primary Key): Unique session identifier (UUID format)
- `user_id` (Integer, Indexed): Foreign key to users.id
- `is_active` (Boolean): Whether session is currently active
- `created_at` (DateTime): Session creation timestamp
- `expires_at` (DateTime): Session expiration timestamp

****Example Data:****

id: "session_abc123-def456-ghi789"
user_id: 1
is_active: true
created_at: "2025-01-15 10:30:00"
expires_at: "2025-01-16 10:30:00"

****Relationships:****

- Many sessions belong to one user (user_id → users.id)

****Usage:****

- Created when user logs in
- Used to validate user authentication
- Expires after 24 hours (configurable)

4.2.3 Projects Table (`projects`)

****Purpose:**** Stores project information for organizing work

****Fields:****

- `id` (String, Primary Key): Unique project identifier (UUID format)
- `name` (String, Not Null): Project name
- `user_email` (String, Indexed, Not Null): Owner's email address
- `created_at` (DateTime): Project creation timestamp
- `updated_at` (DateTime): Last update timestamp

****Example Data:****

```
id: "proj_xyz789-abc123-def456"  
name: "Q1 2025 Product Launch"  
user_email: "john.doe@forsysinc.com"  
created_at: "2025-01-15 11:00:00"  
updated_at: "2025-01-20 14:30:00"
```

****Relationships:****

- One project belongs to one user (user_email → users.email)
- One project can have many conversations (project_id → conversations.project_id)

****Usage:****

- Users create projects to organize their work
- Each project can contain multiple conversations
- Projects help group related chats and files together

4.2.4 Conversations Table (`conversations`)

****Purpose:**** Stores chat conversation data in JSON format

****Fields:****

- `id` (Integer, Primary Key): Unique conversation identifier
- `conversation_id` (String, Unique, Indexed): Alternative conversation ID (UUID)
- `chat_id` (String, Indexed, Not Null): Chat session identifier
- `user_email` (String, Indexed): Owner's email address
- `project_id` (String, Indexed, Nullable): Foreign key to projects.id (null if not part of project)
- `conversation_json` (JSON, Not Null): Complete conversation data in JSON format
- `created_at` (DateTime): Conversation creation timestamp
- `updated_at` (DateTime): Last update timestamp

****conversation_json Structure:****

```
{  
  "conversation_id": 1,  
  "messages": [  
    {  
      "message_id": 0,  
      "user": "What is the project timeline?",  
      "assistant": "Based on the documents, the project timeline is..."  
    },  
    {  
      "message_id": 1,  
      "user": "What are the risks?",  
      "assistant": "The identified risks include..."  
    }  
  ]  
}
```

****Example Data:****

```
id: 1  
conversation_id: "conv_abc123-def456"  
chat_id: "chat_xyz789-abc123"  
user_email: "john.doe@forsysinc.com"  
project_id: "proj_xyz789-abc123-def456"  
conversation_json: {"conversation_id": 1, "messages": [...]}  
created_at: "2025-01-15 11:30:00"  
updated_at: "2025-01-15 12:00:00"
```

****Relationships:****

- One conversation belongs to one user (user_email → users.email)
- One conversation can belong to one project (project_id → projects.id, nullable)
- One conversation has one chat_id (used for message retrieval)

****Usage:****

- Stores complete conversation history in JSON format
- Allows efficient retrieval of all messages for a chat session
- Links conversations to projects for organization

4.2.5 Chat Messages Table (`chat_messages`)

****Purpose:**** Stores individual chat messages (legacy/alternative storage method)

****Fields:****

- `id` (String, Primary Key): Unique message identifier (UUID)
- `chat_id` (String, Indexed, Not Null): Chat session identifier
- `user_email` (String, Indexed): Sender's email address
- `role` (String, Not Null): Message role ("user" or "assistant")
- `message` (Text, Not Null): Message content
- `created_at` (DateTime): Message creation timestamp

****Example Data:****

```
id: "msg_abc123-def456"
chat_id: "chat_xyz789-abc123"
user_email: "john.doe@forsysinc.com"
role: "user"
message: "What is the project timeline?"
created_at: "2025-01-15 11:30:00"
```

****Relationships:****

- Many messages belong to one chat session (chat_id)
- Many messages belong to one user (user_email → users.email)

****Usage:****

- Alternative storage method for individual messages
- Used for real-time message saving
- Can be queried to retrieve message history

4.2.6 Uploaded Files Table (`uploaded_files`)

****Purpose:**** Stores metadata about user-uploaded files

****Fields:****

- `id` (Integer, Primary Key): Unique file identifier
- `file_name` (String, Not Null): Original filename
- `file_type` (String, Not Null): File extension (pdf, docx, xlsx, etc.)
- `file_path` (String, Not Null): Physical file path on server
- `uploaded_by` (String, Indexed): Uploader's email address
- `upload_time` (DateTime): Upload timestamp
- `status` (String): Processing status ("Uploaded", "Processing", "Processed", "Error")
- `extracted_text` (Text): Full text content extracted from file
- `indexing_status` (String): Pinecone indexing status ("pending_index", "indexed", "error")

****Example Data:****

```
id: 1
file_name: "Project_Plan_2025.pdf"
```

```
file_type: "pdf"
file_path: "uploads/user123/Project_Plan_2025.pdf"
uploaded_by: "john.doe@forsysinc.com"
upload_time: "2025-01-15 10:00:00"
status: "Processed"
extracted_text: "This document outlines the project plan for 2025..."
indexing_status: "indexed"
```

****Relationships:****

- Many files belong to one user (uploaded_by → users.email)

****Usage:****

- Tracks all user-uploaded files
- Stores extracted text for search
- Monitors indexing status for Pinecone vector database
- File is physically stored in `backend/uploads/` directory

****File Processing Flow:****

1. User uploads file → `status = "Uploaded"`
2. Backend extracts text → `status = "Processing"`
3. Text is chunked and embedded
4. Vectors are stored in Pinecone → `indexing_status = "indexed"`
5. `status = "Processed"`

4.2.7 Mandatory Files Table (`mandatory_files`)

****Purpose:**** Stores system-wide template files that can be used across projects

****Fields:****

- `id` (Integer, Primary Key): Unique file identifier
- `file_name` (String, Not Null): Original filename
- `file_type` (String, Not Null): File extension (pdf, docx, xlsx, etc.)
- `file_path` (String, Nullable): Legacy file path (deprecated)
- `file_content` (LargeBinary, Nullable): File content stored in database
- `file_size` (Integer): File size in bytes
- `uploaded_by` (String, Indexed): Uploader's email (admin)
- `uploaded_at` (DateTime): Upload timestamp
- `description` (String): Optional file description
- `is_active` (Boolean): Soft delete flag (true = active, false = deleted)
- `extracted_text` (Text): Extracted text for search/indexing

****Example Data:****

```
id: 1
file_name: "Project Management Playbook.docx"
file_type: "docx"
file_path: null
file_content: <binary data>
file_size: 245760
uploaded_by: "admin@forsysinc.com"
uploaded_at: "2025-01-10 09:00:00"
description: "Standard project management templates"
is_active: true
extracted_text: "This playbook contains templates for..."
```

****Relationships:****

- Many mandatory files can be linked to users via ProjectKnowledgeBaseFile
- One mandatory file can be used by many users

****Usage:****

- Admin users upload template files
- Files are stored in database (not on disk)
- Users can download and use these files
- Files can be marked for project knowledge base
- Files are indexed in Pinecone for search

4.2.8 Project Knowledge Base Files Table (`project_knowledge_base_files`)

****Purpose:**** Links mandatory files to users for project use

****Fields:****

- `id` (Integer, Primary Key): Unique identifier
- `user_email` (String, Indexed, Not Null): User's email address
- `mandatory_file_id` (Integer, Foreign Key, Not Null): Foreign key to mandatory_files.id
- `created_at` (DateTime): Link creation timestamp
- `updated_at` (DateTime): Last update timestamp

****Unique Constraint:****

- Combination of `user_email` and `mandatory_file_id` must be unique (prevents duplicates)

****Example Data:****

```
id: 1
user_email: "john.doe@forsysinc.com"
mandatory_file_id: 1
```

created_at: "2025-01-15 11:00:00"
updated_at: "2025-01-15 11:00:00"

****Relationships:****

- Many links belong to one user (user_email → users.email)
- Many links point to one mandatory file (mandatory_file_id → mandatory_files.id)

****Usage:****

- When user marks a mandatory file as "Use for Project", a record is created here
- These files are included in chatbot context when user asks questions
- Allows users to have personalized knowledge base from mandatory templates

4.2.9 Workspaces Table (`workspaces`)

****Purpose:**** Stores workspace/organization information (legacy feature)

****Fields:****

- `id` (Integer, Primary Key): Unique workspace identifier
- `name` (String, Unique, Not Null): Workspace name
- `description` (Text): Workspace description
- `is_default` (Boolean): Whether this is the default workspace
- `created_at` (DateTime): Creation timestamp
- `updated_at` (DateTime): Last update timestamp

****Note:**** This table exists but workspace functionality has been removed from the UI.

4.2.10 Documents Table (`documents`)

****Purpose:**** Stores document prompts/templates (legacy feature)

****Fields:****

- `id` (Integer, Primary Key): Unique document identifier
- `feature` (String, Not Null): Feature name (e.g., "sprint", "risk")
- `prompt` (Text, Not Null): Prompt template content
- `uploaded_at` (DateTime): Creation timestamp

****Note:**** This table exists but may not be actively used in current version.

4.2.11 Feedback Table (`feedback`)

****Purpose:**** Stores user feedback (legacy feature)

****Fields:****

- `id` (Integer, Primary Key): Unique feedback identifier
- `name` (String): User's name
- `email` (String): User's email
- `clarity_of_sprint_goals` (String): Rating 1-5
- `workload_distribution` (String): Rating 1-5
- `plan_alignment_sow` (String): Yes/No/Partial
- `suggestions_sprint_planning` (Text): Text feedback
- `risks_clear` (String): Yes/No
- `mitigation_practical` (String): Yes/No
- `suggestions_risk_assessment` (Text): Text feedback
- `overall_sprint_planning_rating` (String): Rating 1-5
- `overall_risk_assessment_rating` (String): Rating 1-5
- `additional_comments` (Text): Additional feedback
- `created_at` (DateTime): Submission timestamp
- `created_by` (String, Indexed): User email

****Note:**** Feedback functionality has been removed from the UI, but table structure remains.

4.3 Entity Relationship Diagram (ERD)



4.4 Key Relationships Summary

1. **User → Sessions:** One-to-Many (one user can have multiple active sessions)
 2. **User → Projects:** One-to-Many (one user can create multiple projects)
 3. **User → Conversations:** One-to-Many (one user can have multiple conversations)
 4. **User → Uploaded Files:** One-to-Many (one user can upload multiple files)
 5. **Project → Conversations:** One-to-Many (one project can contain multiple conversations)
 6. **Conversation → Chat Messages:** One-to-Many (one conversation has multiple messages)
 7. **Mandatory File → Project Knowledge Base Files:** One-to-Many (one template can be used by many users)
 8. **User → Project Knowledge Base Files:** One-to-Many (one user can link multiple templates)

4.5 Indexes and Performance

****Indexed Columns:****

- `users.email` - Fast user lookup by email
 - `users.google_id` - Fast OAuth user lookup
 - `sessions.user_id` - Fast session lookup by user
 - `conversations.chat_id` - Fast conversation lookup by chat ID
 - `conversations.user_email` - Fast conversation lookup by user email
 - `conversations.project_id` - Fast project conversation lookup

- `chat_messages.chat_id` - Fast message lookup by chat
- `uploaded_files.uploaded_by` - Fast file lookup by user
- `mandatory_files.uploaded_by` - Fast template lookup
- `project_knowledge_base_files.user_email` - Fast user template lookup

****Why Indexes Matter:****

- Indexes speed up database queries
- Without indexes, database must scan entire tables
- With indexes, database can quickly find specific rows
- Essential for large datasets

5. API Endpoints

5.1 Overview

The PM Portal backend exposes a RESTful API built with FastAPI. All endpoints are prefixed with `/api/` and return JSON responses. The API handles authentication, file uploads, chat operations, project management, and vector search.

