OmniSource Chatbot - Complete Technical Documentation

Table of Contents

Project Overview

OmniSource is a multi-source chatbot capstone project that intelligently routes user queries across Excel spreadsheets and PDF documents to deliver accurate, context-aware answers through conversational dialogue.

Key Features

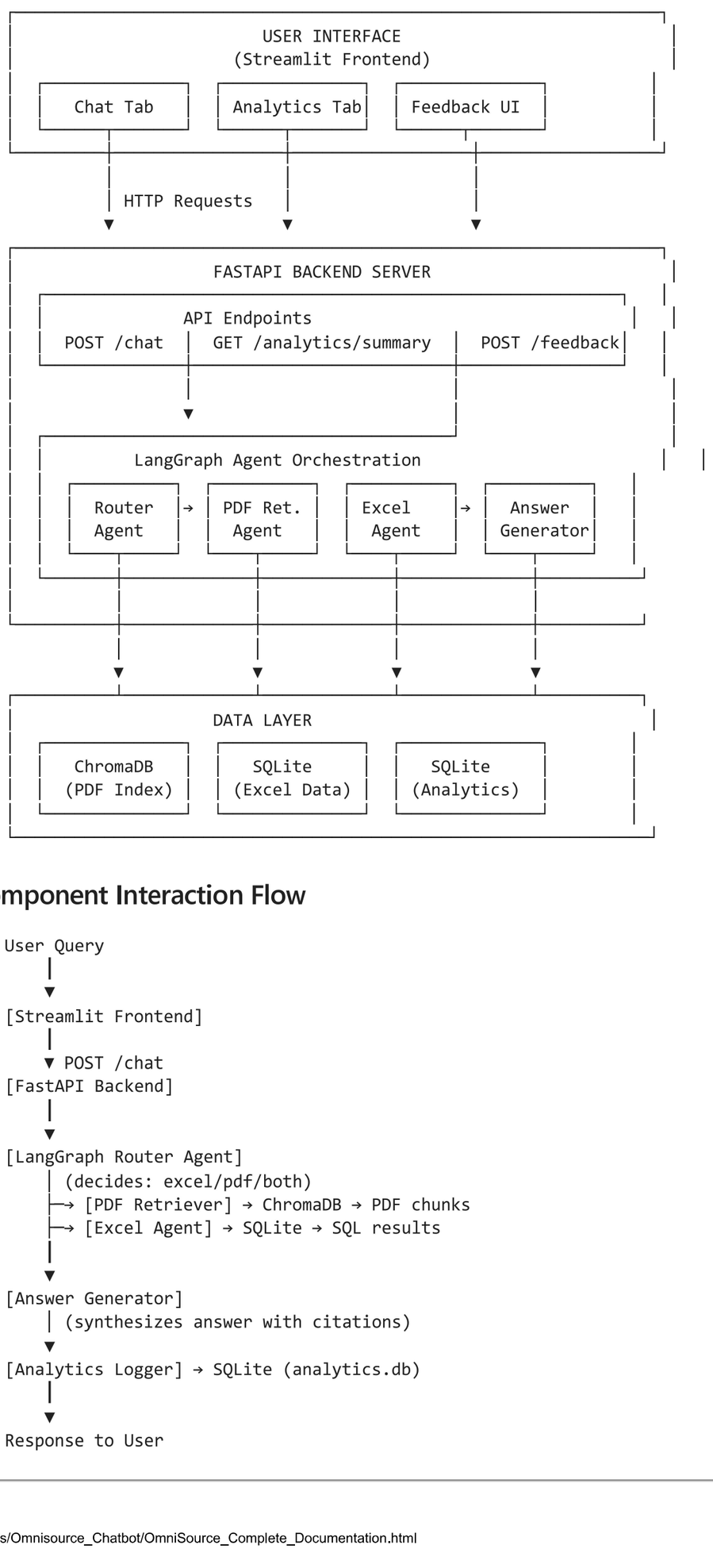
- **Multi-source routing**: Automatically selects between Excel (SQL) and PDF (vector search) based on query type
- **Conversational interface**: Maintains multi-turn conversations with context memory
- **Citation system**: Provides exact source references (Excel rows, PDF pages)
- **Feedback collection**: Thumbs up/down after every response
- **Analytics dashboard**: Tracks source usage, query success rates, and user satisfaction trends

