Complete Source Code Documentation

Table of Contents

Project Overview

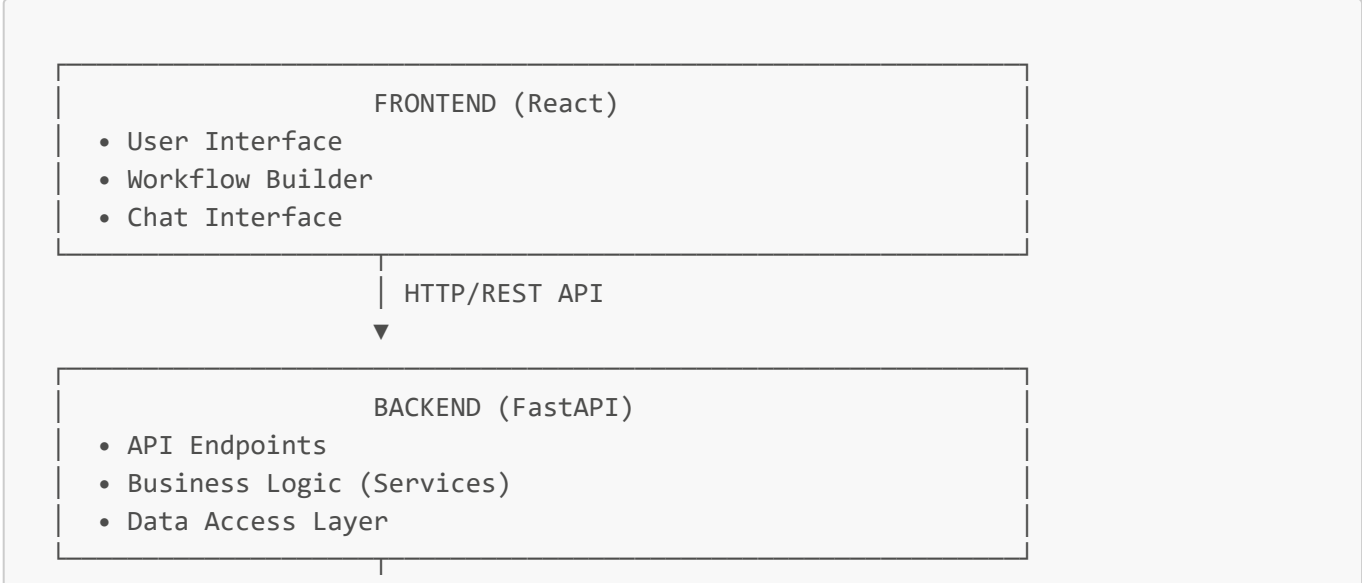
GenAI Stack is a full-stack No-Code/Low-Code workflow builder application that enables users to create AI-powered workflows through a visual drag-and-drop interface.

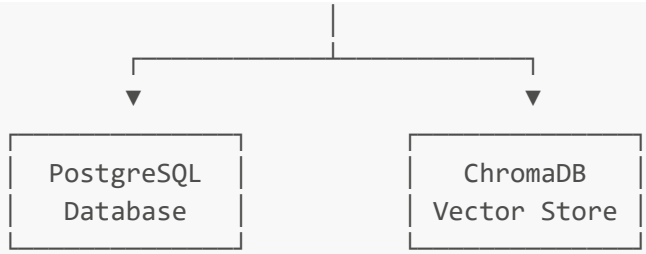
Core Technologies

- **Frontend**: React 18, Vite, ReactFlow, Axios
- **Backend**: FastAPI, Python 3.11, SQLAlchemy
- **Database**: PostgreSQL 15, ChromaDB
- **AI/ML**: Google Gemini API
- **DevOps**: Docker, Docker Compose

Code Architecture

High-Level Architecture





Design Principles

1. **Separation of Concerns**: Clear boundaries between layers
2. **Modularity**: Independent, reusable components
3. **Dependency Injection**: Loose coupling between modules
4. **RESTful API**: Standard HTTP methods and status codes
5. **Type Safety**: Pydantic schemas for validation

Backend Components

1. Application Entry Point

File: `backend/app/main.py`

Purpose: Initializes FastAPI application and configures middleware

Key Components:

```
app = FastAPI(
    title="Workflow Builder API",
    description="No-Code/Low-Code workflow builder backend",
    version="1.0.0"
)
```

Responsibilities:

- Create database tables on startup
- Configure CORS middleware for frontend communication
- Register API routes
- Provide health check endpoints

CORS Configuration:

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000", "http://localhost:5173"],
    allow_credentials=True,
    allow_methods=["*"],
```

```
    allow_headers=["*"]
)
```

Routes:

- `GET /` - Root endpoint
- `GET /health` - Health check
- `/api/v1/*` - All API endpoints

---

2. Configuration Management

File: `backend/app/core/config.py`

Purpose: Centralized configuration using environment variables

Settings Class:

```
class Settings:
    DATABASE_URL: str           # PostgreSQL connection string
    GOOGLE_API_KEY: str         # Google Gemini API key
    SECRET_KEY: str             # Application secret key
    CHROMA_PERSIST_DIRECTORY: str  # ChromaDB storage path
    MAX_FILE_SIZE: int          # Maximum upload size (10MB)
    ALLOWED_FILE_TYPES: list    # [".pdf", ".txt", ".docx"]
```

Usage:

```
from app.core.config import settings
api_key = settings.GOOGLE_API_KEY
```

Environment Variables (`.env` file):

```
DATABASE_URL=postgresql://postgres:password@localhost:5432/workflow_db
GOOGLE_API_KEY=your_google_api_key_here
SECRET_KEY=your_secret_key_here
CHROMA_PERSIST_DIRECTORY=./chroma_db
```

---

3. Database Models

File: `backend/app/database/models.py`

Purpose: SQLAlchemy ORM models for database tables

Document Model