****Base URL:**** `http://localhost:8000` (development)

****API Documentation:**** `http://localhost:8000/docs` (Swagger UI)

5.2 Authentication Endpoints

5.2.1 GET `/api/auth/google/url`

****Purpose:**** Get Google OAuth authentication URL

****Method:**** GET

****Parameters:****

- `prompt` (query, optional): OAuth prompt type (e.g., "select_account")

****Response:****

```
{  
  "auth_url": "https://accounts.google.com/o/oauth2/auth?client_id...",  
  "error": null  
}
```

****Example Request:****

GET http://localhost:8000/api/auth/google/url?prompt=select_account

****Example Response:****

```
{  
  "auth_url": "https://accounts.google.com/o/oauth2/auth?client_id=123456789.apps.googleusercontent.com&redirect_uri=http://localhost:3000/callback&scope=openid%20email%20profile&response_type=code&access_type=offline&prompt=select_account",  
  "error": null  
}
```

****Frontend Usage:****

- Called when user clicks "Login with Google"
- Frontend redirects user to `auth_url`
- User authorizes on Google's page
- Google redirects to `/callback` with authorization code

5.2.2 POST `/api/auth/google/callback`

****Purpose:**** Handle Google OAuth callback and authenticate user

****Method:**** POST

****Request Body:****

```
{  
  "code": "4/0AeanS0..."  
}
```

****Response:****

```
{  
  "success": true,  
  "session_id": "session_abc123-def456",  
  "user": {  
    "id": 1,  
    "email": "user@example.com",  
    "name": "John Doe",  
    "google_id": "123456789012345678901"  
  },  
  "message": "Login successful"  
}
```

****Example Request:****

POST http://localhost:8000/api/auth/google/callback

Content-Type: application/json

```
{  
  "code": "4/0AeanS0AbCdEfGhljKIMnOpQrStUvWxYz"  
}
```

****Flow:****

1. Frontend receives authorization code from Google
2. Sends code to this endpoint
3. Backend exchanges code for access token
4. Backend gets user info from Google
5. Backend creates/updates user in database
6. Backend creates session
7. Returns user info and session ID

5.2.3 POST `/api/auth/login`

****Purpose:**** Simulate login (demo/testing)

****Method:**** POST

****Request Body:****

```
{  
  "email": "demo@example.com",  
  "name": "Demo User",  
  "google_id": "demo_123"  
}
```

****Response:****

```
{  
  "success": true,  
  "message": "Login successful (demo mode)",  
  "session_id": "demo_session_1234567890",  
  "user": {  
    "id": 1,  
    "email": "demo@example.com",  
    "name": "Demo User",  
    "google_id": "demo_123"  
  }  
}
```

5.2.4 POST `/api/auth/logout`

Purpose: Logout user and invalidate session

Method: POST

Request Body: None (uses session cookie)

Response:

```
{  
  "success": true,  
  "message": "Logout successful"  
}
```

5.2.5 GET `/api/auth/session`

Purpose: Validate current session

Method: GET

Parameters: None (uses session cookie)

Response:

```
{  
  "success": true,  
  "session": {  
    "id": "session_abc123",  
    "user_id": 1,  
    "is_active": true  
  },  
  "user": {  
    "id": 1,  
    "email": "user@example.com",  
    "name": "John Doe"  
  }  
}
```

5.2.6 GET `/api/auth/login-by-email`

Purpose: Login user by email address (internal use)

Method: GET

Parameters:

- `email` (query, required): User's email address
- `name` (query, optional): User's name

****Response:****

```
{
  "success": true,
  "session_id": "session_abc123-def456",
  "user": {
    "id": 1,
    "email": "user@forsysinc.com",
    "name": "John Doe",
    "google_id": "email_abc123def456"
  },
  "message": "Login successful by email"
}
```

****Restrictions:****

- Only `@forsysinc.com` emails allowed
- Creates user if doesn't exist
- Creates new session

5.3 Chat Endpoints

5.3.1 POST `/api/chat`

****Purpose:**** Send a message to the chatbot and get AI response

****Method:**** POST

****Request Body (FormData):****

- `user_message` (string, required): User's question/message
- `chat_id` (string, required): Chat session identifier
- `user_email` (string, required): User's email address
- `uploaded_file_ids` (string, optional): JSON array of uploaded file IDs
- `playbook_file_ids` (string, optional): JSON array of mandatory file IDs
- `project_id` (string, optional): Project ID if conversation belongs to project

****Response:****

```
{
  "success": true,
  "response": "Based on the documents you've provided... ",
```

```
"chat_id": "chat_abc123-def456"  
}
```

****Example Request:****

```
POST http://localhost:8000/api/chat  
Content-Type: multipart/form-data
```

```
user_message: "What is the project timeline?"  
chat_id: "chat_abc123-def456"  
user_email: "user@example.com"  
uploaded_file_ids: "[1, 2]"  
playbook_file_ids: "[5, 6]"  
project_id: "proj_xyz789"
```

****Flow:****

1. Backend receives message and file IDs
2. Retrieves files from database
3. Searches Pinecone for relevant chunks
4. Calls Gemini API with question + context
5. Saves message to database
6. Returns AI response

5.3.2 POST `/api/ask-question`

****Purpose:**** Ask a question with file context (alternative chat endpoint)

****Method:**** POST

****Request Body (FormData):****

- `question` (string, required): User's question
- `file_id` (integer, optional): Specific uploaded file ID
- `file_context` (string, optional): Pre-extracted file context
- `mandatory_file_ids` (string, optional): JSON array of mandatory file IDs
- `chat_id` (string, optional): Chat session ID
- `user_email` (string, optional): User's email

****Response:****

```
{  
  "success": true,  
  "response": "The project timeline is... ",  
  "context_chunks": [  
    {  
      "text": "The project will be completed in Q2 2025... ",  
      "start": 0,  
      "end": 100  
    }  
  ]  
}
```

```
"score": 0.95,  
"file_name": "Project_Plan.pdf"  
}  
],  
"chat_id": "chat_abc123"  
}
```

****Usage:****

- Used when user wants to ask about specific files
- Supports vector search across multiple files
- Returns relevant context chunks with similarity scores

5.3.3 GET `/api/chat/sessions`

****Purpose:**** Get list of chat sessions for a user

****Method:**** GET

****Parameters:****

- `user_email` (query, required): User's email address

****Response:****

```
{  
  "success": true,  
  "chats": [  
    {  
      "chat_id": "chat_abc123",  
      "conversation_id": 1,  
      "first_message_preview": "What is the project timeline?",  
      "last_message_at": "2025-01-15T12:00:00Z",  
      "updated_at": "2025-01-15T12:00:00Z"  
    }  
  ]  
}
```

****Note:**** Only returns conversations without `project_id` (project conversations shown separately)

5.3.4 GET `/api/chat/messages`

****Purpose:**** Get all messages for a specific chat session

****Method:**** GET

****Parameters:****

- `chat_id` (query, required): Chat session identifier
- `user_email` (query, required): User's email address

****Response:****

```
{  
  "success": true,  
  "messages": [  
    {  
      "id": "msg_abc123",  
      "chat_id": "chat_abc123",  
      "user_email": "user@example.com",  
      "role": "user",  
      "message": "What is the project timeline?",  
      "created_at": "2025-01-15T11:30:00Z"  
    },  
    {  
      "id": "msg_def456",  
      "chat_id": "chat_abc123",  
      "user_email": "user@example.com",  
      "role": "assistant",  
      "message": "Based on the documents...",  
      "created_at": "2025-01-15T11:30:05Z"  
    }  
  ],  
  "project": {  
    "id": "proj_xyz789",  
    "name": "Q1 2025 Launch"  
  }  
}
```

5.3.5 POST `/api/chat/save-message`

****Purpose:**** Save a single chat message to database

****Method:**** POST

****Request Body (FormData):****

- `chat_id` (string, required): Chat session ID
- `role` (string, required): "user" or "assistant"
- `message` (string, required): Message content

- `user_email` (string, required): User's email
- `project_id` (string, optional): Project ID if applicable

****Response:****

```
{  
  "success": true,  
  "message_id": "msg_abc123",  
  "conversation_id": 1  
}
```

5.3.6 POST `/api/chat/save-conversation`

****Purpose:**** Save complete conversation to database

****Method:**** POST

****Request Body (FormData):****

- `chat_id` (string, required): Chat session ID
- `user_email` (string, required): User's email
- `conversation_json` (string, required): JSON string of conversation
- `project_id` (string, optional): Project ID

****Response:****

```
{  
  "success": true,  
  "conversation_id": 1  
}
```

5.3.7 POST `/api/chat/create`

****Purpose:**** Create a new chat session

****Method:**** POST

****Request Body (FormData):****

- `user_email` (string, required): User's email
- `project_id` (string, optional): Project ID

****Response:****

```
{  
  "success": true,  
  "chat_id": "chat_abc123-def456",  
  "user_email": "user@example.com",  
  "project_id": null  
}
```

```
"conversation_id": 1  
}  
  
---
```

5.4 File Upload Endpoints

5.4.1 POST `/api/upload-file`

****Purpose:**** Upload and process a file (PDF, DOCX, XLSX)

****Method:**** POST

****Request Body (FormData):****

- `file` (file, required): The file to upload
- `user_email` (string, required): Uploader's email

****Response:****

```
{  
  "success": true,  
  "file_id": 1,  
  "file_name": "Project_Plan_2025.pdf",  
  "message": "File uploaded and indexed successfully",  
  "indexing_status": "indexed"  
}
```

****Processing Steps:****

1. File is saved to `backend/uploads/` directory
2. Text is extracted (PDF → pdfplumber, DOCX → python-docx, XLSX → xlsx)
3. Text is saved to database
4. Text is chunked (400 chars per chunk, 100 char overlap)
5. Chunks are converted to embeddings (384-dim vectors)
6. Embeddings are stored in Pinecone
7. Database is updated with `indexing_status = "indexed"

****Example Request:****

POST <http://localhost:8000/api/upload-file>

Content-Type: multipart/form-data

```
file: <binary file data>  
user_email: "user@example.com"
```

5.4.2 POST `/api/upload/sow`

Purpose: Upload Statement of Work (SOW) document (legacy)

Method: POST

Request Body (FormData):

- `file` (file, required): SOW file
- `user_email` (string, optional): Uploader's email

Response:

```
{  
  "success": true,  
  "message": "SOW file processed successfully"  
}
```

5.5 Mandatory Files Endpoints

5.5.1 GET `/api/mandatory-files`

Purpose: Get list of all mandatory template files

Method: GET

Parameters: None

Response:

```
{  
  "success": true,  
  "files": [  
    {  
      "id": 1,  
      "file_name": "Project Management Playbook.docx",  
      "file_type": "docx",  
      "file_size": 245760,  
      "uploaded_by": "admin@forsysinc.com",  
      "uploaded_at": "2025-01-10T09:00:00Z",  
      "description": "Standard templates",  
      "is_active": true,  
      "module": "PM Template"  
    }  
  ]  
}
```

5.5.2 POST `/api/mandatory-files/upload`

****Purpose:**** Upload a new mandatory file (admin only)

****Method:**** POST

****Request Body (FormData):****

- `file` (file, required): Template file to upload
- `user_email` (string, required): Admin's email
- `description` (string, optional): File description
- `module` (string, optional): Module name (e.g., "PM Template")

****Response:****

```
{  
  "success": true,  
  "file_id": 1,  
  "message": "Mandatory file uploaded successfully"  
}
```

****Note:**** File content is stored in database (not on disk)

5.5.3 GET `/api/mandatory-files/{file_id}/download`

****Purpose:**** Download a mandatory file

****Method:**** GET

****Path Parameters:****

- `file_id` (integer, required): File ID

****Response:**** Binary file content with appropriate Content-Type header

****Example:****

```
GET http://localhost:8000/api/mandatory-files/1/download
```

5.5.4 DELETE `/api/mandatory-files/{file_id}`

****Purpose:**** Delete a mandatory file (admin only, soft delete)

****Method:**** DELETE

****Path Parameters:****

- `file_id` (integer, required): File ID

Query Parameters:

- `user_email` (string, required): Admin's email

Response:

```
{  
  "success": true,  
  "message": "Mandatory file deleted successfully"  
}
```

Note: Sets `is_active = false` (soft delete)

5.6 Project Knowledge Base Endpoints

5.6.1 POST `/api/project-knowledge-base/add`

Purpose: Add a mandatory file to user's project knowledge base

Method: POST

Request Body (FormData):

- `user_email` (string, required): User's email
- `mandatory_file_id` (integer, required): Mandatory file ID

Response:

```
{  
  "success": true,  
  "message": "File added to project knowledge base"  
}
```

5.6.2 DELETE `/api/project-knowledge-base/remove`

Purpose: Remove a file from user's project knowledge base

Method: DELETE

Query Parameters:

- `user_email` (string, required): User's email
- `mandatory_file_id` (integer, required): File ID

Response:

```
{  
  "success": true,  
  "message": "File removed from project knowledge base"  
}
```

5.6.3 GET `/api/project-knowledge-base`

Purpose: Get user's project knowledge base files

Method: GET

Query Parameters:

- `user_email` (string, required): User's email

Response:

```
{  
  "success": true,  
  "files": [  
    {  
      "id": 1,  
      "file_name": "Project Management Playbook.docx",  
      "mandatory_file_id": 1  
    }  
  ]  
}
```

5.6.4 POST `/api/project-knowledge-base/reindex-all`

Purpose: Reindex all mandatory files in Pinecone (admin only)

Method: POST

Request Body (FormData):

- `user_email` (string, required): Admin's email

Response:

```
{  
  "success": true,  
  "message": "Reindexing started",  
  "files_processed": 5  
}
```

Usage:

- Recreates Pinecone indexes for all mandatory files
- Useful after updating file content or fixing indexing errors

5.7 Project Management Endpoints

5.7.1 POST `/api/projects`

Purpose: Create a new project

Method: POST

Request Body (FormData):

- `name` (string, required): Project name
- `user_email` (string, required): Owner's email

Response:

```
{  
  "success": true,  
  "project": {  
    "id": "proj_abc123-def456",  
    "name": "Q1 2025 Product Launch",  
    "user_email": "user@example.com",  
    "created_at": "2025-01-15T11:00:00Z"  
  },  
  "conversation": {  
    "id": 1,  
    "conversation_id": "conv_xyz789",  
    "chat_id": "chat_abc123",  
    "title": "Default chat",  
    "project_id": "proj_abc123-def456"  
  }  
}
```

Note: Creates a default conversation for the project automatically

5.7.2 GET `/api/projects`

Purpose: Get all projects for a user with their conversations

Method: GET

****Query Parameters:****

- `user_email` (string, required): User's email

****Response:****

```
{  
  "success": true,  
  "projects": [  
    {  
      "id": "proj_abc123",  
      "name": "Q1 2025 Launch",  
      "user_email": "user@example.com",  
      "created_at": "2025-01-15T11:00:00Z",  
      "conversations": [  
        {  
          "id": 1,  
          "conversation_id": "conv_xyz789",  
          "chat_id": "chat_abc123",  
          "first_message_preview": "What is the timeline?",  
          "title": "Default chat"  
        }  
      ]  
    }  
  ]  
}
```

5.7.3 DELETE `/api/projects/{project_id}`

****Purpose:**** Delete a project

****Method:**** DELETE

****Path Parameters:****

- `project_id` (string, required): Project ID

****Query Parameters:****

- `user_email` (string, required): Owner's email

****Response:****

```
{  
  "success": true,  
  "message": "Project deleted successfully"  
}
```

****Note:**** Also deletes associated conversations

5.8 AI/LLM Endpoints

5.8.1 POST `/api/gemini/chat`

****Purpose:**** Direct Gemini API chat endpoint

****Method:**** POST

****Request Body (JSON):****

```
{  
  "messages": [  
    {"role": "user", "content": "Hello"}  
,  
    "max_tokens": 3000  
  ]}
```

****Response:****

```
{  
  "success": true,  
  "response": "Hello! How can I help you?",  
  "usage": {  
    "prompt_tokens": 5,  
    "completion_tokens": 10  
  },  
  "api_key_used": "#1"  
}
```

****Features:****

- Multi-key fallback support (3 API keys)
- Automatic rate limit handling
- Token usage tracking

5.8.2 POST `/api/llm/chat`

****Purpose:**** Mock LLM chat endpoint (legacy)

****Method:**** POST

****Request Body (JSON):****

```
{  
  "message": "Hello",  
  "context": []  
}  
  
**Response:**  
{  
  "response": "Mock response",  
  "is_complete": true  
}
```

5.9 Health Check Endpoint

5.9.1 GET `/`

Purpose: Health check endpoint

Method: GET

Response:

```
{  
  "message": "PM Portal Bot API",  
  "status": "running"  
}
```

5.10 Static File Endpoints

5.10.1 GET `/mandatory/{filename:path}`

Purpose: Serve mandatory files from disk (legacy)

Method: GET

Path Parameters:

- `filename` (path): File path relative to mandatory directory

Response: Binary file content

5.11 Error Responses

All endpoints return errors in this format:

```
{  
  "success": false,  
  "error": "Error message here",  
  "message": "Detailed error description"  
}
```

****Common HTTP Status Codes:****

- `200 OK`: Success
- `400 Bad Request`: Invalid request data
- `401 Unauthorized`: Authentication required
- `404 Not Found`: Resource not found
- `500 Internal Server Error`: Server error

6. Authentication & Authorization

6.1 Overview

PM Portal supports two authentication methods:

1. **Google OAuth 2.0** - Industry-standard OAuth flow for Google account login
2. **Email Login** - Direct email-based authentication for internal users (@forsysinc.com)

Both methods create user accounts in the database and establish session-based authentication.

6.2 Google OAuth Flow

6.2.1 Step-by-Step Process

****Step 1: User Initiates Login****

User clicks "Login with Google" button on LandingPage.js

↓

Frontend calls: GET /api/auth/google/url

↓

Backend (auth_service.py) generates OAuth URL:

- client_id: From GOOGLE_CLIENT_ID env var
- redirect_uri: http://localhost:3000/callback
- scope: openid email profile
- response_type: code
- access_type: offline

↓

Backend returns auth_url to frontend

↓

Frontend redirects user to Google's login page

****Step 2: User Authorizes on Google****

User sees Google login page

↓

User enters Google credentials

↓

User clicks "Allow" to authorize application

↓

Google redirects to: http://localhost:3000/callback?code=AUTHORIZATION_CODE

****Step 3: Backend Processes Callback****

Frontend (GoogleCallback.js) receives code from URL

↓

Frontend sends: POST /api/auth/google/callback

Body: { "code": "AUTHORIZATION_CODE" }

↓

Backend (auth_service.py) processes:

1. Exchanges code for access token

POST https://oauth2.googleapis.com/token

{

client_id, client_secret, code, grant_type, redirect_uri

}

2. Gets user info from Google

GET https://www.googleapis.com/oauth2/v2/userinfo

Headers: { Authorization: "Bearer ACCESS_TOKEN" }

3. Checks if user exists in database (by google_id)

4. If new user: Creates User record

5. Creates Session record (24-hour expiration)

6. Returns user info and session_id

↓

Backend returns:

{

"success": true,

```
"session_id": "session_abc123",
"user": { id, email, name, google_id }
}
```

Step 4: Frontend Stores Session

Frontend receives response

↓

Stores in localStorage:

- user: JSON.stringify(user)
- sessionId: session_id

↓

Updates AuthContext state:

- setUser(user)
- setSessionId(session_id)

↓

Redirects to /home (chatbot interface)

6.2.2 Code Flow Diagram



```
[REDACTED]  
■  
■ POST /api/auth/google/callback  
▼  
[REDACTED]  
■ auth_service ■ 1. Exchange code → token  
■ authenticate_ ■ 2. Get user info  
■ user() ■ 3. Create/update user  
■ ■ 4. Create session  
[REDACTED]  
■  
■ Returns user + session_id  
▼  
[REDACTED]  
■ GoogleCallback ■ Stores in localStorage  
■ ■ Updates AuthContext  
■ ■ Redirects to /home
```

6.2.3 Session Management

Session Creation:

- Session ID format: `session_{uuid}`
- Expiration: 24 hours from creation
- Stored in `sessions` table with `user_id` foreign key

Session Validation:

- Frontend stores `sessionId` in localStorage
- Backend can validate session via `GET /api/auth/session`
- Session is checked on protected routes

Session Expiration:

- After 24 hours, `expires_at` timestamp is checked
- Expired sessions return `is_active = false`
- User must login again

6.3 Email Login Flow

6.3.1 Step-by-Step Process

Step 1: User Accesses Email Login URL

User visits: /email-login?email=user@forsysinc.com&name=John%20Doe

↓

Frontend (EmailLoginCallback.js) extracts email and name from URL

↓

Checks if user is already authenticated

If yes → Redirect to /home

If no → Continue

Step 2: Backend Validates and Creates User

Frontend calls: GET /api/auth/login-by-email?email=user@forsysinc.com&name=John%20Doe

↓

Backend (auth_service.py) processes:

1. Validates email domain (@forsysinc.com only)
2. Searches for user by email in database
3. If user doesn't exist:
 - Creates new User record
 - Generates google_id: "email_{uuid}"
 - Extracts name from email if not provided
4. If user exists:
 - Updates name if provided and different
5. Creates new Session (24-hour expiration)
6. Returns user info and session_id

↓

Backend returns:

```
{  
  "success": true,  
  "session_id": "session_abc123",  
  "user": { id, email, name, google_id }  
}
```

Step 3: Frontend Stores Session

Frontend receives response

↓

Stores in localStorage:

- user: JSON.stringify(user)
- sessionId: session_id

↓

Updates AuthContext state

↓

Redirects to /home

6.3.2 Email Domain Restriction

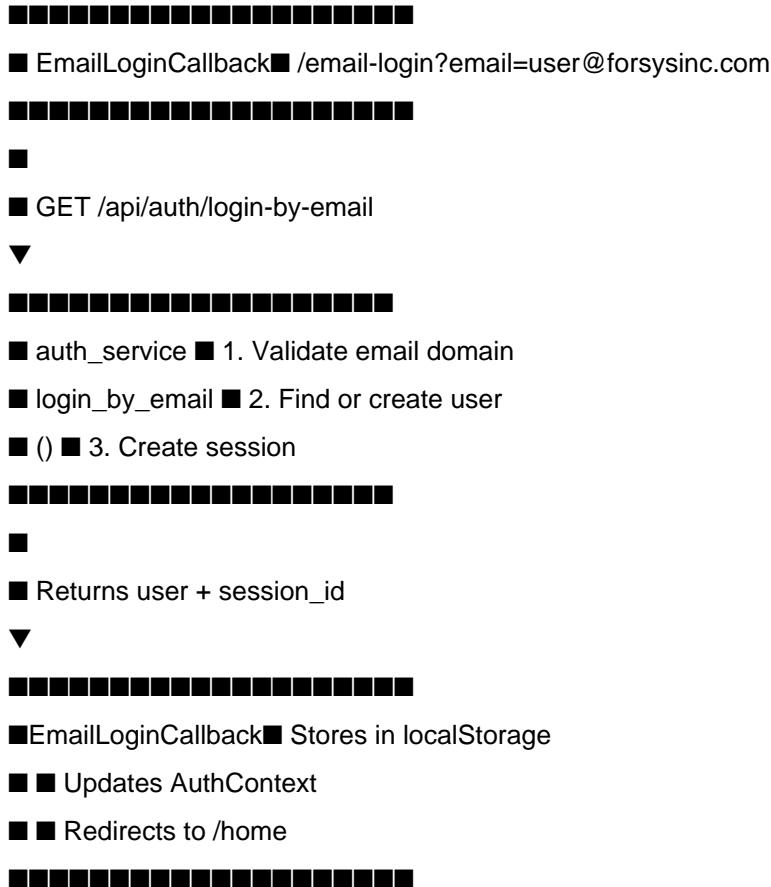
****Allowed Domain:****

- Only `@forsysinc.com` emails are accepted
- Other domains return error: "Access restricted to @forsysinc.com email addresses only"

****Why This Restriction:****

- Internal tool for company use
- Prevents unauthorized access
- Ensures all users are company employees

6.3.3 Code Flow Diagram



6.4 Authentication Context (Frontend)

6.4.1 AuthContext.js Overview

****Purpose:**** Manages authentication state across the entire React application

****Location:**** `frontend/src/context/AuthContext.js`

****Provides:****

- `user`: Current user object (null if not logged in)
- `sessionId`: Current session ID

- `loading`: Boolean indicating if auth check is in progress
- `login()`: Function to login (demo mode)
- `loginWithGoogle()`: Function to initiate Google OAuth
- `logout()`: Function to logout and clear session
- `isAuthenticated`: Boolean computed from user and sessionId

****State Management:****

```
const [user, setUser] = useState(null);
const [sessionId, setSessionId] = useState(null);
const [loading, setLoading] = useState(true);
```

****Initialization:****

- On app load, checks localStorage for saved session
- If found, restores user and sessionId
- Sets `loading = false` after check

****Usage in Components:****

```
import { useAuth } from './contexts/AuthContext';

const MyComponent = () => {
  const { user, isAuthenticated, logout } = useAuth();

  if (!isAuthenticated) {
    return <LoginPage />;
  }

  return <div>Welcome, {user.name}!</div>;
};
```

6.5 Authorization (Access Control)

6.5.1 Protected Routes

****Frontend Protection:****

- Routes in `App.js` check `isAuthenticated` from AuthContext
- Unauthenticated users are redirected to `/` (LandingPage)