Technology Stack

- **Backend**: FastAPI (Python web framework)
- **Frontend**: Streamlit (Python web UI framework)
- **Agent Framework**: LangGraph (graph-based orchestration)
- **Vector Database**: ChromaDB (for PDF embeddings)
- **SQL Database**: SQLite (for Excel/CSV data)
- **LLM**: Google Gemini 2.5 Flash (via API)
- **Data Processing**: Pandas (Excel parsing), LangChain (PDF chunking)

System Architecture

High-Level Architecture Diagram



Component Interaction Flow

```
User Query
    │
    ▼
[Streamlit Frontend]
    │
    ▼ POST /chat
[FastAPI Backend]
    │
    ▼
[LangGraph Router Agent]
    │ (decides: excel/pdf/both)
    ├──→ [PDF Retriever] → ChromaDB → PDF chunks
    ├──→ [Excel Agent] → SQLite → SQL results
    │
    ▼
[Answer Generator]
    │ (synthesizes answer with citations)
    ▼
[Analytics Logger] → SQLite (analytics.db)
    │
    ▼
Response to User
```

Project Structure

```
Omnisource_Chatbot/
├── backend/                    # FastAPI backend application
│   ├── __init__.py
│   ├── main.py                 # FastAPI app, API endpoints
│   ├── graph.py                # LangGraph agent orchestration
│   ├── llm.py                  # Gemini LLM integration
│   ├── db.py                   # Database connection & schema
│   ├── pdf_ingestion.py        # PDF processing & ChromaDB
│   ├── excel_ingestion.py       # Excel/CSV processing & SQLite
│   └── models.py               # Pydantic data models
│
├── frontend/                   # Streamlit frontend application
│   └── app.py                  # Streamlit UI & chat interface
│
├── Data/                       # Source data files
│   ├── omnisource_1.pdf        # PDF document 1 (1000+ pages)
│   ├── omnisource_2.pdf        # PDF document 2 (1000+ pages)
│   └── social-listening.csv    # Excel/CSV data
│
├── db/                         # SQLite databases
│   ├── excel.db                # Excel data storage
│   └── analytics.db             # Query analytics storage
│
├── chroma_pdfs/                # ChromaDB vector store
│   └── (ChromaDB files)
│
├── requirements.txt            # Python dependencies
└── .env                        # Environment variables (GEMINI_API_KEY)
```

Backend Components

1. main.py - FastAPI Application Server

Purpose: Main entry point for the FastAPI backend. Defines all HTTP endpoints, middleware, and startup logic.

Key Responsibilities:

- Initialize FastAPI application with CORS middleware
- Auto-ingest data files on startup
- Expose REST API endpoints for chat, analytics, and feedback
- Handle error logging and HTTP exceptions

---

2. graph.py - LangGraph Agent Orchestration

Purpose: Implements the agentic workflow using LangGraph. Defines the state machine, routing logic, and agent nodes.

Key Components:

- `OmniState` : TypedDict defining the graph state
- Router agent: Decides which source to use
- PDF retriever agent: Semantic search over PDFs
- Excel agent: SQL query generation and execution
- Answer generator: Synthesizes final response with citations

---

3. llm.py - Gemini LLM Integration

Purpose: Wrapper around Google Gemini API. Handles authentication, model configuration, and message formatting.

Key Functions:

- `_get_client()` : Creates Gemini client with system instructions
- `chat_completion()` : Formats messages and calls Gemini API

---

4. db.py - Database Management

Purpose: Manages SQLite database connections and schema initialization.