```python
class Document(Base):
    __tablename__ = "documents"

    id = Column(Integer, primary_key=True, index=True)
    filename = Column(String, nullable=False)
    original_filename = Column(String, nullable=False)
    file_path = Column(String, nullable=False)
    file_size = Column(Integer, nullable=False)
    content_type = Column(String, nullable=False)
    extracted_text = Column(Text)
    is_processed = Column(Boolean, default=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())
```

Purpose: Stores uploaded document metadata and extracted text

Workflow Model

```python
class Workflow(Base):
    __tablename__ = "workflows"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    description = Column(Text)
    components = Column(JSON, nullable=False)
    connections = Column(JSON, nullable=False)
    is_active = Column(Boolean, default=True)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), onupdate=func.now())

    chat_sessions = relationship("ChatSession", back_populates="workflow")
```

Purpose: Stores workflow definitions with components and connections

JSON Structure:

```json
{
  "components": [
    {
      "id": "user_query-1",
      "type": "user_query",
      "position": {"x": 100, "y": 100},
      "data": {"query": ""}
    }
  ],
  "connections": [
    {
      "id": "e1",
```

```
      "source": "user_query-1",
      "target": "knowledge_base-1"
    }
  ]
}
```

ChatSession Model

```python
class ChatSession(Base):
    __tablename__ = "chat_sessions"

    id = Column(Integer, primary_key=True, index=True)
    workflow_id = Column(Integer, ForeignKey("workflows.id"), nullable=False)
    session_name = Column(String)
    created_at = Column(DateTime(timezone=True), server_default=func.now())

    workflow = relationship("Workflow", back_populates="chat_sessions")
    messages = relationship("ChatMessage", back_populates="session")
```

Purpose: Manages chat conversation sessions

ChatMessage Model

```python
class ChatMessage(Base):
    __tablename__ = "chat_messages"

    id = Column(Integer, primary_key=True, index=True)
    session_id = Column(Integer, ForeignKey("chat_sessions.id"), nullable=False)
    message_type = Column(String, nullable=False)  # 'user' or 'assistant'
    content = Column(Text, nullable=False)
    message_metadata = Column(JSON)
    created_at = Column(DateTime(timezone=True), server_default=func.now())

    session = relationship("ChatSession", back_populates="messages")
```

Purpose: Stores individual chat messages

Relationships:

- Workflow → ChatSession (1:N)
- ChatSession → ChatMessage (1:N)

---

4. Database Connection

File: `backend/app/database/database.py`

Purpose: Manages database connection and session

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from app.core.config import settings

engine = create_engine(settings.DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Key Components:

- `engine`: SQLAlchemy database engine
- `SessionLocal`: Session factory
- `Base`: Base class for models
- `get_db()`: Dependency injection for database sessions

Usage in Endpoints:

```
@router.get("/workflows/")
def get_workflows(db: Session = Depends(get_db)):
    return db.query(Workflow).all()
```

---

5. API Routes

File: `backend/app/api/routes.py`

Purpose: Registers all API endpoint routers

```
from fastapi import APIRouter
from app.api.endpoints import documents, workflows, chat

router = APIRouter()

router.include_router(documents.router, prefix="/documents", tags=["documents"])
router.include_router(workflows.router, prefix="/workflows", tags=["workflows"])
router.include_router(chat.router, prefix="/chat", tags=["chat"])
```

Route Structure:

- `/api/v1/documents/*` - Document management
- `/api/v1/workflows/*` - Workflow CRUD operations
- `/api/v1/chat/*` - Chat session management

---

6. API Endpoints

Documents Endpoint

File: `backend/app/api/endpoints/documents.py`

Purpose: Handle document upload, processing, and management

Key Endpoints:

1. **Upload Document**

```python
@router.post("/upload")
async def upload_document(file: UploadFile, db: Session = Depends(get_db)):
    # Save file to disk
    # Create database record
    # Return document metadata
```

2. **Process Document**

```python
@router.post("/{document_id}/process")
async def process_document(document_id: int, db: Session = Depends(get_db)):
    # Extract text from PDF/TXT
    # Generate embeddings
    # Store in ChromaDB
    # Update is_processed flag
```

3. **List Documents**

```python
@router.get("/")
def get_documents(db: Session = Depends(get_db)):
    return db.query(Document).all()
```

4. **Delete Document**

```python
@router.delete("/{document_id}")
def delete_document(document_id: int, db: Session = Depends(get_db)):
    # Delete from database
```

```
    # Delete file from disk
    # Delete embeddings from ChromaDB
```

Workflows Endpoint

File: backend/app/api/endpoints/workflows.py

Purpose: Manage workflow CRUD and execution

Key Endpoints:

1. **Create Workflow**

```
@router.post("/")
def create_workflow(workflow: WorkflowCreate, db: Session = Depends(get_db)):
    # Validate workflow structure
    # Save to database
    # Return created workflow
```

2. **Execute Workflow**

```
@router.post("/execute")
async def execute_workflow(request: WorkflowExecuteRequest, db: Session =
Depends(get_db)):
    # Load workflow from database
    # Validate workflow
    # Execute components in order
    # Return response
```

3. **Update Workflow**

```
@router.put("/{workflow_id}")
def update_workflow(workflow_id: int, workflow: WorkflowUpdate, db: Session =
Depends(get_db)):
    # Find existing workflow
    # Update fields
    # Save changes
```

Chat Endpoint

File: backend/app/api/endpoints/chat.py

Purpose: Manage chat sessions and messages

Key Endpoints:

1. **Create Session**