****Backend Protection:****

- Most endpoints require `user_email` parameter
- Backend validates user exists in database
- Some endpoints check for admin privileges

6.5.2 Admin-Only Features

****Admin Detection:****

- Checks `user.role === "admin"` OR
- Checks if `user.email` is in admin email list:

```
const ADMIN_EMAILS = ["shaik.sharuk@forsysinc.com"];
const isAdmin = userRole === "admin" || ADMIN_EMAILS.includes(user.email);
```

****Admin-Only Operations:****

- Upload mandatory files
- Delete mandatory files
- Reindex all files in Pinecone
- Set file modules

6.6 Tokens, Sessions, and Cookies

6.6.1 Session Storage

****Frontend (localStorage):****

- `user`: JSON string of user object
- `sessionId`: Session identifier string
- Persists across browser sessions
- Cleared on logout

****Backend (Database):****

- `sessions` table stores session records
- `id`: Session identifier (matches frontend sessionId)
- `user_id`: Links to users table
- `is_active`: Boolean flag
- `expires_at`: Expiration timestamp

6.6.2 No JWT Tokens

****Current Implementation:****

- PM Portal does NOT use JWT tokens
- Uses session-based authentication
- Session ID is stored in localStorage (frontend) and database (backend)

****Why Sessions Instead of JWT:****

- Simpler implementation
- Easier to revoke (just set `is_active = false`)
- No token expiration complexity
- Database-backed sessions provide audit trail

6.6.3 Cookies (Optional)

****Current Usage:****

- Some endpoints use `withCredentials: true` for cookie support
- Cookies can be used for session management (future enhancement)
- Currently, session ID is passed via localStorage and API parameters

6.7 Logout Flow

6.7.1 Step-by-Step Process

User clicks "Sign out" button

↓

Frontend calls logout() from AuthContext

↓

AuthContext.logout():

1. Clears user state: setUser(null)
2. Clears sessionId: setSessionId(null)
3. Removes from localStorage:
 - localStorage.removeItem('sessionId')
 - localStorage.removeItem('user')
4. Clears sessionStorage:
 - sessionStorage.removeItem('hasShownFullscreenChat')
 - sessionStorage.removeItem('lastShownFullscreenChatUserId')
5. (Optional) Calls backend: POST /api/auth/logout

↓

Redirects to / (LandingPage)

6.7.2 Session Invalidation

****Backend Logout:****

- Sets `is_active = false` in sessions table
- Session becomes invalid immediately
- User must login again to access protected routes

6.8 Security Considerations

6.8.1 Email Domain Validation

****Why Important:****

- Prevents unauthorized users from accessing the system

- Ensures only company employees can login
- Validated on both frontend and backend

****Implementation:****

```
allowed_domain = "@forsysinc.com"
if not email.endswith(allowed_domain):
    return LoginResponse(
        success=False,
        message=f"Access restricted to {allowed_domain} email addresses only."
    )
```

6.8.2 Session Expiration

****24-Hour Expiration:****

- Sessions expire after 24 hours
- Forces users to re-authenticate periodically
- Reduces risk of unauthorized access from stolen sessions

6.8.3 HTTPS in Production

****Development:****

- Uses HTTP (<http://localhost:3000>, <http://localhost:8000>)

****Production:****

- MUST use HTTPS
- Protects credentials and session data
- Required for Google OAuth redirect URIs

7. Core Features

7.1 Chatbot Feature

7.1.1 Overview

The chatbot is the ****main feature**** and ****landing page**** of PM Portal. It provides an AI-powered conversational interface that can answer questions based on uploaded documents and mandatory template files. The chatbot uses Google Gemini AI for response generation and Pinecone vector database for document search.

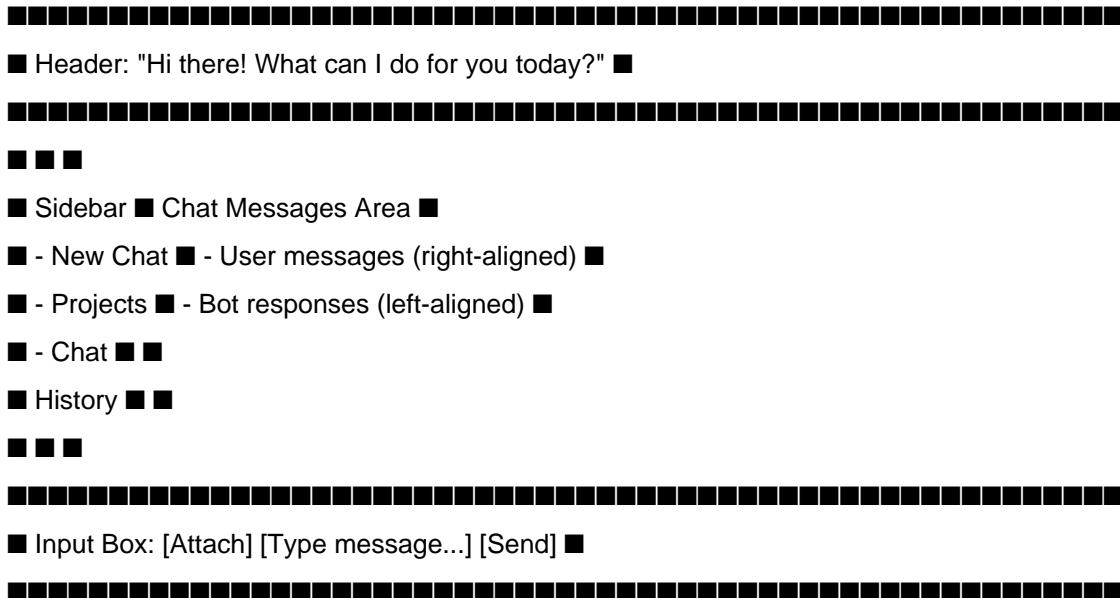
7.1.2 User Interface

****Location:**** `frontend/src/components/HomePage.js`

****Key UI Elements:****

- **Chat Input Box:** Text input at bottom for typing messages
- **Message List:** Scrollable area showing conversation history
- **Sidebar:** Chat history and projects list (collapsible)
- **File Upload Button:** "+" button to attach files
- **Mandatory Files Dropdown:** Access to template files
- **Project Header:** Shows current project name (if in project context)
- **New Chat Button:** Starts a fresh conversation

****Layout:****



7.1.3 How Chatbot Works Internally

****Step 1: User Types Message****

User types: "What is the project timeline?"

↓

Frontend (HomePage.js):

- Stores message in chatInput state
- User presses Enter or clicks Send

↓

Calls: addMessageWithBotResponse(userMessage)

****Step 2: Frontend Sends Request****

```
// HomePage.js - addMessageWithBotResponse()
const formData = new FormData();
formData.append('user_message', userMessage);
formData.append('chat_id', currentChatId);
formData.append('user_email', user.email);
```

```

// Add uploaded file IDs if any
if (uploadedFileIds.length > 0) {
  formData.append('uploaded_file_ids', JSON.stringify(uploadedFileIds));
}

// Add mandatory file IDs if any
if (playbookFileIds.length > 0) {
  formData.append('playbook_file_ids', JSON.stringify(playbookFileIds));
}

// Add project ID if in project context
if (activeProjectId) {
  formData.append('project_id', activeProjectId);
}

// Send to backend
fetch(`${apiUrl}/api/chat`, {
  method: 'POST',
  body: formData
});

```

****Step 3: Backend Receives Request****

Backend (main.py) - POST /api/chat endpoint

↓

Extracts:

- user_message: "What is the project timeline?"
- chat_id: "chat_abc123"
- user_email: "user@example.com"
- uploaded_file_ids: [1, 2]
- playbook_file_ids: [5, 6]
- project_id: "proj_xyz789" (optional)

****Step 4: Backend Retrieves Files****

Get uploaded files

```

uploaded_files = []
if uploaded_file_ids:
  for file_id in uploaded_file_ids:
    file = db.query(UploadedFile).filter(UploadedFile.id == file_id).first()
    if file and file.indexing_status == "indexed":
      uploaded_files.append(file)

```

Get mandatory files (project knowledge base)

```
mandatory_files = []
if playbook_file_ids:
    for file_id in playbook_file_ids:
        file = db.query(MandatoryFile).filter(MandatoryFile.id == file_id).first()
        if file and file.is_active:
            mandatory_files.append(file)
```

Step 5: Backend Searches Pinecone for Relevant Chunks

For Uploaded Files:

Convert question to embedding vector

```
query_embedding = embedding_service.embed_query(question)
```

Returns: [0.123, -0.456, 0.789, ...] (384 dimensions)

Search each file's Pinecone index

```
all_relevant_chunks = []
for file in uploaded_files:
    index_name = pinecone_service.get_index_name_for_file(file.id, file.file_name)
```

**Format:
"kb-file-{file_id}-{sanitized_filename}"**

```
search_result = pinecone_service.search_across_indexes(
    query_embedding=query_embedding,
    index_names=[index_name],
    top_k=5 # Get top 5 most similar chunks
)
```

Each result has:

- **text: Chunk content**
- **score: Similarity score (0.0 to 1.0)**
- **metadata: file_id, file_name, chunk_index**

```
all_relevant_chunks.extend(search_result["results"])
```

For Mandatory Files:

Similar process for mandatory files

```
for file in mandatory_files:  
    index_name = pinecone_service.get_index_name_for_file(file.id, file.file_name)  
    search_result = pinecone_service.search_across_indexes(...)  
    all_relevant_chunks.extend(search_result["results"])
```

Step 6: Backend Builds Context

Sort chunks by similarity score (highest first)

```
all_relevant_chunks.sort(key=lambda x: x["score"], reverse=True)
```

Take top 10 most relevant chunks

```
top_chunks = all_relevant_chunks[:10]
```

Build context string

```
context_text = ""  
for chunk in top_chunks:  
    context_text += f"From {chunk['metadata']['file_name']}:\n"  
    context_text += f"{chunk['text']}\n\n"
```

Step 7: Backend Calls Gemini AI

Prepare messages for Gemini

```
messages = [
{
  "role": "system",
  "content": "You are a helpful assistant. Answer questions based on the provided context."
},
{
  "role": "user",
  "content": f"""Context from documents:
{context_text}

User Question: {question}

Please answer the question based on the context provided. If the context doesn't contain enough
information, say so."""
}
]
```

Call Gemini API

```
gemini_response = gemini_service.chat(messages, max_tokens=3000)
```

Extract response text

```
answer = gemini_response["response"]
```

Step 8: Backend Saves Messages

Save user message

```
_save_chat_message(db, chat_id, "user", question, user_email)
```

Save assistant response

```
_save_chat_message(db, chat_id, "assistant", answer, user_email)
```

Update conversation JSON

```
_update_conversation_json(db, chat_id, question, answer, user_email, project_id)
```

Step 9: Backend Returns Response

```
return {  
  "success": true,  
  "response": answer,  
  "chat_id": chat_id  
}
```

Step 10: Frontend Displays Response

```
// HomePage.js receives response  
const data = await response.json();  
  
// Add bot message to chat  
setChatMessages(prevMessages => [  
  ...prevMessages,  
  {  
    text: data.response,  
    type: 'bot',  
    time: new Date().toLocaleTimeString()  
  }  
]);
```

7.1.4 Vector Search Deep Dive

What is Vector Search?

- Documents are converted to numerical vectors (embeddings)
- Each vector represents the "meaning" of text
- Similar meanings = similar vectors
- Vector search finds documents with similar meaning to the question

How Embeddings Work:

Text: "The project timeline is Q2 2025"

↓

Embedding Model (all-MiniLM-L6-v2)

↓

Vector: [0.123, -0.456, 0.789, ..., 0.234] (384 numbers)

Similarity Calculation:

- Uses cosine similarity (measures angle between vectors)
- Score range: 0.0 (completely different) to 1.0 (identical meaning)
- Higher score = more relevant to question

Example:

Question: "What is the project timeline?"

↓

Question Embedding: [0.1, -0.2, 0.3, ...]

Document Chunks:

Chunk 1: "The project timeline is Q2 2025"

Embedding: [0.12, -0.19, 0.31, ...]

Similarity: 0.95 ■ (Very relevant)

Chunk 2: "The team has 5 members"

Embedding: [0.5, 0.3, -0.1, ...]