Key Functions:

- `get_excel_engine()` : Connection to Excel data database
- `get_analytics_engine()` : Connection to analytics database
- `init_analytics_schema()` : Creates analytics tables

---

5. pdf_ingestion.py - PDF Processing

Purpose: Handles PDF ingestion into ChromaDB vector store.

Key Functions:

- `ingest_pdfs()` : Loads, chunks, and stores PDFs in ChromaDB
- `pdf_semantic_search()` : Performs semantic search over PDF collection

---

6. excel_ingestion.py - Excel/CSV Processing

Purpose: Handles CSV/Excel ingestion into SQLite.

Key Functions:

- `ingest_social_listening()` : Loads CSV into SQLite table

---

7. models.py - Data Models

Purpose: Pydantic models for API request/response validation.

Key Models:

- `ChatRequest` , `ChatResponse`
- `FeedbackRequest`
- `AnalyticsSummary`
- `IngestionResponse`

Frontend Components

app.py - Streamlit User Interface

Purpose: Complete Streamlit frontend with chat interface and analytics dashboard.

Key Functions:

- `render_chat()`: Chat UI with message history and feedback buttons
- `render_analytics()`: Analytics dashboard with charts
- `main()`: Application entry point with tabs

Data Flow

Query Processing Flow

1. **User Input** → Streamlit frontend captures user message
2. **HTTP Request** → POST to `/chat` endpoint with conversation history
3. **Router Agent** → LLM decides: excel/pdf/both
4. **Retrieval**:
   - If PDF: Semantic search in ChromaDB → top 5 chunks
   - If Excel: LLM generates SQL → SQLite executes → results
5. **Answer Generation** → LLM synthesizes answer with citations
6. **Analytics Logging** → Query metadata saved to analytics.db
7. **Response** → JSON response with answer, citations, query_id
8. **UI Update** → Streamlit displays answer and feedback buttons

---

API Endpoints

Endpoint	Method	Purpose	Request Body	Response
`/chat`	POST	Process user query	`ChatRequest`	`ChatResponse`
`/feedback`	POST	Submit user feedback	`FeedbackRequest`	`{"status": "ok"}`
`/analytics/summary`	GET	Get analytics data	None	`AnalyticsSummary`
`/ingest`	POST	Re-ingest data files	None	`IngestionResponse`

Database Schemas

analytics.db - queries table

Column	Type	Description
id	INTEGER PRIMARY KEY	Auto-incrementing query ID
timestamp	TEXT	ISO format timestamp
conversation_id	TEXT	UUID of conversation
user_query	TEXT	Original user question
routed_source	TEXT	Source used: "pdf", "excel", "both"
success	INTEGER	Success flag (nullable)
feedback	INTEGER	User feedback: +1 (helpful), -1 (not helpful)
response_time_ms	REAL	Response time in milliseconds

excel.db - social_listening table

Contains all columns from the CSV file (21 columns including ProductModelName, ProductCategory, ProductPrice, RetailerName, etc.)

Line-by-Line Code Analysis

The following sections provide detailed line-by-line explanations of each file.

Detailed Code Analysis: backend/main.py

File Purpose

`main.py` is the FastAPI application entry point. It initializes the web server, sets up middleware, defines API endpoints, and handles startup tasks like data ingestion.