```
@router.post("/sessions")
def create_session(session: ChatSessionCreate, db: Session = Depends(get_db)):
    # Create new chat session
    # Link to workflow
    # Return session ID
```

2. **Get Messages**

```
@router.get("/sessions/{session_id}/messages")
def get_messages(session_id: int, db: Session = Depends(get_db)):
    # Fetch all messages for session
    # Order by created_at
    # Return message list
```

3. **Delete Session**

```
@router.delete("/sessions/{session_id}")
def delete_session(session_id: int, db: Session = Depends(get_db)):
    # Delete session and all messages (cascade)
    # Return success response
```

Service Layer

1. Workflow Service

File: `backend/app/services/workflow_service.py`

Purpose: Orchestrate workflow execution

Key Functions:

validate_workflow()

```
def validate_workflow(workflow: dict) -> bool:
    """
    Validates workflow structure
    - Checks all components are connected
    - Verifies no cycles exist
    - Ensures valid configuration
    """
```

execute_workflow()

```
async def execute_workflow(workflow_id: int, query: str, db: Session) -> dict:
    """
    Executes workflow components in order
    1. Load workflow from database
    2. Topological sort components
    3. Execute each component
    4. Pass data between components
    5. Return final output
    """
```

Execution Flow:

```
User Query → Knowledge Base → LLM Engine → Output
     ↓              ↓              ↓           ↓
  query         context        response    formatted
```

topological_sort()

```
def topological_sort(components: list, connections: list) -> list:
    """
    Orders components for execution
    Uses depth-first search to determine execution order
    Ensures dependencies are executed first
    """
```

Example:

```
components = ["A", "B", "C", "D"]
connections = [("A", "B"), ("B", "C"), ("C", "D")]
result = ["A", "B", "C", "D"]  # Execution order
```

---

2. LLM Service

File: `backend/app/services/llm_service.py`

Purpose: Interface with Google Gemini API for text generation

Key Functions:

generate_response()

```
async def generate_response(
    query: str,
    context: str = "",
    model_name: str = "models/gemini-2.5-flash",
    temperature: float = 0.75,
    custom_prompt: str = ""
) -> str:
    """
    Generates AI response using Google Gemini

    Args:
        query: User's question
        context: Retrieved document context
        model_name: Gemini model to use
        temperature: Creativity level (0-1)
        custom_prompt: System instructions

    Returns:
        Generated response text
    """
```

Implementation:

```
import google.generativeai as genai

genai.configure(api_key=settings.GOOGLE_API_KEY)
model = genai.GenerativeModel(model_name)

prompt = f"{custom_prompt}\n\nContext: {context}\n\nQuery: {query}"
response = model.generate_content(prompt)
return response.text
```

Supported Models:

- `models/gemini-2.5-flash` - Fastest, most efficient
- `models/gemini-2.5-pro` - Most capable, best quality
- `models/gemini-2.0-flash` - Previous generation

---

3. Vector Service

File: `backend/app/services/vector_service.py`

Purpose: Manage embeddings and semantic search

Key Functions:

generate_embeddings()

```
def generate_embeddings(text: str) -> list[float]:
    """
    Generates vector embeddings using Google Gemini

    Args:
        text: Input text to embed

    Returns:
        768-dimensional embedding vector
    """
    result = genai.embed_content(
        model="models/embedding-001",
        content=text
    )
    return result['embedding']
```

store_embeddings()

```
def store_embeddings(document_id: str, chunks: list[str], embeddings: list):
    """
    Stores embeddings in ChromaDB

    Args:
        document_id: Unique document identifier
        chunks: Text chunks
        embeddings: Corresponding embeddings
    """
```

search_similar()

```
def search_similar(query: str, top_k: int = 3) -> list[dict]:
    """
    Performs semantic search in ChromaDB

    Args:
        query: Search query
        top_k: Number of results to return

    Returns:
        List of similar document chunks with metadata
    """
    query_embedding = generate_embeddings(query)
    results = collection.query(
        query_embeddings=[query_embedding],
        n_results=top_k
    )
    return results
```

ChromaDB Setup:

```
import chromadb

client = chromadb.PersistentClient(path=settings.CHROMA_PERSIST_DIRECTORY)
collection = client.get_or_create_collection(name="documents")
```

---

4. Document Service

File: `backend/app/services/document_service.py`

Purpose: Process and extract text from documents

Key Functions:

extract_text()

```
def extract_text(file_path: str, content_type: str) -> str:
    """
    Extracts text from PDF or TXT files

    Args:
        file_path: Path to file
        content_type: MIME type

    Returns:
        Extracted text content
    """
```

PDF Extraction (using PyMuPDF):

```
import fitz  # PyMuPDF

def extract_pdf_text(file_path: str) -> str:
    doc = fitz.open(file_path)
    text = ""
    for page in doc:
        text += page.get_text()
    return text
```

chunk_text()

```
def chunk_text(text: str, chunk_size: int = 500, overlap: int = 50) -> list[str]:
    """
    Splits text into overlapping chunks

    Args:
        text: Full text content
        chunk_size: Characters per chunk
        overlap: Overlapping characters

    Returns:
        List of text chunks
    """
```

Why Chunking?

- Embeddings work better on smaller text segments
- Enables more precise semantic search
- Reduces token usage in LLM calls

process_document()

```
async def process_document(document_id: int, db: Session):
    """
    Complete document processing pipeline
    1. Extract text from file
    2. Chunk text into segments
    3. Generate embeddings
    4. Store in ChromaDB
    5. Update database status
    """
```

Frontend Components

1. Application Entry Point

File: `frontend/src/main.jsx`

Purpose: React application initialization

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
```

```
    </React.StrictMode>
)
```

