Project 1 - Basic LLM Chatbot - Technical Implementation Guide

1. Project Overview & Learning Objectives

Business Context

Build a foundational LLM-powered chatbot that demonstrates core AI integration patterns, secure API management, and professional UI development. This project establishes essential skills for all subsequent AI applications.

Architecture Overview

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│  Streamlit UI   │───▶│ FastAPI Backend │───▶│  LLM Services   │
│                 │    │                 │    │                 │
│ • Chat Interface│    │ • API Routes    │    │ • OpenAI GPT-4  │
│ • Session Mgmt  │    │ • Validation    │    │ • Google Gemini │
│ • Error Display │    │ • Rate Limiting │    │ • Error Handling│
└─────────────────┘    └─────────────────┘    └─────────────────┘
                                │
                       ┌─────────────────┐
                       │  Configuration  │
                       │                 │
                       │ • Environment   │
                       │ • Security      │
                       │ • Logging       │
                       └─────────────────┘
```

Core Learning Goals

- API Integration: Professional LLM service integration patterns
- Security Management: Environment variables and API key security
- Error Handling: Comprehensive error management and user feedback
- UI Development: Interactive chat interface with state management
- Production Patterns: Logging, validation, and deployment readiness

2. Implementation Strategy & Copilot Integration

Development Approach

This foundational project introduces core patterns that will be used throughout all subsequent projects. Focus on establishing clean, secure, and maintainable code patterns.

Copilot Optimization Tips

- Specify frameworks (Streamlit, FastAPI, OpenAI) in prompts
- Include security requirements for API key management
- Request error handling patterns for production readiness
- Ask for validation logic and input sanitization

3. Milestone 1: Backend Foundation & API Integration

3.1 Project Structure Setup

Recommended Directory Structure

Copilot Prompt: *"Create a well-organized project structure for a Streamlit + FastAPI chatbot with folders for services, API routes, configuration, and frontend components. Include proper init.py files and separation of concerns."*

```
None
project_root/
├── src/
│   ├── api/
│   │   ├── __init__.py
│   │   ├── routes.py
│   │   └── models.py
│   ├── services/
│   │   ├── __init__.py
│   │   ├── llm_service.py
│   │   └── chat_service.py
│   ├── config/
│   │   ├── __init__.py
│   │   └── settings.py
│   └── utils/
│       ├── __init__.py
│       └── validators.py
├── frontend/
│   └── streamlit_app.py
├── tests/
├── requirements.txt
├── .env.example
└── README.md
```

3.2 Environment Configuration

Configuration Management Pattern

Copilot Prompt: *"Create a Pydantic settings class for managing OpenAI and Gemini API keys, application configuration, and validation rules. Include environment variable loading and secure defaults."*

Key Implementation Areas:

- Environment variable validation
- API key security patterns
- Configuration inheritance
- Default value management

Expected Components:

```python
class Settings(BaseSettings):

    # API Keys with validation

    # Application configuration

    # Security settings

    # Rate limiting configuration
```

3.3 LLM Service Integration

Multi-Provider LLM Service

Copilot Prompt: *"Build a flexible LLM service class that supports both OpenAI GPT-4 and Google Gemini APIs with error handling, timeout management, and response standardization. Include async support and proper logging."*

Core Patterns to Implement:

- Provider abstraction layer
- Unified response format
- Error recovery mechanisms
- Token usage tracking
- Performance monitoring

Service Architecture:

```python
class LLMService:

    def __init__(self):

        # Initialize multiple providers
```

```
async def generate_response(self, message, model="openai"):
    # Unified response generation
def get_available_models(self):
    # Dynamic model discovery
```

3.4 Chat Management Service

Conversation State Management

Copilot Prompt: *"Create a chat service that manages conversation sessions, message history, and user context. Include session creation, history retrieval, conversation clearing, and export functionality."*

Implementation Requirements:

- Session lifecycle management
- Message persistence patterns
- Context window management
- Conversation analytics
- Data export capabilities

4. Milestone 2: FastAPI Backend Development

4.1 API Model Definitions

Request/Response Models

Copilot Prompt: *"Design Pydantic models for chat requests, responses, conversation history, and error handling. Include comprehensive validation rules, field constraints, and documentation strings."*

Model Structure:

- ChatRequest: Input validation and constraints
- ChatResponse: Standardized output format
- ConversationHistory: Session data management
- ErrorResponse: Consistent error reporting

4.2 RESTful API Routes

Endpoint Implementation

Copilot Prompt: *"Build FastAPI routes for chat processing, conversation management, model selection, and health checks. Include proper HTTP status codes, error handling, and API documentation."*

Required Endpoints:

- POST /api/chat: Message processing
- GET /api/conversation/{session_id}: History retrieval
- DELETE /api/conversation/{session_id}: Session clearing
- GET /api/models: Available model list
- GET /api/health: System status

4.3 Middleware and Security

Production-Ready Middleware

Copilot Prompt: *"Implement CORS middleware, request validation, rate limiting, and security headers for a production FastAPI application. Include proper error handling and logging integration."*

Security Components:

- CORS configuration
- Input sanitization
- Rate limiting implementation

- Security headers
- Request logging

5. Milestone 3: Streamlit Frontend Development

5.1 Chat Interface Design

Interactive Chat Component

Copilot Prompt: *"Create a Streamlit chat interface with message history display, user input handling, model selection sidebar, and conversation management features. Include proper state management and error display."*

UI Requirements:

- Message thread display
- Real-time response rendering
- Model selection controls
- Session management UI
- Export functionality

5.2 State Management

Session State Patterns

Copilot Prompt: *"Implement Streamlit session state management for chat history, user settings, API connection status, and conversation persistence. Include proper initialization and cleanup."*

State Components:

- Message history persistence
- User preference storage
- API connection status
- Session ID management

- Error state handling

5.3 Advanced UI Features

Enhanced User Experience

Copilot Prompt: *"Add advanced Streamlit features including response streaming, typing indicators, message metadata display, conversation export, and responsive layout design."*

Feature Implementation:

- Progressive response display
- Response time metrics
- Token usage indicators
- Conversation statistics
- Mobile-responsive design

8. Extension Opportunities

Advanced Features

- Multi-turn Context: Enhanced conversation memory
- Custom Prompts: User-defined system prompts
- Response Streaming: Real-time response display
- Usage Analytics: Detailed conversation metrics
- Theme Customization: UI personalization options

Integration Possibilities

- Database Backend: Persistent conversation storage
- Authentication: User account management
- API Rate Limiting: Advanced usage controls
- Content Filtering: Automated content moderation
- Export Options: Multiple format support