Line-by-Line Explanation

```python
from __future__ import annotations
```
Line 1: Enables postponed evaluation of annotations (Python 3.7+). Allows using forward references in type hints without quotes.

```python
import traceback
```
Line 3: Imports Python's traceback module for detailed error stack traces when exceptions occur.

```python
from pathlib import Path
```
Line 4: Imports Path from pathlib for cross-platform file path handling (better than os.path).

```python
from typing import Dict
```
Line 5: Imports Dict type hint for type annotations.

```python
from fastapi import FastAPI, HTTPException
```
Line 7: Imports FastAPI framework class and HTTPException for error handling.

```python
from fastapi.middleware.cors import CORSMiddleware
```
Line 8: Imports CORS middleware to allow cross-origin requests from Streamlit frontend.

```python
from .db import DATA_DIR, init_analytics_schema, get_analytics_engine
```
Line 10: Imports database utilities:

- `DATA_DIR`: Path to Data/ folder
- `init_analytics_schema`: Function to create analytics tables
- `get_analytics_engine`: Function to get SQLite connection

```python
from .excel_ingestion import ingest_social_listening
```

Line 11: Imports function to load CSV into SQLite.

```
from .graph import run_omni_graph
```
Line 12: Imports the main LangGraph workflow function.

```
from .models import (
    AnalyticsSummary,
    ChatRequest,
    ChatResponse,
    IngestionResponse,
    FeedbackRequest,
)
```
Lines 13-19: Imports Pydantic models for request/response validation.

```
from .pdf_ingestion import ingest_pdfs
```
Line 20: Imports function to load PDFs into ChromaDB.

```
from sqlalchemy import text
```
Line 21: Imports SQLAlchemy's text() for raw SQL queries.

```
app = FastAPI(title="OmniSource Backend", version="0.1.0")
```
Line 24: Creates FastAPI application instance with metadata.

```
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```
Lines 26-32: Adds CORS middleware to allow all origins, methods, and headers. This enables Streamlit (running on different port) to call the API.

```
@app.on_event("startup")
def startup_event():
```
Lines 35-36: Decorator registers function to run once when FastAPI server starts.

```
    init_analytics_schema()
```
Line 37: Creates analytics.db tables if they don't exist.

```
    pdf_paths = []
    for name in ["omnisource_1.pdf", "omnisource_2.pdf"]:
        p = DATA_DIR / name
        if p.exists():
            pdf_paths.append(p)
```
Lines 39-43: Collects PDF file paths that exist in Data/ folder.

```
    if pdf_paths:
        ingest_pdfs(pdf_paths)
```
Lines 44-45: If PDFs found, ingest them into ChromaDB on startup.

```
    csv_path = DATA_DIR / "social-listening.csv"
    if csv_path.exists():
        ingest_social_listening(csv_path)
```
Lines 47-49: If CSV exists, load it into SQLite on startup.

```
@app.post("/ingest", response_model=IngestionResponse)
def ingest_all():
```
Lines 52-53: POST endpoint to manually trigger data ingestion. Returns IngestionResponse with counts.

```
    pdf_paths = []
    for name in ["omnisource_1.pdf", "omnisource_2.pdf"]:
        p = DATA_DIR / name
        if p.exists():
            pdf_paths.append(p)
    pdf_chunks = ingest_pdfs(pdf_paths) if pdf_paths else 0
```
Lines 54-59: Same PDF collection logic, then ingests and gets chunk count.

```
    csv_path = DATA_DIR / "social-listening.csv"
    excel_rows = ingest_social_listening(csv_path) if csv_path.exists() else 0
```
Lines 61-62: Loads CSV if exists, gets row count.

```
    return IngestionResponse(pdf_chunks=pdf_chunks, excel_rows=excel_rows)
```
Line 64: Returns Pydantic model with ingestion statistics.

```
@app.post("/chat", response_model=ChatResponse)
def chat(req: ChatRequest):
```
Lines 67-68: Main chat endpoint. Accepts ChatRequest, returns ChatResponse.

```
    try:
        result = run_omni_graph(
            conversation_id=req.conversation_id,
            history=[m.dict() for m in req.messages],
        )
        return ChatResponse(**result)
```
Lines 69-74: Calls LangGraph workflow with conversation ID and message history. Converts Pydantic messages to dicts. Returns ChatResponse.