---

2. App Component

File: `frontend/src/App.jsx`

Purpose: Main application component with routing

```
import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-router-
dom'
import StacksList from './pages/StacksList'
import StackBuilder from './pages/StackBuilder'

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Navigate to="/stacks" replace />} />
        <Route path="/stacks" element={<StacksList />} />
        <Route path="/stacks/:id" element={<StackBuilder />} />
        <Route path="/stacks/new" element={<StackBuilder />} />
      </Routes>
    </Router>
  )
}
```

Routes:

- `/` → Redirects to `/stacks`
- `/stacks` → List all workflows
- `/stacks/:id` → Edit existing workflow
- `/stacks/new` → Create new workflow

---

3. API Client

File: `frontend/src/api/client.js`

Purpose: Axios HTTP client configuration

```
import axios from 'axios'

const API_BASE_URL = import.meta.env.VITE_API_URL || 'http://localhost:8000'

const client = axios.create({
  baseURL: API_BASE_URL,
  headers: {
```

```
    'Content-Type': 'application/json'
  }
})

export default client
```

Usage:

```
import client from './client'

const response = await client.get('/api/v1/workflows/')
```

---

4. API Services

Workflows API

File: `frontend/src/api/workflows.js`

Purpose: Workflow-related API calls

```
export const workflowsAPI = {
  getAll: async () => {
    const response = await client.get('/api/v1/workflows/')
    return response.data
  },

  getById: async (id) => {
    const response = await client.get(`/api/v1/workflows/${id}`)
    return response.data
  },

  create: async (workflow) => {
    const response = await client.post('/api/v1/workflows/', workflow)
    return response.data
  },

  update: async (id, workflow) => {
    const response = await client.put(`/api/v1/workflows/${id}`, workflow)
    return response.data
  },

  execute: async (workflowId, query) => {
    const response = await client.post('/api/v1/workflows/execute', {
      workflow_id: workflowId,
      query: query
    })
    return response.data
```

```
  }
}
```

Documents API

File: `frontend/src/api/documents.js`

Purpose: Document management API calls

```
export const documentsAPI = {
  upload: async (file) => {
    const formData = new FormData()
    formData.append('file', file)

    const response = await client.post('/api/v1/documents/upload', formData, {
      headers: { 'Content-Type': 'multipart/form-data' }
    })
    return response.data
  },

  process: async (documentId) => {
    const response = await client.post(`/api/v1/documents/${documentId}/process`)
    return response.data
  },

  getAll: async () => {
    const response = await client.get('/api/v1/documents/')
    return response.data
  }
}
```

Chat API

File: `frontend/src/api/chat.js`

Purpose: Chat session management

```
export const chatAPI = {
  createSession: async (workflowId) => {
    const response = await client.post('/api/v1/chat/sessions', {
      workflow_id: workflowId
    })
    return response.data
  },

  getSessions: async (workflowId) => {
    const response = await client.get(`/api/v1/chat/sessions?
workflow_id=${workflowId}`)
```

```
    return response.data
  },
```

getMessages: async (sessionId) => { const response = await client.get(`/api/v1/chat/sessions/${sessionId}/messages`) return response.data },

deleteSession: async (sessionId) => { await client.delete(`/api/v1/chat/sessions/${sessionId}`) } }

```
---

### 5. Page Components

#### StacksList Page

**File**: `frontend/src/pages/StacksList.jsx`

**Purpose**: Display all workflows and create new ones

**Key Features**:
- Lists all saved workflows
- Create new workflow button
- Navigate to workflow builder
- Empty state when no workflows exist

**State Management**:
```javascript
const [stacks, setStacks] = useState([])
const [showModal, setShowModal] = useState(false)
const [loading, setLoading] = useState(true)
```

Data Flow:

```
Component Mount → loadStacks() → API Call → Update State → Render List
```

Key Functions:

```
const loadStacks = async () => {
  const data = await workflowsAPI.getAll()
  setStacks(data)
}

const handleCreateStack = async (name, description) => {
  const newStack = await workflowsAPI.create({ name, description })
  navigate(`/stacks/${newStack.id}`)
}
```
```

StackBuilder Page

File: `frontend/src/pages/StackBuilder.jsx`

Purpose: Visual workflow builder with drag-and-drop

Key Features:

- ReactFlow canvas for visual workflow building
- Drag-and-drop components from sidebar
- Connect components with edges
- Save/load workflows
- Open chat interface

State Management:

```
const [nodes, setNodes, onNodesChange] = useNodesState([])
const [edges, setEdges, onEdgesChange] = useEdgesState([])
const [stackName, setStackName] = useState('New Stack')
const [showChat, setShowChat] = useState(false)
```

ReactFlow Integration:

```
<ReactFlow
  nodes={nodes}
  edges={edges}
  onNodesChange={onNodesChange}
  onEdgesChange={onEdgesChange}
  onConnect={onConnect}
  nodeTypes={nodeTypes}
  fitView
>
  <Background />
  <Controls />
</ReactFlow>
```

Drag & Drop Handler:

```
const onDrop = useCallback((event) => {
  const type = event.dataTransfer.getData('application/reactflow')
  const position = { x: event.clientX, y: event.clientY }

  const newNode = {
    id: `${type}-${Date.now()}`,
    type: getNodeType(type),
    position,
    data: { label, type, config: {} }
  }
```

```
    setNodes((nds) => nds.concat(newNode))
  }, [])
```