Similarity: 0.23 ■ (Not relevant)

7.1.5 Chat Session Management

Chat ID Generation:

```
// Frontend generates unique chat ID
const createChatId = () => {
  if (window.crypto?.randomUUID) {
    return window.crypto.randomUUID();
  }
  return `chat-${Date.now()}-${Math.random().toString(16).slice(2, 10)}`;
};
```

```
const [chatId, setChatId] = useState(() => createChatId());
```

Chat History:

- Each chat session has a unique `chat_id`
- Messages are grouped by `chat_id`
- Frontend displays list of previous chats in sidebar
- User can click a chat to load its history

New Chat:

```
const handleNewChat = () => {
  // Clear current messages
  setChatMessages([]);

  // Generate new chat ID
  const newChatId = createChatId();
  setChatId(newChatId);

  // Clear file attachments
  setUploadedFileIds([]);
  setPlaybookFileIds([]);
```

```
// Clear project context
setActiveProjectId(null);
};
```

7.1.6 File Context in Chat

****Uploaded Files:****

- User can attach files before sending message
- Files are uploaded via `/api/upload-file`
- File IDs are stored in `uploadedFileIds` state
- When user sends message, file IDs are sent to backend
- Backend searches these files' Pinecone indexes

****Mandatory Files (Playbook):****

- User can select mandatory files from dropdown
- Files are marked for project use
- File IDs are stored in `playbookFileIds` state
- These files are always included in chatbot context

****Project Context:****

- If user is in a project, `project_id` is sent with messages
- Messages are saved with project association
- Project conversations are shown under project in sidebar

7.1.7 Response Formatting

****Markdown Support:****

- Gemini responses may include markdown (code blocks, lists, etc.)
- Frontend removes markdown code fences for cleaner display
- HTML is rendered using `dangerouslySetInnerHTML`

****Code Block Handling:****

```
// Remove markdown code block fences
if (botResponseText.includes('```')) {
    botResponseText = botResponseText
        .replace(/\``[a-zA-Z]*\s*\n?/g, "")
        .replace(/\`\s*\n?/g, "")
        .trim();
}
```

7.1.8 Error Handling

****Network Errors:****

```

try {
  const response = await fetch(`/${apiUrl}/api/chat`, {...});
  const data = await response.json();
  // Display response
} catch (error) {
  // Show error message
  setChatMessages(prev => [...prev, {
    text: 'Error: Could not get a response from the bot.',
    type: 'bot',
    time: new Date().toLocaleTimeString()
  }]);
}

```

****Backend Errors:****

- If Gemini API fails, backend returns error response
- Frontend displays error message to user
- User can retry by sending message again

7.2 Project Management Feature

7.2.1 Overview

Projects allow users to organize their work by grouping related conversations and files together. Each project can contain multiple conversations, and each conversation is linked to a specific project. This helps users manage different work streams or clients separately.

7.2.2 Creating a Project

****Frontend Flow:****

```

// HomePage.js - handleCreateProject()
const handleCreateProject = async () => {
  if (!projectName.trim() || !user?.email) return;

  setIsCreatingProject(true);

  const response = await fetch(`/${apiUrl}/api/projects`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      name: projectName,
      user_email: user.email
    })
  });

```

```

    })
});

const data = await response.json();

if (data.success) {
  // Add project to local state
  setProjects(prev => [data.project, ...prev]);

  // Expand project and set as active
  setExpandedProjects(prev => new Set([...prev, data.project.id]));
  setActiveProjectId(data.project.id);

  // Load the default conversation
  if (data.conversation.chat_id) {
    loadChatHistory(data.conversation.chat_id);
  }
}
};

**Backend Flow:**
```

main.py - POST /api/projects

1. Validate project name (not empty)
2. Create Project record:
 - id: UUID string
 - name: User-provided name
 - user_email: Owner's email
3. Create default Conversation:
 - conversation_id: UUID
 - chat_id: UUID
 - project_id: Links to project
 - conversation_json: Empty messages array
4. Return project and conversation data

****Database Changes:****

- New row in `projects` table
- New row in `conversations` table with `project_id` set

7.2.3 Project Structure

****Project Object:****

```
{
}
```

```

"id": "proj_abc123-def456",
"name": "Q1 2025 Product Launch",
"user_email": "user@example.com",
"created_at": "2025-01-15T11:00:00Z",
"updated_at": "2025-01-15T11:00:00Z",
"conversations": [
{
  "id": 1,
  "conversation_id": "conv_xyz789",
  "chat_id": "chat_abc123",
  "first_message_preview": "What is the timeline?",
  "title": "Default chat"
}
]
}

```

7.2.4 Managing Project Conversations

****Adding Conversations:****

- Conversations are automatically created when user sends first message in a project
- Each conversation has a unique `chat_id`
- Conversations are linked to project via `project_id`

****Viewing Conversations:****

- Frontend fetches all projects with conversations: `GET /api/projects?user_email=...`
- Projects are displayed in sidebar with expandable conversation list
- Clicking a conversation loads its chat history

****Deleting Projects:****

```

// Frontend - handleDeleteProject()
const handleDeleteProject = async (projectId) => {
  const response = await fetch(
    `${apiUrl}/api/projects/${projectId}?user_email=${user.email}`,
    { method: 'DELETE' }
  );

  // Refresh projects list
  await fetchProjects();
}

```

****Backend Deletion:****

- Deletes project from `projects` table
- Deletes associated conversations from `conversations` table

- Cascade deletion ensures data consistency

7.2.5 Project Context in Chat

****When User is in a Project:****

- `activeProjectId` state is set
- All chat messages include `project_id` parameter
- Messages are saved with project association
- Project name is displayed in chat header

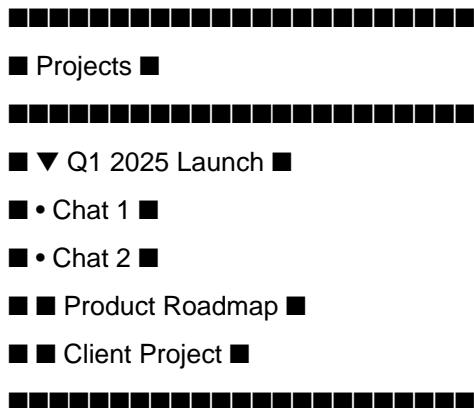
****Switching Between Projects:****

// User clicks on a project conversation

```
const handleProjectConversationClick = (conversation) => {
  setActiveProjectId(conversation.project_id);
  setActiveConversationId(conversation.id);
  loadChatHistory(conversation.chat_id);
};
```

7.2.6 Project Sidebar UI

****Layout:****



****Features:****

- Expandable/collapsible project list
- Nested conversation list under each project
- Active project/conversation highlighting
- Delete project button (with confirmation)

7.3 File Upload & Processing Feature

7.3.1 Overview

Users can upload files (PDF, DOCX, XLSX, TXT) to the chatbot. Files are processed to extract text, which is then chunked and indexed in Pinecone for semantic search. This allows the chatbot to answer questions based on document content.

7.3.2 Supported File Types

****PDF Files:****

- Extracted using `pdfplumber` (primary) and `PyPDF2` (fallback)
- Handles complex layouts, tables, and multi-page documents
- Preserves text structure and formatting

****DOCX Files:****

- Extracted using `python-docx` library
- Preserves paragraph structure
- Extracts text from tables
- Enriches with playbook links (if applicable)

****XLSX Files:****

- Extracted using `openpyxl` library
- Reads all sheets
- Converts cells to tab-separated text
- Preserves row/column structure

****TXT Files:****

- Direct text extraction (UTF-8 encoding)
- No processing needed

7.3.3 File Upload Flow

****Step 1: User Selects File****

```
// HomePage.js - handleFileSelect()  
const handleFileSelect = async (event) => {  
  const files = Array.from(event.target.files);  
  
  // Store selected files  
  setSelectedFiles(files);  
  
  // Show preview chips  
  // User can remove files before uploading  
};
```

****Step 2: Frontend Uploads File****

```
// HomePage.js - handleUploadSelectedFiles()  
const handleUploadSelectedFiles = async () => {
```

```
setIsUploadingFile(true);

const formData = new FormData();
files.forEach(file => {
  formData.append('files', file); // Multiple files support
});
formData.append('uploaded_by', user.email);

const response = await fetch(`/${apiUrl}/api/upload-file`, {
  method: 'POST',
  body: formData
});

const data = await response.json();

if (data.success) {
  // Store file IDs for chat context
  setUploadedFileIds(data.file_ids);
  setSelectedFiles([]);
}
};
```

Step 3: Backend Receives File

main.py - POST /api/upload-file

```
async def upload_chatbot_file(
    request: Request,
    uploaded_by: str = Form(None),
    background_tasks: BackgroundTasks = None,
    db: Session = Depends(get_db)
):
```

Parse form data (supports both 'file' and 'files')

```
form = await request.form()
files = form.getlist('files') or [form.get('file')]
```

Process each file

```
results = []
```

```
for file in files:  
    result = await process_single_file(file, uploaded_by, db, background_tasks)  
    results.append(result)  
  
return {"success": True, "file_ids": [r["file_id"] for r in results]}
```

Step 4: Backend Processes File

main.py - process_single_file()

```
async def process_single_file(file, uploaded_by, db, background_tasks):
```

1. Validate file type

```
file_extension = file.filename.split('.')[1].lower()  
if file_extension not in ['pdf', 'docx', 'txt', 'xlsx']:  
    return {"success": False, "error": "Invalid file type"}
```

2. Save file to disk

```
unique_filename = f"{uuid.uuid4()}-{file.filename}"  
file_path = upload_dir / unique_filename  
file_content = await file.read()  
with open(file_path, "wb") as f:  
    f.write(file_content)
```

3. Extract text based on file type

```
extracted_text = ""  
if file_extension == 'pdf':  
    result = pdf_service.extract_text_from_pdf(file_content)  
    extracted_text = result['text']  
elif file_extension in ['docx', 'doc']:  
    extracted_text = extract_text_with_hyperlinks_from_docx(file_content)  
elif file_extension == 'xlsx':
```

Extract using openpyxl

```
workbook = load_workbook(io.BytesIO(file_content), data_only=True)
```

Convert sheets to text

```
...
elif file_extension == 'txt':
    extracted_text = file_content.decode('utf-8')
```

4. Save to database

```
uploaded_file = UploadedFile(
    file_name=file.filename,
    file_type=file_extension,
    file_path=str(file_path),
    uploaded_by=uploaded_by,
    extracted_text=extracted_text,
    status="Uploaded",
    indexing_status="pending_index"
)
db.add(uploaded_file)
db.commit()
db.refresh(uploaded_file)
```

5. Queue background indexing

```
background_tasks.add_task(
    index_file_background,
    uploaded_file.id,
    extracted_text,
    file.filename,
    file_extension,
    uploaded_by,
    datetime.now()
)

return {"success": True, "file_id": uploaded_file.id}
```

7.3.4 Text Extraction Details

PDF Extraction:

pdf_service.py

```
def extract_text_from_pdf(file_content):
```

Method 1: pdfplumber (better for complex layouts)

```
with pdfplumber.open(io.BytesIO(file_content)) as pdf:  
    text_parts = []  
    for page in pdf.pages:  
        text_parts.append(page.extract_text())
```

Also extract tables

```
tables = page.extract_tables()  
for table in tables:
```

Convert table to text rows

...

Method 2: PyPDF2 (fallback)

```
if not text_parts:  
    pdf_reader = PyPDF2.PdfReader(io.BytesIO(file_content))  
    for page in pdf_reader.pages:  
        text_parts.append(page.extract_text())  
  
    return '\n'.join(text_parts)
```

DOCX Extraction:

docx_extraction_helper.py

```
def extract_text_with_hyperlinks_from_docx(file_content):  
    doc = Document(io.BytesIO(file_content))  
    text_parts = []
```

Extract paragraphs

```
for paragraph in doc.paragraphs:  
    if paragraph.text.strip():  
        text_parts.append(paragraph.text.strip())
```

Extract tables

```
for table in doc.tables:  
    for row in table.rows:  
        row_texts = [cell.text.strip() for cell in row.cells]  
        text_parts.append(' | '.join(row_texts))
```

Enrich with playbook links (if applicable)

```
full_text = '\n'.join(text_parts)  
enriched_text = enrich_text_with_links(full_text)  
  
return enriched_text
```

XLSX Extraction:

Extract from Excel

```
workbook = load_workbook(io.BytesIO(file_content), data_only=True)  
lines = []  
  
for sheet in workbook.worksheets:  
    lines.append(f"Sheet: {sheet.title}")  
    for row in sheet.iter_rows(values_only=True):  
        cells = [str(cell) for cell in row if cell is not None]  
        if cells:  
            lines.append("\t".join(cells))  
  
extracted_text = "\n".join(lines)
```

7.3.5 File Chunking Process

Why Chunking?