```
    except Exception as e:
        error_msg = f"Chat error: {str(e)}\n{traceback.format_exc()}"
        print(error_msg)  # Log to console
        raise HTTPException(status_code=500, detail=str(e))
```
Lines 75-78: Catches any exception, logs full traceback to console, returns HTTP 500 to client.

```
@app.post("/feedback")
def feedback(req: FeedbackRequest):
```
Lines 81-82: Endpoint to record user feedback (thumbs up/down).

```
    engine = get_analytics_engine()
    with engine.begin() as conn:
        conn.execute(
            text("UPDATE queries SET feedback = :fb WHERE id = :qid"),
            dict(fb=req.feedback, qid=req.query_id),
        )
    return {"status": "ok"}
```
Lines 83-89: Gets analytics DB connection, updates feedback column for specific query_id, returns success.

```
@app.get("/analytics/summary", response_model=AnalyticsSummary)
def analytics_summary():
```
Lines 92-93: GET endpoint returning aggregated analytics data.

```
    engine = get_analytics_engine()
    with engine.begin() as conn:
        total = conn.execute(text("SELECT COUNT(*) FROM queries")).scalar_one()
```
Lines 94-96: Gets total query count.

```
        by_source_rows = conn.execute(
            text(
                "SELECT routed_source, COUNT(*) AS c FROM queries "
```

```
            "GROUP BY routed_source"
        )
    ).fetchall()
    by_source: Dict[str, int] = {r[0] or "unknown": r[1] for r in by_source_rows}
```

Lines 97-103: Groups queries by routed_source (pdf/excel/both), creates dict mapping source → count.

```
    avg_rt = conn.execute(
        text("SELECT AVG(response_time_ms) FROM queries")
    ).scalar()
```

Lines 104-106: Calculates average response time.

```
    fb_rows = conn.execute(
        text(
            "SELECT feedback, COUNT(*) FROM queries "
            "WHERE feedback IS NOT NULL "
            "GROUP BY feedback"
        )
    ).fetchall()
```

Lines 107-113: Groups non-null feedback values, counts each.

```
feedback_summary: Dict[str, int] = {
    "up": 0,
    "down": 0,
}
for fb, count in fb_rows:
    if fb is None:
        continue
    if fb > 0:
        feedback_summary["up"] += count
    elif fb < 0:
        feedback_summary["down"] += count
```

Lines 115-125: Initializes feedback dict, then aggregates: positive feedback → "up", negative → "down".

```
return AnalyticsSummary(
    total_queries=int(total or 0),
    by_source=by_source,
    avg_response_time_ms=float(avg_rt or 0.0),
    feedback_summary=feedback_summary,
)
```

Lines 127-132: Returns AnalyticsSummary Pydantic model with all aggregated data.

Detailed Code Analysis: backend/graph.py

File Purpose

`graph.py` implements the core agentic workflow using LangGraph. It defines the state machine, routing logic, and specialized agent nodes that work together to process user queries.

Key Concepts

LangGraph: A library for building stateful, multi-actor applications with LLMs. It uses a graph-based approach where nodes are functions and edges define the flow.

State Machine: The `OmniState` TypedDict holds all data that flows through the graph:

- `messages`: Conversation history
- `routing_decision`: Which source to use (pdf/excel/both)
- `retrieval_context`: Retrieved data from sources
- `citations`: Source metadata for citations
- `routed_source`: Final source used (for analytics)
- `start_time`: Performance tracking

Line-by-Line Explanation

```
from __future__ import annotations
```

Line 1: Postponed annotation evaluation.

```
import time
```

Line 3: For timing query execution.

```
from datetime import datetime
```

Line 4: For timestamp generation in analytics.

```
from typing import Annotated, Dict, List, Literal, Optional, TypedDict
```

Line 5: Type hints:

- `Annotated`: For metadata in type hints
- `Literal`: For exact string values
- `TypedDict`: For structured dictionaries

```
from langchain_core.messages import AIMessage, HumanMessage, BaseMessage
```

Line 7: LangChain message types for conversation history.