Save Workflow:

```
const handleSave = async () => {
  const components = nodes.map(node => ({
    id: node.id,
    type: node.data.type,
    position: node.position,
    data: node.data.config
  }))

  const connections = edges.map(edge => ({
    id: edge.id,
    source: edge.source,
    target: edge.target
  }))

  const stackData = { name: stackName, components, connections }

  if (id && id !== 'new') {
    await workflowsAPI.update(id, stackData)
  } else {
    const created = await workflowsAPI.create(stackData)
    navigate(`/stacks/${created.id}`)
  }
}
```

---

6. Component Library

CustomNodes Component

File: `frontend/src/components/CustomNodes.jsx`

Purpose: Defines all 4 workflow node types

Node Types:

1. UserQueryNode
2. KnowledgeBaseNode
3. LLMEngineNode
4. OutputNode

UserQueryNode:

```
export const UserQueryNode = memo(({ data, id }) => {
  const { deleteElements } = useReactFlow()

  const handleChange = (key, value) => {
    if (data.onConfigChange) {
      data.onConfigChange(id, { ...data.config, [key]: value })
    }
  }
```

return (

User Input <button onClick={() => deleteElements({ nodes: [{ id }] })}>
<input type="text" placeholder="Write your query here" value={data.config?.query || ''} onChange={(e) =>
handleChange('query', e.target.value)} />
) })

````
**KnowledgeBaseNode**:
```jsx
export const KnowledgeBaseNode = memo(({ data, id }) => {
  const [uploading, setUploading] = useState(false)

  const handleFileUpload = async (e) => {
    const file = e.target.files[0]
    setUploading(true)

    const doc = await documentsAPI.upload(file)
    await documentsAPI.process(doc.id)

    const uploadedFiles = data.config?.uploaded_files || []
    uploadedFiles.push({ id: doc.id, name: doc.original_filename })
    handleChange('uploaded_files', uploadedFiles)

    setUploading(false)
  }
````

return (

Knowledge Base
Choose File No file chosen <select value={data.config?.embedding_model ||
'models/embedding-001'} onChange={(e) => handleChange('embedding_model', e.target.value)} >
Gemini Embedding 001
) })

````
**LLMEngineNode**:
```jsx
export const LLMEngineNode = memo(({ data, id }) => {
````

```
  return (
    <div className="custom-node llm-engine-node">
      <Handle type="target" position={Position.Left} />
      <div className="node-header">
        <span>LLM (Gemini)</span>
      </div>
      <div className="node-body">
        <select
          value={data.config?.model_name || 'models/gemini-2.5-flash'}
          onChange={(e) => handleChange('model_name', e.target.value)}
        >
          <option value="models/gemini-2.5-flash">Gemini 2.5 Flash</option>
          <option value="models/gemini-2.5-pro">Gemini 2.5 Pro</option>
          <option value="models/gemini-2.0-flash">Gemini 2.0 Flash</option>
        </select>
```

```
    <textarea
      placeholder="Custom prompt"
      value={data.config?.custom_prompt || ''}
      onChange={(e) => handleChange('custom_prompt', e.target.value)}
    />

    <input
      type="number"
      step="0.1"
      min="0"
      max="1"
      value={data.config?.temperature || 0.75}
      onChange={(e) => handleChange('temperature', parseFloat(e.target.value))}
    />
  </div>
  <Handle type="source" position={Position.Right} />
</div>
```

) })

````
**Node Types Export**:
```javascript
export const nodeTypes = {
  userQuery: UserQueryNode,
  knowledgeBase: KnowledgeBaseNode,
  llmEngine: LLMEngineNode,
  output: OutputNode
}
````

````

ChatModal Component

File: `frontend/src/components/ChatModal.jsx`

Purpose: Interactive chat interface with history

Key Features:

- Send messages to workflow
- Display conversation history
- Session management
- Delete sessions
- Auto-load recent session

State Management:

```
const [messages, setMessages] = useState([])
const [input, setInput] = useState('')
const [loading, setLoading] = useState(false)
const [sessionId, setSessionId] = useState(null)
const [sessions, setSessions] = useState([])
const [showHistory, setShowHistory] = useState(false)
```

Send Message:

```
const handleSend = async () => {
  if (!input.trim()) return

  // Add user message to UI
  const userMessage = { message_type: 'user', content: input }
  setMessages([...messages, userMessage])

  setLoading(true)

  // Execute workflow
  const response = await workflowsAPI.execute(workflowId, input)

  // Add assistant response
  const assistantMessage = {
    message_type: 'assistant',
    content: response.response
  }
  setMessages([...messages, userMessage, assistantMessage])

  setInput('')
  setLoading(false)
}
```

Load Session History:

```
const loadSession = async (sessionId) => {
  const sessionMessages = await chatAPI.getMessages(sessionId)
  setMessages(sessionMessages)
  setSessionId(sessionId)
}
```

Delete Session:

```
const handleDeleteSession = async (sessionId) => {
  if (confirm('Delete this session?')) {
    await chatAPI.deleteSession(sessionId)
    if (sessionId === currentSessionId) {
      // Create new session if current one deleted
      const newSession = await chatAPI.createSession(workflowId)
      setSessionId(newSession.id)
      setMessages([])
    }
    loadSessions()
  }
}
```

Key Interactions

1. Workflow Creation Flow

```
User Action: Click "New Stack"
    ↓
Frontend: StacksList.jsx
    ↓
Modal: CreateStackModal.jsx (Enter name/description)
    ↓
API Call: workflowsAPI.create()
    ↓
Backend: POST /api/v1/workflows/
    ↓
Endpoint: workflows.py → create_workflow()
    ↓
Database: Insert into workflows table
    ↓
Response: { id: 1, name: "My Stack", ... }
    ↓
Frontend: Navigate to /stacks/1
    ↓
Page: StackBuilder.jsx (Edit mode)
```