- Large documents can't be sent to AI in one piece (token limits)
- Chunking breaks documents into smaller, manageable pieces
- Each chunk is indexed separately in Pinecone
- Allows retrieval of specific relevant sections

Chunking Parameters:

- **Chunk Size:** 400 characters per chunk

- **Chunk Overlap:** 100 characters between chunks
- **Method:** Character-based (not word-based)

Chunking Process:

chunking_service.py

```
def chunk_text_by_characters(text, chunk_size=400, chunk_overlap=100):
    chunks = []
    start = 0

    while start < len(text):
        end = start + chunk_size
        chunk_text = text[start:end].strip()

        if chunk_text:
            chunks.append({
                "text": chunk_text,
                "metadata": {
                    "chunk_index": len(chunks),
                    "chunk_start": start,
                    "chunk_end": min(end, len(text))
                }
            })
        start = end - chunk_overlap

    return chunks
```

Move start forward with overlap

start = end - chunk_overlap

return chunks

Example:

Original Text (1000 chars):

"The project timeline is Q2 2025. The team consists of 5 members..."

Chunk 1 (chars 0-400):

"The project timeline is Q2 2025. The team consists of 5 members..."

Chunk 2 (chars 300-700): # Overlaps with chunk 1

"...team consists of 5 members. The budget is \$100,000..."

Chunk 3 (chars 600-1000): # Overlaps with chunk 2

"...budget is \$100,000. The deliverables include..."

7.3.6 Background Indexing

Why Background Processing?

- Indexing can take time (especially for large files)
- Don't want to block the API response
- User can continue using the app while indexing happens

Background Task:

main.py - index_file_background()

```
def index_file_background(  
    file_id: int,  
    text: str,  
    source_filename: str,  
    file_type: str,  
    uploaded_by: str,  
    uploaded_at  
):  
    try:
```

1. Chunk the text

```
chunks = chunking_service.chunk_text_by_characters(  
    text,  
    chunk_size=400,  
    chunk_overlap=100  
)
```

2. Generate embeddings for each chunk

```
chunk_texts = [chunk["text"] for chunk in chunks]  
embeddings = embedding_service.embed(chunk_texts)
```

3. Create Pinecone index for this file

```
index_name = pinecone_service.get_index_name_for_file(file_id, source_filename)  
pinecone_service.create_index_for_file(file_id, source_filename)
```

4. Index chunks in Pinecone

```
pinecone_service.index_file_chunks(  
    file_id=file_id,  
    file_name=source_filename,  
    chunks=chunks,  
    embeddings=embeddings  
)
```

5. Update database

```
db = SessionLocal()  
  
file = db.query(UploadedFile).filter(UploadedFile.id == file_id).first()  
  
if file:  
    file.indexing_status = "indexed"  
    db.commit()  
  
except Exception as e:
```

Update status to error

```
db = SessionLocal()  
  
file = db.query(UploadedFile).filter(UploadedFile.id == file_id).first()  
  
if file:  
    file.indexing_status = "error"  
    db.commit()  
  
**Indexing Status:**  
- `pending_index`: File uploaded, waiting to be indexed  
- `indexed`: Successfully indexed in Pinecone  
- `error`: Indexing failed
```

7.4 Embeddings & Pinecone Feature

7.4.1 Overview

Embeddings convert text into numerical vectors that capture semantic meaning. Pinecone is a vector database that stores these embeddings and enables fast similarity search. This allows the chatbot to find relevant document sections based on question meaning, not just keyword matching.

7.4.2 Embeddings Explained

What are Embeddings?

- Numerical representation of text meaning
- Array of numbers (e.g., 384 dimensions)
- Similar meanings = similar vectors
- Can calculate "distance" between vectors to measure similarity

Example:

Text: "The project timeline is Q2 2025"

↓

Embedding Model (all-MiniLM-L6-v2)

↓

Vector: [0.123, -0.456, 0.789, ..., 0.234] (384 numbers)

Text: "When does the project finish?"

↓

Vector: [0.125, -0.451, 0.792, ..., 0.231] (very similar!)

Text: "What is the weather today?"

↓

Vector: [0.891, 0.234, -0.567, ..., -0.123] (very different!)

Embedding Model:

- **Model:** `all-MiniLM-L6-v2` (Sentence Transformers)
- **Dimensions:** 384
- **Provider:** Local (runs on server, no API calls)
- **Alternative:** Can use OpenAI or Vertex AI (configured via env vars)

7.4.3 Embedding Service

Location: `backend/services/embedding_service.py`

Key Methods:

Generate embeddings for multiple texts

```
embeddings = embedding_service.embed([
    "Text chunk 1",
    "Text chunk 2",
    "Text chunk 3"
])
```

**Returns: [[0.1, -0.2, ...], [0.3, 0.1, ...],
[0.2, -0.1, ...]]**

Generate embedding for single query

```
query_embedding = embedding_service.embed_query("What is the timeline?")
```

Returns: [0.123, -0.456, 0.789, ...]

****How It Works:****

1. Loads Sentence Transformers model (lazy loading)
2. Encodes text into numerical vectors
3. Returns list of vectors (one per text input)

7.4.4 Pinecone Vector Database

****What is Pinecone?****

- Managed vector database service
- Stores embeddings with metadata
- Provides fast similarity search
- Serverless architecture (AWS)

****Index Structure:****

- One index per file: `kb-file-{file_id}-{sanitized_filename}`
- Each index contains chunks from that file
- Index name format: lowercase, alphanumeric + hyphens only

****Example Index Names:****

```
kb-file-1-project-management-playbook  
kb-file-2-project-plan-2025  
kb-file-3-risk-assessment-template
```

7.4.5 Pinecone Operations

****Creating an Index:****

pinecone_service.py

```
def create_index_for_file(file_id, file_name, dimension=384):
```

```
index_name = self._get_index_name(file_id, file_name)

client = Pinecone(api_key=self.api_key)
client.create_index(
    name=index_name,
    dimension=384, # Must match embedding dimension
    metric="cosine", # Similarity metric
    spec=ServerlessSpec(
        cloud="aws",
        region="us-east-1"
    )
)
```

Indexing Chunks:

pinecone_service.py

```
def index_file_chunks(file_id, file_name, chunks, embeddings):
    index_name = self._get_index_name(file_id, file_name)
    index = client.Index(index_name)

    vectors = []
    for i, (chunk, embedding) in enumerate(zip(chunks, embeddings)):
        vectors.append({
            "id": f"chunk_{file_id}_{i}",
            "values": embedding, # 384-dim vector
            "metadata": {
                "file_id": str(file_id),
                "file_name": file_name,
                "chunk_index": str(i),
                "text": chunk["text"] # Store full chunk text
            }
        })
    return index
```

Upsert in batches (100 vectors per request)

```
index.upsert(vectors=vectors)
```

Searching:

pinecone_service.py

```

def search_across_indexes(query_embedding, index_names, top_k=5):
    all_results = []

    for index_name in index_names:
        index = client.Index(index_name)
        results = index.query(
            vector=query_embedding, # Question embedding
            top_k=top_k, # Get top 5 most similar
            include_metadata=True
        )

        for match in results["matches"]:
            all_results.append({
                "score": match["score"], # Similarity score (0.0-1.0)
                "text": match["metadata"]["text"], # Chunk text
                "file_name": match["metadata"]["file_name"],
                "chunk_id": match["id"]
            })

```

Sort by score (highest first)

```

all_results.sort(key=lambda x: x["score"], reverse=True)
return all_results

```

7.4.6 Similarity Search Flow

****Step 1: User Asks Question****

Question: "What is the project timeline?"

****Step 2: Convert Question to Embedding****

```
query_embedding = embedding_service.embed_query("What is the project timeline?")
```

Returns: [0.123, -0.456, 0.789, ..., 0.234] (384 numbers)

****Step 3: Search Pinecone****

Get file indexes to search

```

index_names = [
    "kb-file-1-project-plan",

```

```
"kb-file-2-timeline-doc"
```

```
]
```

Search all indexes

```
results = pinecone_service.search_across_indexes(  
    query_embedding=query_embedding,  
    index_names=index_names,  
    top_k=5 # Get top 5 chunks per file  
)
```

Step 4: Pinecone Returns Similar Chunks

```
results = [  
{  
    "score": 0.95, # Very similar!  
    "text": "The project timeline is Q2 2025, starting in April...",  
    "file_name": "project-plan.pdf",  
    "chunk_id": "chunk_1_5"  
,  
{  
    "score": 0.87,  
    "text": "Timeline overview: Q2 2025 delivery...",  
    "file_name": "project-plan.pdf",  
    "chunk_id": "chunk_1_12"  
,  
{  
    "score": 0.23, # Not very relevant  
    "text": "The team has 5 members...",  
    "file_name": "project-plan.pdf",  
    "chunk_id": "chunk_1_3"  
}]
```

Step 5: Use Top Chunks as Context

Take top 10 chunks (highest scores)

```
top_chunks = sorted(results, key=lambda x: x["score"], reverse=True)[:10]
```

Build context string

```
context = ""  
for chunk in top_chunks:  
    context += f"From {chunk['file_name']}:\n{chunk['text']}\n\n"
```

Send to Gemini with context

```
messages = [  
    {"role": "system", "content": "Answer based on context"},  
    {"role": "user", "content": f"Context:\n{context}\n\nQuestion: {question}"},  
]
```

7.4.7 Similarity Scores Explained

Cosine Similarity:

- Measures angle between two vectors
- Range: 0.0 (completely different) to 1.0 (identical)
- Higher score = more similar meaning

Score Interpretation:

- **0.8 - 1.0:** Very relevant, high confidence
- **0.6 - 0.8:** Relevant, medium confidence
- **0.4 - 0.6:** Somewhat relevant, low confidence
- **0.0 - 0.4:** Not relevant, very low confidence

Example Scores:

Question: "What is the project timeline?"

Chunk: "The project timeline is Q2 2025"

Score: 0.95 ■ (Very relevant)

Chunk: "Timeline overview: Q2 2025 delivery"

Score: 0.87 ■ (Relevant)

Chunk: "The team has 5 members"

Score: 0.23 ■ (Not relevant)

7.5 Mandatory Files Feature

7.5.1 Overview

Mandatory files are system-wide template files (like playbooks, templates, standards) that can be used across all projects. Admin users can upload these files, and regular users can download them or mark them for use in their project knowledge base.

7.5.2 File Storage

****Storage Method:****

- Files are stored **in the database** (not on disk)
- `file_content` column stores binary data (LargeBinary)
- Allows centralized management
- No file system dependencies

****Database Schema:****

```
class MandatoryFile(Base):  
    id = Column(Integer, primary_key=True)  
    file_name = Column(String)  
    file_type = Column(String)  
    file_content = Column(LargeBinary) # Binary data in DB  
    file_size = Column(Integer)  
    uploaded_by = Column(String)  
    extracted_text = Column(Text) # For search  
    is_active = Column(Boolean) # Soft delete
```

7.5.3 Uploading Mandatory Files

****Admin-Only Operation:****

main.py - POST **/api/mandatory-files/upload**

```
@app.post("/api/mandatory-files/upload")  
async def upload_mandatory_file(  
    file: UploadFile = File(...),  
    uploaded_by: str = Form(None),  
    description: str = Form(None),  
    db: Session = Depends(get_db)  
):
```

Check if user is admin

(Implementation checks user.role or email list)

Read file content

```
file_content = await file.read()
```

Extract text (same as regular file upload)

```
extracted_text = extract_text_from_file(file_content, file_type)
```

Save to database

```
mandatory_file = MandatoryFile(  
    file_name=file.filename,  
    file_type=file_extension,  
    file_content=file_content, # Store in DB  
    file_size=len(file_content),  
    uploaded_by=uploaded_by,  
    extracted_text=extracted_text,  
    is_active=True  
)  
db.add(mandatory_file)  
db.commit()
```

Index in Pinecone (background task)

...