```
from langgraph.graph import StateGraph, END
```

Line 8: LangGraph classes:

- `StateGraph`: Graph builder
- `END`: Terminal node

```
from langgraph.graph.message import add_messages
```

Line 9: Reducer function that merges message lists in state.

```
from sqlalchemy import text
```

Line 10: For SQL execution.

```
from .db import get_excel_engine, get_analytics_engine
```

Line 12: Database connection functions.

```
from .llm import chat_completion
```

Line 13: LLM wrapper function.

```
from .pdf_ingestion import pdf_semantic_search
```

Line 14: PDF search function.

```
class OmniState(TypedDict):
    messages: Annotated[List[BaseMessage], add_messages]
    routing_decision: Optional[Literal["pdf", "excel", "both"]]
    retrieval_context: Optional[str]
    citations: List[Dict]
    routed_source: Optional[str]
    start_time: float
```

Lines 17-23: Defines the graph state structure:

- `messages`: Annotated with `add_messages` reducer (automatically merges new messages)

- `routing_decision` : Can only be "pdf", "excel", or "both"
- `retrieval_context` : Accumulated context from sources
- `citations` : List of source metadata dicts
- `routed_source` : Final source used (for logging)
- `start_time` : Unix timestamp for performance tracking

```
ROUTER_SYSTEM_PROMPT = """
You are a routing controller for an analytics chatbot.
Decide the best primary source for answering the user's question:
- 'excel' for numeric, tabular, or metric-oriented questions about social listening, KPIs, or statistics.
- 'pdf' for conceptual, policy, or long-form document questions.
- 'both' if you clearly need quantitative evidence from Excel and narrative context from PDFs.

Answer with a single word: excel, pdf, or both.
"""
```

Lines 26-34: System prompt for the router agent. Instructs LLM to classify query type and return routing decision.

```
def _router_node(state: OmniState) -> OmniState:
```

Line 37: Router node function. Takes state, returns modified state.

```
    last_user_msg = [m for m in state["messages"] if isinstance(m, HumanMessage)][-1]
```

Line 38: Filters messages to find HumanMessage instances, gets the last one (most recent user query).

```
    route = chat_completion(
        ROUTER_SYSTEM_PROMPT,
        [{"role": "user", "content": last_user_msg.content}],
    ).strip().lower()
```

Lines 39-42: Calls LLM with router prompt and user question. Strips whitespace and lowercases response.

```
    if route not in {"excel", "pdf", "both"}:
        route = "pdf"
```

Lines 43-44: Validates LLM response. Defaults to "pdf" if invalid.

```
    state["routing_decision"] = route
    state["routed_source"] = route
    return state
```

Lines 45-47: Updates state with routing decision and returns.

```
def _pdf_retriever_node(state: OmniState) -> OmniState:
```

Line 50: PDF retriever node. Performs semantic search over PDF collection.

```
    last_user_msg = [m for m in state["messages"] if isinstance(m, HumanMessage)][-1]
```

Line 51: Gets last user message.

```
    result = pdf_semantic_search(last_user_msg.content, k=5)
```

Line 52: Searches ChromaDB for top 5 relevant chunks.

```
    docs = result.get("documents", [[]])[0]
    metas = result.get("metadatas", [[]])[0]
```

Lines 53-54: Extracts documents and metadata from ChromaDB result. `[[]]` is default if key missing, `[0]` gets first query result.

```
    context_parts = []
    citations: List[Dict] = []
```

Lines 56-57: Initialize lists for context text and citation metadata.