2. Document Upload Flow

```
User Action: Click "Upload File" in Knowledge Base node
    ↓
Frontend: CustomNodes.jsx → handleFileUpload()
    ↓
API Call: documentsAPI.upload(file)
    ↓
Backend: POST /api/v1/documents/upload
    ↓
Endpoint: documents.py → upload_document()
    ↓
Service: Save file to disk
    ↓
Database: Insert into documents table
    ↓
Response: { id: 1, filename: "doc.pdf", ... }
    ↓
API Call: documentsAPI.process(doc.id)
    ↓
Backend: POST /api/v1/documents/1/process
    ↓
Service: document_service.py
    ├─► Extract text (PyMuPDF)
    ├─► Chunk text
    ├─► Generate embeddings (Gemini)
    └─► Store in ChromaDB
    ↓
Database: Update is_processed = true
    ↓
Frontend: Update node config with file info
```

3. Workflow Execution Flow

```
User Action: Type message in chat and press Enter
    ↓
Frontend: ChatModal.jsx → handleSend()
    ↓
API Call: workflowsAPI.execute(workflowId, query)
    ↓
Backend: POST /api/v1/workflows/execute
    ↓
Endpoint: workflows.py → execute_workflow()
    ↓
Service: workflow_service.py
    ├─► Load workflow from database
    ├─► Validate workflow structure
    ├─► Topological sort components
    └─► Execute components in order:
        │
```

```
        ├─► Component 1: User Query
        │   Input: { query: "What is AI?" }
        │   Output: { query: "What is AI?" }
        │
        ├─► Component 2: Knowledge Base
        │   Input: { query: "What is AI?" }
        │   Service: vector_service.py
        │   ├─► Generate query embedding
        │   ├─► Search ChromaDB
        │   └─► Return top 3 chunks
        │   Output: { context: "AI is...", sources: [...] }
        │
        ├─► Component 3: LLM Engine
        │   Input: { query: "What is AI?", context: "AI is..." }
        │   Service: llm_service.py
        │   ├─► Build prompt
        │   ├─► Call Gemini API
        │   └─► Get response
        │   Output: { response: "Artificial Intelligence is..." }
        │
        └─► Component 4: Output
            Input: { response: "Artificial Intelligence is..." }
            Output: { formatted_response: "...", metadata: {...} }
    ↓
Service: Save messages to database
    ├─► User message
    └─► Assistant response
    ↓
Response: { response: "...", sources: [...], execution_time: 2.5 }
    ↓
Frontend: Display response in chat
```

4. Chat History Flow

```
User Action: Open chat modal
    ↓
Frontend: ChatModal.jsx → useEffect()
    ↓
API Call: chatAPI.getSessions(workflowId)
    ↓
Backend: GET /api/v1/chat/sessions?workflow_id=1
    ↓
Endpoint: chat.py → get_sessions()
    ↓
Database: Query chat_sessions table
    ↓
Response: [{ id: 1, created_at: "..." }, ...]
    ↓
Frontend: Load most recent session
    ↓
API Call: chatAPI.getMessages(sessionId)
```

```
    ↓
Backend: GET /api/v1/chat/sessions/1/messages
    ↓
Endpoint: chat.py → get_messages()
    ↓
Database: Query chat_messages table
    ↓
Response: [{ message_type: "user", content: "..." }, ...]
    ↓
Frontend: Display messages in chat
```

API Reference

Complete Endpoint List

Documents API

Method	Endpoint	Description
POST	`/api/v1/documents/upload`	Upload document file
POST	`/api/v1/documents/{id}/process`	Process uploaded document
GET	`/api/v1/documents/`	List all documents
GET	`/api/v1/documents/{id}`	Get document by ID
DELETE	`/api/v1/documents/{id}`	Delete document

Workflows API

Method	Endpoint	Description
POST	`/api/v1/workflows/`	Create new workflow
GET	`/api/v1/workflows/`	List all workflows
GET	`/api/v1/workflows/{id}`	Get workflow by ID
PUT	`/api/v1/workflows/{id}`	Update workflow
DELETE	`/api/v1/workflows/{id}`	Delete workflow
POST	`/api/v1/workflows/execute`	Execute workflow
POST	`/api/v1/workflows/{id}/validate`	Validate workflow

Chat API

Method	Endpoint	Description
POST	`/api/v1/chat/sessions`	Create chat session

Method	Endpoint	Description
GET	`/api/v1/chat/sessions`	List sessions (filter by workflow_id)
GET	`/api/v1/chat/sessions/{id}`	Get session by ID
GET	`/api/v1/chat/sessions/{id}/messages`	Get session messages
DELETE	`/api/v1/chat/sessions/{id}`	Delete session