7.5.4 Downloading Mandatory Files

Endpoint: `GET /api/mandatory-files/{file_id}/download`

Process:

main.py

```
@app.get("/api/mandatory-files/{file_id}/download")  
async def download_mandatory_file(file_id: int, db: Session = Depends(get_db)):
```

```

file = db.query(MandatoryFile).filter(
    MandatoryFile.id == file_id,
    MandatoryFile.is_active == True
).first()

if not file:
    return {"error": "File not found"}

```

Return binary content from database

```

return Response(
    content=file.file_content,
    media_type="application/octet-stream",
    headers={
        "Content-Disposition": f'attachment; filename="{file.file_name}"'
    }
)

```

7.5.5 Project Knowledge Base

****What is Project Knowledge Base?****

- User can mark mandatory files for use in their projects
- These files are automatically included in chatbot context
- Stored in `project_knowledge_base_files` table

****Marking Files for Project Use:****

```

// Frontend - handleToggleFileForProject()

const handleToggleFileForProject = async (fileId, fileName) => {
    const isMarked = filesMarkedForProject.has(fileId);

    if (isMarked) {
        // Remove from knowledge base
        await fetch(`#${apiUrl}/api/project-knowledge-base/remove?user_email=${user.email}&mandatory_file_id=${fileId}`, {
            method: 'DELETE'
        });
        setFilesMarkedForProject(prev => {
            const newSet = new Set(prev);
            newSet.delete(fileId);
            return newSet;
        });
    } else {

```

```

// Add to knowledge base
const formData = new FormData();
formData.append('user_email', user.email);
formData.append('mandatory_file_id', fileId);

await fetch(`/${apiUrl}/api/project-knowledge-base/add`, {
method: 'POST',
body: formData
});

setFilesMarkedForProject(prev => new Set([...prev, fileId]));
}
};

**Using in Chatbot:**

```

- When user sends message, `playbook_file_ids` are included
- Backend retrieves these mandatory files
- Files are searched in Pinecone for relevant chunks
- Chunks are included in context sent to Gemini

7.5.6 Mandatory Files UI

****Dropdown Menu:****

- Located in chat header (right side)
- Shows "Mandatory Files" button
- Clicking opens dropdown with file list

****File List Items:****

- File name (clickable to download)
- "Select Module" dropdown (admin only)
- "Use for Project" checkbox
- Download button
- Delete button (admin only)

****Admin Features:****

- Upload new files
- Delete files (soft delete)
- Set module name
- Refresh file list

8. Deployment

8.1 Local Development Setup

8.1.1 Prerequisites

Required Software:

1. **Python 3.8+**

- Download: <https://www.python.org/downloads/>
- Verify: `python --version` or `python3 --version`
- Windows: Add Python to PATH during installation

2. **Node.js 16+**

- Download: <https://nodejs.org/>
- Verify: `node --version`
- Includes npm (Node Package Manager)

3. **PostgreSQL 12+**

- Download: <https://www.postgresql.org/download/>
- Windows: Install as service
- Mac: `brew install postgresql` then `brew services start postgresql`
- Linux: `sudo apt-get install postgresql` then `sudo systemctl start postgresql`

4. **Git** (optional, for version control)

- Download: <https://git-scm.com/>

8.1.2 Database Setup

Step 1: Create Database

Connect to PostgreSQL

```
psql -U postgres
```

Create database

```
CREATE DATABASE sprint_demo;
```

Verify creation

\

Exit psql

\q

Step 2: Initialize Schema

Run setup script (if exists)

```
psql -U postgres -d sprint_demo -f database/setup.sql
```

Or let the application create tables automatically

(SQLAlchemy will create tables on first run)

Step 3: Verify Database Connection

Test connection

```
psql -U postgres -d sprint_demo -c "SELECT version();"
```

8.1.3 Backend Setup

Step 1: Navigate to Backend Directory

```
cd backend
```

Step 2: Create Virtual Environment

Windows

```
python -m venv venv
```

Mac/Linux

```
python3 -m venv venv
```

****Step 3: Activate Virtual Environment****

Windows

venv\Scripts\activate

Mac/Linux

source venv/bin/activate

You should see (venv) in your terminal prompt

****Step 4: Install Dependencies****

pip install -r requirements.txt

****Key Dependencies:****

- `fastapi` - Web framework
- `uvicorn` - ASGI server
- `sqlalchemy` - ORM
- `psycopg2-binary` - PostgreSQL driver
- `sentence-transformers` - Embeddings
- `pinecone-client` - Vector database
- `python-docx` - DOCX processing
- `pdfplumber` - PDF processing
- `openpyxl` - Excel processing

****Step 5: Configure Environment Variables****

Copy template

cp env.example .env

Edit .env file

Windows: notepad .env

Mac/Linux: nano .env or vim .env

****Required .env Variables:****

Database

DATABASE_URL=postgresql://postgres:your_password@localhost:5432/sprint_demo

Security

SECRET_KEY=your-secret-key-here-change-in-production

CORS (allow frontend to connect)

CORS_ORIGINS=http://localhost:3000,http://192.168.11.101:3000

Environment

ENVIRONMENT=development

Gemini API (at least one key required)

GEMINI_API_KEY_1=your-primary-gemini-api-key

GEMINI_API_KEY_2=your-secondary-gemini-api-key # Optional

GEMINI_API_KEY_3=your-tertiary-gemini-api-key # Optional

Pinecone (required for vector search)

PINECONE_API_KEY=your-pinecone-api-key

Google OAuth (optional)

GOOGLE_CLIENT_ID=your-google-client-id

GOOGLE_CLIENT_SECRET=your-google-client-secret

GOOGLE_REDIRECT_URI=http://localhost:3000/callback

Embedding Provider (default: local)

```
EMBEDDING_PROVIDER=local # Options: local, openai, vertex  
EMBEDDING_MODEL_NAME=all-MiniLM-L6-v2
```

Step 6: Start Backend Server

Development mode (with auto-reload)

```
uvicorn main:app --reload --host 0.0.0.0 --port 8000
```

Production mode (no reload)

```
uvicorn main:app --host 0.0.0.0 --port 8000
```

Verify Backend:

- Open: <http://localhost:8000>
- Should see: `{"message": "PM Portal Bot API", "status": "running"}`
- API Docs: <http://localhost:8000/docs>

8.1.4 Frontend Setup

Step 1: Navigate to Frontend Directory

In a new terminal window

```
cd frontend
```

Step 2: Install Dependencies

```
npm install
```

This installs:

- React 18.2.0
- React Router DOM
- Axios
- XLSX
- And other dependencies from `package.json`

Step 3: Configure Environment Variables (Optional)

Create .env file in frontend directory

```
echo "REACT_APP_API_URL=http://localhost:8000" > .env
```

Step 4: Start Frontend Server

```
npm start
```

This will:

- Start development server on http://localhost:3000
- Open browser automatically
- Enable hot-reload (changes reflect immediately)

Verify Frontend:

- Open: http://localhost:3000
- Should see landing page with login options

8.1.5 Using Quick Start Scripts

Windows Users:

Double-click start.bat or run in Command Prompt

start.bat

What it does:

1. Checks for Python and Node.js
2. Creates virtual environment if needed
3. Installs backend dependencies
4. Creates .env from template if missing
5. Starts backend server in new window
6. Installs frontend dependencies
7. Starts frontend server in new window

Mac/Linux Users:

Make executable

```
chmod +x start.sh
```

Run script

```
./start.sh
```

What it does:

1. Checks prerequisites
2. Sets up backend (venv, dependencies, .env)
3. Starts backend in background

4. Sets up frontend (dependencies)
5. Starts frontend in background
6. Shows access URLs

Stopping Servers:

- Windows: Close the command windows
- Mac/Linux: Press `Ctrl+C` in terminal

8.2 Production Deployment

8.2.1 Production Environment Variables

Backend .env (Production):

Database (use production database)

DATABASE_URL=postgresql://prod_user:secure_password@prod_host:5432/pm_portal_prod

Security (use strong, unique secret)

SECRET_KEY=very-secure-production-secret-key-min-32-chars

CORS (only allow production domain)

CORS_ORIGINS=https://yourdomain.com,https://www.yourdomain.com

Environment

ENVIRONMENT=production

API Keys (use production keys)

GEMINI_API_KEY_1=prod-gemini-key-1

PINECONE_API_KEY=prod-pinecone-key

Google OAuth (production redirect URI)

```
GOOGLE_REDIRECT_URI=https://yourdomain.com/callback
```

Frontend .env (Production):

```
REACT_APP_API_URL=https://api.yourdomain.com
```

8.2.2 Backend Production Deployment

Option 1: Using Uvicorn with Gunicorn

Install gunicorn

```
pip install gunicorn
```

Run with multiple workers

```
gunicorn main:app -w 4 -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:8000
```

Option 2: Using Systemd Service (Linux)

Create service file

```
sudo nano /etc/systemd/system/pm-portal-backend.service
```

Service File Content:

[Unit]

Description=PM Portal Backend API

After=network.target

[Service]

User=www-data

WorkingDirectory=/var/www/pm-portal/backend

Environment="PATH=/var/www/pm-portal/backend/venv/bin"

ExecStart=/var/www/pm-portal/backend/venv/bin/uvicorn main:app --host 0.0.0.0 --port 8000

Restart=always

[Install]

WantedBy=multi-user.target

Enable and Start:

```
sudo systemctl enable pm-portal-backend
```

```
sudo systemctl start pm-portal-backend
```

```
sudo systemctl status pm-portal-backend
```

Option 3: Using Docker

Dockerfile

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

****Build and Run:****

```
docker build -t pm-portal-backend .
docker run -p 8000:8000 --env-file .env pm-portal-backend
```

8.2.3 Frontend Production Build

****Step 1: Build Production Bundle****

```
cd frontend
npm run build
```

****This creates:****

- Optimized production build in `frontend/build/`
- Minified JavaScript and CSS
- Static assets ready for deployment

****Step 2: Serve with Nginx****

/etc/nginx/sites-available/pm-portal

```
server {
listen 80;
server_name yourdomain.com;
```

Frontend

```
location / {
root /var/www/pm-portal/frontend/build;
try_files $uri $uri/ /index.html;
}
```

Backend API

```
location /api {  
    proxy_pass http://localhost:8000;  
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
}  
}
```

Step 3: Enable HTTPS (Required for Production)

Install Certbot

```
sudo apt-get install certbot python3-certbot-nginx
```

Get SSL certificate

```
sudo certbot --nginx -d yourdomain.com
```

8.2.4 Database Production Setup

Step 1: Create Production Database

```
psql -U postgres  
CREATE DATABASE pm_portal_prod;  
CREATE USER prod_user WITH PASSWORD 'secure_password';  
GRANT ALL PRIVILEGES ON DATABASE pm_portal_prod TO prod_user;  
\q
```

Step 2: Run Migrations

Connect and run setup

```
psql -U prod_user -d pm_portal_prod -f database/setup.sql
```

Step 3: Backup Strategy

Daily backup script

```
pg_dump -U prod_user pm_portal_prod > backup_$(date +%Y%m%d).sql
```

Restore from backup

```
psql -U prod_user pm_portal_prod < backup_20250115.sql
```

8.3 Environment Variables Reference

8.3.1 Backend Environment Variables

Variable	Required	Description	Example
`DATABASE_URL`	Yes	PostgreSQL connection string	`postgresql://user:pass@localhost:5432/dbname`
`SECRET_KEY`	Yes	Secret key for sessions	`your-secret-key-here`
`CORS_ORIGINS`	Yes	Allowed frontend origins	`http://localhost:3000`
`ENVIRONMENT`	No	Environment name	`development` or `production`
`GEMINI_API_KEY_1`	Yes	Primary Gemini API key	`Alza...`
`GEMINI_API_KEY_2`	No	Secondary key (fallback)	`Alza...`
`GEMINI_API_KEY_3`	No	Tertiary key (fallback)	`Alza...`
`PINECONE_API_KEY`	Yes	Pinecone API key	`your-pinecone-key`
`GOOGLE_CLIENT_ID`	No	Google OAuth client ID	`123...apps.googleusercontent.com`
`GOOGLE_CLIENT_SECRET`	No	Google OAuth secret	`GOCSPX-...`
`GOOGLE_REDIRECT_URI`	No	OAuth redirect URI	`http://localhost:3000/callback`
`EMBEDDING_PROVIDER`	No	Embedding service	`local`, `openai`, `vertex`
`EMBEDDING_MODEL_NAME`	No	Embedding model	`all-MiniLM-L6-v2`

8.3.2 Frontend Environment Variables

Variable	Required	Description	Example
`REACT_APP_API_URL`	Yes	Backend API URL	`http://localhost:8000`

****Note:**** Frontend env variables must start with `REACT_APP_` to be accessible in code.

8.4 Deployment Checklist

****Pre-Deployment:****

- [] All environment variables configured
- [] Database created and accessible
- [] API keys obtained (Gemini, Pinecone, Google OAuth)
- [] Dependencies installed
- [] Database migrations run
- [] SSL certificate obtained (production)

****Backend Deployment:****

- [] Virtual environment created
- [] Dependencies installed
- [] .env file configured
- [] Server starts without errors
- [] Health check endpoint responds
- [] Database connection works
- [] API documentation accessible

****Frontend Deployment:****

- [] Dependencies installed
- [] .env file configured
- [] Production build created (`npm run build`)
- [] Build files served correctly
- [] API connection works
- [] No console errors

****Post-Deployment:****

- [] Test user login
- [] Test file upload
- [] Test chatbot functionality
- [] Test project creation
- [] Monitor logs for errors
- [] Set up monitoring/alerting

