Project 6: Image-Generating Chatbot

Objective (Why?)

Build an intelligent image-generating chatbot that creates high-quality images from natural language descriptions. This project introduces multimodal AI capabilities to your existing chat platform from Project 5. You will practice:

- Multimodal AI Integration: Combining text processing with image generation APIs
- Direct API Communication: Simple request-response pattern with LLM image generation
- Image Storage & Management: Using database storage for generated images within the application architecture
- Real-time UI Feedback: Loading states and immediate image display

Core Requirements (Must-have)

Component	Requirement
Image Generation	Integrate OpenAI DALL·E 3 for direct text-to-image conversion with immediate response
Simple Processing	Direct API calls to DALL-E with loading states - no background processing needed
Database Storage	Store generated images as Base64 strings in PostgreSQL database with metadata
Chat Integration	Seamlessly integrate image generation into existing chat platform from Project 5
Gallery Management	Store, organize, and retrieve generated images with search and download functionality

Milestone 1: Direct Image Generation & Storage

Deliverables:

- DALL-E 3 API integration with secure key management
- Direct API call implementation with proper error handling
- Image metadata and Base64 content schema in database
- Basic image generation endpoint with loading states

Review Requirements (Must Pass to Proceed):

- Security Review: API key security, input validation, rate limiting
- Architecture Review: Clean direct API integration
- Performance Review: Efficient image generation and storage

Milestone 2: Chat Integration & UI Enhancement

Deliverables:

- Enhanced chat interface with image generation commands
- Loading states and progress indicators during API calls
- Image display and gallery components
- Integration with existing Project 5 chat system
- Error handling and user feedback

Review Requirements (Must Pass to Proceed):

- AI Integration Review: Seamless image generation within conversations
- Architecture Review: Clean chat and image generation integration
- Performance Review: Responsive UI during API calls

Milestone 3: Production Features & Creative Tools

Deliverables:

- Advanced image management (organize, search, download)
- Creative prompt assistance and generation history
- Rate limiting and cost management systems
- Performance optimization and caching strategies

- Comprehensive testing and production deployment

Review Requirements (Must Pass for Project Completion):

- AI Integration Review: Creative AI workflow optimization
- Architecture Review: Complete creative platform architecture
- Security Review: Production security and cost control
- Performance Review: Optimized image generation performance

Milestone Progression Rules:

- Cannot advance to next milestone without passing all review requirements
- Flexible timing allows for learning at individual pace
- Quality gates ensure each milestone meets professional standards
- Mentor support available for concept clarification and review failures

Technical Specifications

Simplified Database Schema

```sql
-- Image generation tables (simplified)
CREATE TABLE generated_images (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id) NOT NULL,
    thread_id INTEGER REFERENCES chat_threads(id),
    prompt_text TEXT NOT NULL,
    image_data TEXT NOT NULL,  -- Base64 encoded image
    thumbnail_data TEXT,       -- Base64 encoded thumbnail
    metadata JSONB DEFAULT '{}',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```sql
-- Simple image metadata tracking
CREATE TABLE image_metadata (
  id SERIAL PRIMARY KEY,
  image_id INTEGER REFERENCES generated_images(id) NOT NULL,
  original_prompt TEXT NOT NULL,
  revised_prompt TEXT,
  dalle_metadata JSONB,
  generation_time_ms INTEGER,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Indexes for performance
CREATE INDEX idx_images_user_id ON generated_images(user_id);
CREATE INDEX idx_images_thread_id ON generated_images(thread_id);
CREATE INDEX idx_images_created_at ON generated_images(created_at);
```

Simplified Image Processing Pipeline

Python
```python
# Direct API implementation (Python/FastAPI)
import base64
import httpx
from openai import OpenAI
from fastapi import FastAPI, HTTPException
from sqlalchemy.orm import Session

class SimpleImageGenerator:
  def __init__(self):
    self.openai_client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
```

```
async def generate_image(self, prompt: str, user_id: int, thread_id: int) -> dict:
    """Generate image directly via DALL-E API"""
    try:
        # Generate image with DALL-E 3
        response = self.openai_client.images.generate(
            model="dall-e-3",
            prompt=prompt,
            size="1024x1024",
            quality="standard",
            n=1
        )

        # Download image and convert to base64
        image_url = response.data[0].url
        async with httpx.AsyncClient() as client:
            image_response = await client.get(image_url)
            image_response.raise_for_status()

        image_base64 = base64.b64encode(image_response.content).decode('utf-8')

        # Store in database
        image_record = self._store_image(
            user_id=user_id,
            thread_id=thread_id,
            prompt=prompt,
            image_data=image_base64,
            revised_prompt=response.data[0].revised_prompt
        )

        return {
                "image_id": image_record.id,
                "image_data": f"data:image/png;base64,{image_base64}",
                "original_prompt": prompt,
                "revised_prompt": response.data[0].revised_prompt
            }

        except Exception as e:
            raise HTTPException(status_code=500, detail=f"Image generation failed: {str(e)}")

    def _store_image(self, user_id: int, thread_id: int, prompt: str,
                    image_data: str, revised_prompt: str) -> any:
        """Store image in database"""
        # Database storage implementation
        pass

# FastAPI endpoint
@app.post("/api/chat/generate-image")
async def generate_image_endpoint(request: ImageGenerationRequest):
    generator = SimpleImageGenerator()
    result = await generator.generate_image(
        prompt=request.prompt,
        user_id=request.user_id,
        thread_id=request.thread_id
    )
    return result
```

Frontend Implementation (React)

TypeScript
```
// Simplified React component
```

```
import React, { useState } from 'react';

interface ImageGenerationState {
  isGenerating: boolean;
  generatedImage: any | null;
  error: string | null;
}

const ImageGenerationComponent: React.FC = () => {
  const [state, setState] = useState<ImageGenerationState>({
    isGenerating: false,
    generatedImage: null,
    error: null
  });

  const generateImage = async (prompt: string) => {
    setState(prev => ({ ...prev, isGenerating: true, error: null }));

    try {
      const response = await fetch('/api/chat/generate-image', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          prompt,
          user_id: getCurrentUserId(),
          thread_id: getCurrentThreadId()
        })
      });

      if (!response.ok) {
        throw new Error('Image generation failed');
      }

      const result = await response.json();

      setState(prev => ({
        ...prev,
        isGenerating: false,
        generatedImage: result
      }));

    } catch (error) {
      setState(prev => ({
        ...prev,
        isGenerating: false,
        error: error.message
      }));
    }
  };

  return (
    <div className="image-generation-container">
      {state.isGenerating && (
        <div className="loading-state">
          <div className="spinner" />
          <p>Generating your image... This may take up to 30 seconds.</p>
        </div>
      )}

      {state.generatedImage && (
        <div className="generated-image">
          <img
            src={state.generatedImage.image_data}
            alt={state.generatedImage.revised_prompt}
            className="generated-image-display"
          />
          <div className="image-metadata">
            <p><strong>Original:</strong> {state.generatedImage.original_prompt}</p>
            <p><strong>Revised:</strong> {state.generatedImage.revised_prompt}</p>
          </div>
        </div>
      )}

      {state.error && (
        <div className="error-message">
          <p>Failed to generate image: {state.error}</p>
          <button onClick={() => setState(prev => ({ ...prev, error: null }))}>
            Try Again
          </button>
        </div>
      )}
    </div>
  );
};
```