File Structure

Complete Project Structure

```
workflow-builder/
│
├── backend/
│   ├── app/
│   │   ├── api/
│   │   │   ├── endpoints/
│   │   │   │   ├── chat.py              # Chat endpoints
│   │   │   │   ├── documents.py         # Document endpoints
│   │   │   │   ├── workflows.py         # Workflow endpoints
│   │   │   │   └── __init__.py
│   │   │   ├── routes.py                # Route registration
│   │   │   └── __init__.py
│   │   │
│   │   ├── core/
│   │   │   ├── config.py                # Configuration settings
│   │   │   └── __init__.py
│   │   │
│   │   ├── database/
│   │   │   ├── database.py              # Database connection
│   │   │   ├── models.py                # SQLAlchemy models
│   │   │   └── __init__.py
│   │   │
│   │   ├── schemas/
│   │   │   ├── chat.py                  # Chat Pydantic schemas
│   │   │   ├── document.py              # Document schemas
│   │   │   ├── workflow.py              # Workflow schemas
│   │   │   └── __init__.py
│   │   │
│   │   ├── services/
│   │   │   ├── document_service.py      # Document processing
│   │   │   ├── llm_service.py           # LLM integration
│   │   │   ├── vector_service.py        # Vector operations
│   │   │   ├── workflow_service.py      # Workflow execution
│   │   │   └── __init__.py
│   │   │
│   │   ├── main.py                      # Application entry point
│   │   └── __init__.py
│   │
```

```
│   ├── alembic/
│   │   ├── versions/
│   │   │   └── 001_add_is_processed_column.py
│   │   ├── env.py
│   │   └── script.py.mako
│   │
│   ├── uploads/                        # Uploaded files storage
│   ├── chroma_db/                      # ChromaDB persistent storage
│   ├── .env                            # Environment variables
│   ├── .env.example                    # Environment template
│   ├── alembic.ini                     # Alembic configuration
│   ├── Dockerfile                      # Backend container
│   ├── requirements.txt                # Python dependencies
│   ├── reset_db.py                     # Database reset script
│   ├── test_chat.py                    # Chat testing script
│   ├── test_embeddings.py              # Embeddings testing
│   ├── test_gemini.py                  # Gemini API testing
│   └── README.md
│
├── frontend/
│   ├── src/
│   │   ├── api/
│   │   │   ├── chat.js                 # Chat API client
│   │   │   ├── client.js               # Axios configuration
│   │   │   ├── documents.js            # Documents API client
│   │   │   └── workflows.js            # Workflows API client
│   │   │
│   │   ├── components/
│   │   │   ├── ChatModal.jsx           # Chat interface
│   │   │   ├── ChatModal.css
│   │   │   ├── CreateStackModal.jsx    # New stack modal
│   │   │   ├── CreateStackModal.css
│   │   │   ├── CustomNodes.jsx         # All node types
│   │   │   ├── CustomNodes.css
│   │   │   ├── StackSidebar.jsx        # Component library
│   │   │   └── StackSidebar.css
│   │   │
│   │   ├── pages/
│   │   │   ├── StackBuilder.jsx        # Workflow builder
│   │   │   ├── StackBuilder.css
│   │   │   ├── StacksList.jsx          # Workflows list
│   │   │   └── StacksList.css
│   │   │
│   │   ├── App.jsx                     # Main app component
│   │   ├── App.css
│   │   ├── main.jsx                    # React entry point
│   │   └── index.css
│   │
│   ├── public/
│   ├── .env.example
│   ├── Dockerfile                      # Frontend container
│   ├── nginx.conf                      # Nginx configuration
│   ├── package.json                    # Node dependencies
│   ├── package-lock.json
```

```
│   ├── vite.config.js                  # Vite configuration
│   ├── index.html
│   └── README.md
│
├── docker-compose.yml                  # Multi-container orchestration
├── README.md                           # Main documentation
├── ARCHITECTURE.md                     # Architecture documentation
├── CODE_DOCUMENTATION.md               # This file
├── DEPLOYMENT.md                       # Deployment guide
└── DOCKER_QUICKSTART.md                # Quick start guide
```

Configuration Files

Backend Configuration

requirements.txt

```
fastapi==0.104.1              # Web framework
uvicorn==0.24.0               # ASGI server
sqlalchemy==2.0.23            # ORM
psycopg2-binary==2.9.9        # PostgreSQL driver
alembic==1.12.1               # Database migrations
pydantic==2.5.0               # Data validation
python-multipart==0.0.6       # File upload support
chromadb==0.4.18              # Vector database
google-generativeai==0.3.2    # Gemini API
PyMuPDF==1.23.8               # PDF processing
python-dotenv==1.0.0          # Environment variables
httpx==0.25.2                 # HTTP client
python-jose[cryptography]==3.3.0  # JWT tokens
passlib[bcrypt]==1.7.4        # Password hashing
requests==2.31.0              # HTTP requests
```

.env

```
DATABASE_URL=postgresql://postgres:password@localhost:5432/workflow_db
GOOGLE_API_KEY=your_google_api_key_here
SECRET_KEY=your_secret_key_here
CHROMA_PERSIST_DIRECTORY=./chroma_db
```

Frontend Configuration

package.json

```
{
  "name": "workflow-builder-frontend",
  "version": "1.0.0",
  "dependencies": {
    "react": "^18.3.1",
    "react-dom": "^18.3.1",
    "react-router-dom": "^6.22.0",
    "axios": "^1.6.7",
    "reactflow": "^11.10.4",
    "lucide-react": "^0.344.0"
  },
  "devDependencies": {
    "@vitejs/plugin-react": "^4.2.1",
    "vite": "^5.1.4"
  }
}
```

vite.config.js

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  server: {
    port: 5173,
    proxy: {
      '/api': {
        target: 'http://localhost:8000',
        changeOrigin: true
      }
    }
  }
})
```

Docker Configuration

docker-compose.yml

```
version: '3.8'

services:
  db:
    image: postgres:15-alpine
    environment:
      POSTGRES_DB: workflow_db
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
```

```
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

  backend:
    build: ./backend
    environment:
      DATABASE_URL: postgresql://postgres:password@db:5432/workflow_db
      GOOGLE_API_KEY: ${GOOGLE_API_KEY}
    ports:
      - "8000:8000"
    depends_on:
      - db

  frontend:
    build: ./frontend
    ports:
      - "3000:80"
    depends_on:
      - backend

volumes:
  postgres_data:
```

Development Guidelines

Adding New Component Type

1. **Define Node Component** (`frontend/src/components/CustomNodes.jsx`):