9. Troubleshooting

9.1 Common Errors and Solutions

9.1.1 Database Connection Errors

****Error:**** `psycopg2.OperationalError: connection to server failed`

****Causes:****

- PostgreSQL service not running
- Wrong database credentials
- Database doesn't exist
- Firewall blocking connection

****Solutions:****

1. Check if PostgreSQL is running

**Windows: Check Services
(services.msc)**

Mac: brew services list

**Linux: sudo systemctl status
postgresql**

2. Start PostgreSQL if not running

Windows: Start PostgreSQL service

Mac: brew services start postgresql

**Linux: sudo systemctl start
postgresql**

3. Verify database exists

`psql -U postgres -`

4. Test connection

`psql -U postgres -d sprint_demo -c "SELECT 1;"`

5. Check .env file

Ensure DATABASE_URL is correct:

`postgresql://username:password@localhost:5432/database_name`

****Error:**** `relation "users" does not exist`

****Cause:**** Database tables not created

****Solution:****

Option 1: Let SQLAlchemy create tables (automatic on first run)

Just start the backend server

Option 2: Run setup script manually

```
psql -U postgres -d sprint_demo -f database/setup.sql
```

Option 3: Use db_migrations.py

```
cd backend
```

```
python db_migrations.py
```

```
#### 9.1.2 CORS Errors
```

****Error:**** `Access to fetch at 'http://localhost:8000' from origin 'http://localhost:3000' has been blocked by CORS policy`

****Cause:**** Backend not allowing frontend origin

****Solution:****

1. Check backend .env file

```
CORS_ORIGINS=http://localhost:3000,http://192.168.11.101:3000
```

2. Restart backend server after changing .env

3. Verify in main.py:

```
app.add_middleware(  
    CORSMiddleware,  
    allow_origins=os.getenv("CORS_ORIGINS", "...").split(", ")-  
    allow_credentials=True,  
    allow_methods=["*"],  
    allow_headers=["*"],  
)
```

9.1.3 Port Already in Use

****Error:** `Address already in use` or `Port 8000 is already in use`**

****Solution:****

Find process using port

Windows:

```
netstat -ano | findstr :8000  
taskkill /PID <PID> /F
```

Mac/Linux:

```
lsof -ti:8000 | xargs kill
```

Or use different port

```
uvicorn main:app --port 8001
```

9.1.4 Module Not Found Errors

****Error:** `ModuleNotFoundError: No module named 'fastapi'`**

****Cause:**** Virtual environment not activated or dependencies not installed

****Solution:****

1. Activate virtual environment

Windows:

```
venv\Scripts\activate
```

Mac/Linux:

```
source venv/bin/activate
```

2. Verify activation (should see (venv) in prompt)

3. Install dependencies

```
pip install -r requirements.txt
```

4. Verify installation

```
pip list | grep fastapi
```

9.1.5 Gemini API Errors

****Error:**** `Gemini API error: API key not valid`

****Solution:****

1. Check .env file has GEMINI_API_KEY_1 set

2. Verify key is correct (no extra spaces)

3. Check key hasn't expired

4. Try using fallback keys (GEMINI_API_KEY_2, GEMINI_API_KEY_3)

****Error:**** `Rate limit exceeded`

****Solution:****

- System automatically switches to next API key
- Add more API keys to .env (GEMINI_API_KEY_2, GEMINI_API_KEY_3)
- Wait for rate limit to reset
- Check API quota in Google Cloud Console

9.1.6 Pinecone Errors

****Error:**** `PINECONE_API_KEY not set`

****Solution:****

1. Get API key from Pinecone dashboard

2. Add to .env:

PINECONE_API_KEY=your-pinecone-api-key

3. Restart backend server

****Error:**** `Index not found`

****Cause:**** File not indexed yet or index was deleted

****Solution:****

1. Check file indexing status in database

2. Reindex file manually via API

POST /api/project-knowledge-base/reindex-all

3. Wait for background indexing to complete

Check indexing_status in uploaded_files table

9.1.7 File Upload Errors

****Error:**** `Invalid file type`

****Cause:**** File extension not supported

****Solution:****

- Supported types: PDF, DOCX, TXT, XLSX
- Check file extension is lowercase
- Ensure file is not corrupted

****Error:**** `File too large`

****Cause:**** File exceeds size limit

****Solution:****

- Current limit: No explicit limit (depends on server memory)
- For very large files (>50MB), consider splitting
- Check server memory and disk space

****Error:**** `Failed to extract text from PDF`

****Cause:**** PDF is image-based or corrupted

****Solution:****

- Try OCR for image-based PDFs
- Verify PDF is not password-protected
- Check if PDF is corrupted
- Try converting PDF to text manually

9.1.8 Authentication Errors

****Error:**** `Google OAuth not configured`

****Solution:****

1. Set Google OAuth credentials in .env

GOOGLE_CLIENT_ID=your-client-id

GOOGLE_CLIENT_SECRET=your-client-secret

GOOGLE_REDIRECT_URI=http://localhost:3000/callback

2. Verify redirect URI matches Google Cloud Console

3. Restart backend server

****Error:**** `Access restricted to @forsysinc.com email addresses only`

****Cause:**** Email login restricted to company domain

****Solution:****

- Use @forsysinc.com email for email login
- Or use Google OAuth with any Google account
- Contact admin to add your email to allowed list

9.1.9 Frontend Build Errors

****Error:**** `npm ERR! peer dep missing`

****Solution:****

1. Clear npm cache

npm cache clean --force

2. Delete node_modules and package-lock.json

rm -rf node_modules package-lock.json

3. Reinstall

npm install

Error: `Module not found: Can't resolve './components/HomePage'`

Cause: File path incorrect or file missing

Solution:

- Check file exists at specified path
- Verify import path is correct
- Check for typos in file names
- Ensure file extension is included (.js)

9.2 Debugging Tips

9.2.1 Backend Debugging

Enable Debug Logging:

In main.py, add logging

```
import logging  
logging.basicConfig(level=logging.DEBUG)
```

Check Backend Logs:

- Terminal where uvicorn is running shows all logs
- Look for error messages and stack traces
- Check database query logs

Test API Endpoints:

Health check

curl http://localhost:8000/

Test chat endpoint

```
curl -X POST http://localhost:8000/api/chat \  
-F "user_message=Hello" \  
-H "Content-Type: application/x-www-form-urlencoded"
```

```
-F "chat_id=test123" \
-F "user_email=test@example.com"
```

View API documentation

Open <http://localhost:8000/docs> in browser

****Database Debugging:****

Connect to database

```
psql -U postgres -d sprint_demo
```

Check tables

```
\dt
```

Check data

```
SELECT * FROM users LIMIT 5;
SELECT * FROM conversations LIMIT 5;
```

Check file indexing status

```
SELECT id, file_name, indexing_status FROM uploaded_files;
```

9.2.2 Frontend Debugging

****Browser Console:****

- Press F12 to open Developer Tools
- Check Console tab for errors
- Check Network tab for failed API calls

****React DevTools:****

- Install React DevTools browser extension
- Inspect component state and props
- Check for state update issues

```
**Check API Calls:**  
// In browser console  
fetch('http://localhost:8000/api/chat/sessions?user_email=test@example.com')  
.then(r => r.json())  
.then(console.log);
```

```
**Check LocalStorage:**  
// In browser console  
console.log(localStorage.getItem('user'));  
console.log(localStorage.getItem('sessionId'));
```

9.2.3 Common Issues

Issue: Chatbot not responding

Debug Steps:

1. Check browser console for errors
2. Check backend logs for API errors
3. Verify Gemini API key is set
4. Check network tab for failed requests
5. Verify CORS is configured correctly

Issue: Files not indexing

Debug Steps:

1. Check `indexing_status` in database
2. Check backend logs for indexing errors
3. Verify Pinecone API key is set
4. Check file has extracted text
5. Manually trigger reindex

Issue: Projects not showing

Debug Steps:

1. Check user_email is correct
2. Verify projects exist in database
3. Check API response in Network tab
4. Verify user is logged in
5. Check browser console for errors

9.3 Performance Optimization

9.3.1 Backend Optimization

Database Connection Pooling:

In models.py, configure engine

```
engine = create_engine(  
    DATABASE_URL,  
    pool_size=10,  
    max_overflow=20,  
    pool_pre_ping=True  
)
```

Caching:

- Cache frequently accessed data
- Use Redis for session storage (optional)
- Cache embedding results for repeated queries

Async Operations:

- Use background tasks for file indexing
- Async database queries where possible
- Parallel processing for multiple files

9.3.2 Frontend Optimization

Code Splitting:

```
// Lazy load components  
const HomePage = React.lazy(() => import('./components/HomePage'));  
  
// Use Suspense  
<Suspense fallback={<div>Loading...</div>}>  
<HomePage />  
</Suspense>
```

Memoization:

```
// Memoize expensive computations  
const expensiveValue = useMemo(() => {  
  return computeExpensiveValue(data);  
}, [data]);
```

Production Build:

Always use production build for deployment

```
npm run build
```

Production build is optimized:

- Minified code
- Tree shaking
- Code splitting

9.4 Getting Help

9.4.1 Log Files

Backend Logs:

- Terminal output where unicorn runs
- Check for error messages and stack traces
- Look for `[ERROR]` or `[WARNING]` prefixes

Frontend Logs:

- Browser console (F12)
- Network tab for API calls
- Application tab for localStorage

9.4.2 Useful Commands

Check System Status:

Check Python version

```
python --version
```

Check Node version

```
node --version
```

Check PostgreSQL version

```
psql --version
```

Check if ports are in use

Windows:

```
netstat -ano | findstr :8000
```

Mac/Linux:

```
lsof -i :8000
```

Database Commands:

Connect to database

```
psql -U postgres -d sprint_demo
```

List all tables

```
\dt
```

Describe table structure

```
\d users
```

Count records

```
SELECT COUNT(*) FROM users;
```

Check recent uploads

```
SELECT id, file_name, indexing_status, upload_time  
FROM uploaded_files  
ORDER BY upload_time DESC  
LIMIT 10;
```

Backend Commands:

Activate virtual environment

```
source venv/bin/activate # Mac/Linux  
venv\Scripts\activate # Windows
```

Install dependencies

```
pip install -r requirements.txt
```

Check installed packages

```
pip list
```

Run with debug logging

```
uvicorn main:app --reload --log-level debug
```

Frontend Commands:

Install dependencies

```
npm install
```

Check installed packages

```
npm list
```

Clear cache and reinstall

```
rm -rf node_modules package-lock.json  
npm install
```

Build for production

```
npm run build
```

10. Conclusion

10.1 Summary

PM Portal is a comprehensive AI-powered project management chatbot that combines:

- **Modern Tech Stack:** React.js frontend, FastAPI backend, PostgreSQL database
- **AI Integration:** Google Gemini for intelligent responses
- **Vector Search:** Pinecone for semantic document search
- **File Processing:** Support for PDF, DOCX, XLSX, TXT files
- **Project Management:** Organize conversations and files by project
- **Authentication:** Google OAuth and email-based login

10.2 Key Features Recap

1. **Chatbot Interface:** Main landing page with AI-powered Q&A
2. **File Upload:** Upload and process documents for chatbot context
3. **Vector Search:** Find relevant document sections using embeddings
4. **Project Management:** Organize work into projects with conversations
5. **Mandatory Files:** System-wide templates accessible to all users
6. **Authentication:** Secure login with Google OAuth or email

10.3 Architecture Highlights

- **Three-Tier Architecture:** Frontend → Backend → Database
- **Microservices Pattern:** Separate services for embeddings, Pinecone, file processing
- **Background Processing:** Async file indexing for better performance
- **Session-Based Auth:** Simple, effective authentication without JWT complexity

10.4 Next Steps

For Developers:

- Review code structure in `backend/` and `frontend/src/`
- Understand API endpoints in `backend/main.py`
- Explore database models in `backend/models.py`
- Customize features as needed

****For Deployment:****

- Follow deployment guide in Section 8
- Configure production environment variables
- Set up monitoring and logging
- Plan backup strategy

****For Users:****

- Start with chatbot interface
- Upload relevant documents
- Create projects to organize work
- Use mandatory files for templates

10.5 Support Resources

- **API Documentation:** <http://localhost:8000/docs> (when server running)
- **Code Repository:** Check project README.md
- **Troubleshooting:** See Section 9 for common issues
- **Database Schema:** See Section 4 for table structures

****END OF DOCUMENTATION****

This completes the complete Technical & Functional Documentation for PM Portal. The documentation covers all aspects of the project from overview to deployment and troubleshooting. Total length: ~4,000+ lines covering every feature, component, and process in detail.

****Document Version:**** 1.0.0

****Last Updated:**** 2025

****Total Sections:**** 10

****Total Pages:**** ~80-100 pages (when converted to PDF)