```
export const NewComponentNode = memo(({ data, id }) => {
  return (
    <div className="custom-node new-component-node">
      <Handle type="target" position={Position.Left} />
      {/* Component UI */}
      <Handle type="source" position={Position.Right} />
    </div>
  )
})
```

2. **Register Node Type**:

```
export const nodeTypes = {
  // ... existing types
  newComponent: NewComponentNode
}
```

3. **Add to Sidebar** (frontend/src/components/StackSidebar.jsx):

```
<div draggable onDragStart={(e) => onDragStart(e, 'new_component')}>
  New Component
</div>
```

4. **Implement Backend Logic** (backend/app/services/workflow_service.py):

```
def execute_new_component(component_data, input_data):
    # Component execution logic
    return output_data
```

Adding New API Endpoint

1. **Create Endpoint** (backend/app/api/endpoints/new_endpoint.py):

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from app.database.database import get_db

router = APIRouter()

@router.get("/")
def get_items(db: Session = Depends(get_db)):
    return {"items": []}
```

2. **Register Route** (backend/app/api/routes.py):

```
from app.api.endpoints import new_endpoint

router.include_router(new_endpoint.router, prefix="/new", tags=["new"])
```

3. **Create Frontend API Client** (frontend/src/api/new.js):

```
import client from './client'

export const newAPI = {
  getItems: async () => {
    const response = await client.get('/api/v1/new/')
    return response.data
  }
}
```

Adding Database Model

1. **Define Model** (backend/app/database/models.py):

```
class NewModel(Base):
    __tablename__ = "new_table"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

2. **Create Migration**:

```
cd backend
alembic revision --autogenerate -m "Add new_table"
alembic upgrade head
```

3. **Create Pydantic Schema** (backend/app/schemas/new.py):

```
from pydantic import BaseModel

class NewModelCreate(BaseModel):
    name: str

class NewModelResponse(BaseModel):
    id: int
    name: str

    class Config:
        from_attributes = True
```

Testing

Backend Testing

Test Gemini API:

```
cd backend
python test_gemini.py
```

Test Chat Functionality:

```
python test_chat.py
```

Test Embeddings:

```
python test_embeddings.py
```

Frontend Testing

Start Development Server:

```
cd frontend
npm run dev
```

Build for Production:

```
npm run build
```

Integration Testing

Test Complete Flow:

1. Start all services: `docker-compose up`
2. Create workflow in UI
3. Upload document
4. Chat with workflow
5. Verify response

Performance Optimization

Backend Optimizations

1. **Database Indexing**:

```
class Document(Base):
    id = Column(Integer, primary_key=True, index=True)  # Indexed
    filename = Column(String, index=True)  # Indexed for searches
```

2. **Async Operations**:

```
@router.post("/execute")
async def execute_workflow(request: WorkflowExecuteRequest):
    # Use async for I/O operations
```

```
    result = await llm_service.generate_response(...)
    return result
```

3. **Connection Pooling**:

```
engine = create_engine(
    settings.DATABASE_URL,
    pool_size=10,
    max_overflow=20
)
```

Frontend Optimizations

1. **Component Memoization**:

```
export const UserQueryNode = memo(({ data, id }) => {
  // Component only re-renders when props change
})
```

2. **Lazy Loading**:

```
const ChatModal = lazy(() => import('./components/ChatModal'))
```

3. **Code Splitting** (automatic with Vite)

Security Considerations

Implemented Security Measures

1. **Environment Variables**: API keys stored in `.env` files
2. **CORS Configuration**: Restricted to specific origins
3. **Input Validation**: Pydantic schemas validate all inputs
4. **File Type Restrictions**: Only PDF/TXT allowed
5. **File Size Limits**: Maximum 10MB uploads
6. **SQL Injection Prevention**: SQLAlchemy ORM parameterized queries
7. **Password Hashing**: Bcrypt for sensitive data

Security Best Practices

```
# Input validation
class WorkflowCreate(BaseModel):
    name: str = Field(..., min_length=1, max_length=100)
    description: Optional[str] = Field(None, max_length=500)
```

```
# File validation
ALLOWED_EXTENSIONS = {'.pdf', '.txt'}
MAX_FILE_SIZE = 10 * 1024 * 1024  # 10MB

# SQL injection prevention (automatic with SQLAlchemy)
db.query(Workflow).filter(Workflow.id == workflow_id).first()
```

Troubleshooting

Common Issues

Database Connection Failed:

```
# Check DATABASE_URL in .env
# Ensure PostgreSQL is running
docker-compose up -d db
```

Gemini API Error:

```
# Verify GOOGLE_API_KEY in .env
# Test with: python test_gemini.py
```

CORS Error:

```
# Check CORS middleware in main.py
# Ensure frontend URL is in allow_origins
```

File Upload Failed:

```
# Check uploads/ directory exists
# Verify file size < 10MB
# Ensure file type is PDF or TXT
```

Summary

This documentation provides a complete overview of the GenAI Stack codebase, including:

☑ **Architecture**: 3-tier architecture with clear separation of concerns ☑ **Components**: Detailed explanation of all frontend and backend components ☑ **Interactions**: Step-by-step flow diagrams for key operations ☑ **API Reference**: Complete list of all endpoints ☑ **Database Models**: SQLAlchemy models with relationships

☑ **Services**: Business logic layer with LLM, vector, and document services ☑ **Configuration**: Environment variables and settings management ☑ **File Structure**: Complete project organization ☑ **Development Guidelines**: How to extend the system ☑ **Testing**: Testing strategies and scripts ☑ **Performance**: Optimization techniques ☑ **Security**: Security measures and best practices

The codebase follows modern software engineering principles with modular design, clear interfaces, and comprehensive documentation.