```
    for doc, meta in zip(docs, metas):
        file_name = meta.get("file_name")
        page = meta.get("page")
        context_parts.append(f"[{file_name}, page {page}]\n{doc}")
        citations.append(
            {
                "source_type": "pdf",
                "file_name": file_name,
                "page": page,
            }
        )
```

Lines 58-68: For each retrieved chunk:

- Formats context with file name and page number
- Creates citation dict with metadata

```
    if context_parts:
        state["retrieval_context"] = "\n\n".join(context_parts)
        state["citations"].extend(citations)
```

Lines 70-72: If chunks found, joins them with double newlines and adds citations to state.

```
    return state
```

Line 73: Returns updated state.

```
EXCEL_SQL_SYSTEM_PROMPT = """
You translate natural language questions into safe SQL queries for a SQLite database.
...
"""
```

Lines 76-123: Detailed system prompt for SQL generation:

- Describes database schema (all 21 columns)
- Provides rules (SELECT only, use WHERE filters, aggregations)
- Includes 3 concrete examples

```
def _excel_agent_node(state: OmniState) -> OmniState:
```

Line 126: Excel agent node. Generates SQL, executes it, formats results.

```
    last_user_msg = [m for m in state["messages"] if isinstance(m, HumanMessage)][-1]
    nl = last_user_msg.content
```

Lines 127-128: Gets user question in natural language.

```
    sql = chat_completion(
        EXCEL_SQL_SYSTEM_PROMPT,
        [{"role": "user", "content": nl}],
    ).strip()
```

Lines 129-132: Calls LLM to generate SQL from natural language.

```
    if not sql.lower().startswith("select"):
        sql = "SELECT 'NO_ANSWER' AS note;"
```

Lines 134-136: Safety check: only allows SELECT queries. Rejects anything else.

```
    engine = get_excel_engine()
    rows = []
    columns: List[str] = []
```

Lines 138-140: Gets database connection, initializes result containers.

```
    with engine.begin() as conn:
        result = conn.execute(text(sql))
        columns = list(result.keys())
        rows = [dict(zip(columns, r)) for r in result.fetchall()]
```

Lines 141-144: Executes SQL in transaction:

- `result.keys()`: Column names
- `fetchall()`: All rows
- Converts each row to dict mapping column → value

```
if len(rows) == 1 and rows[0].get("note") == "NO_ANSWER":
    table_text = "No structured answer available from Excel for this question."
```

Lines 146-147: If LLM returned NO_ANSWER, sets error message.

```
else:
    header = " | ".join(columns)
    sep = " | ".join(["---"] * len(columns))
    body_lines = []
    for r in rows[:50]:
        body_lines.append(" | ".join(str(r.get(c, "")) for c in columns))
    table_text = header + "\n" + sep + "\n" + "\n".join(body_lines)
```

Lines 148-155: Formats SQL results as markdown table:

- Header row with column names
- Separator row with "---"
- Data rows (max 50 to avoid token limits)
- Each cell converted to string

```
context = f"Structured result from Excel (table social_listening):\n{table_text}"
prev = state.get("retrieval_context")
state["retrieval_context"] = (prev + "\n\n" + context) if prev else context
```

Lines 157-159: Adds Excel results to retrieval context. If PDF context exists, appends; otherwise sets.

```
    state["citations"].append(
        {
            "source_type": "excel",
            "table": "social_listening",
            "note": "SQLite query result",
        }
    )
```

Lines 160-166: Adds Excel citation to citations list.

```
    return state
```

Line 167: Returns updated state.

```
ANSWER_SYSTEM_PROMPT = """
You are OmniSource, a multi-source analytics assistant.
...
"""
```

Lines 170-181: System prompt for answer generation:

- Describes available sources
- Guidelines: use context, don't invent, be conversational, cite sources

```
def _answer_node(state: OmniState) -> OmniState:
```

Line 184: Answer generator node. Synthesizes final response.

```
    last_user_msg = [m for m in state["messages"] if isinstance(m, HumanMessage)][-1]
    retrieval_context = state.get("retrieval_context") or "No external context."
    citations = state.get("citations", [])
```

Lines 185-187: Gets user question, retrieval context (or default), and citations.

```
    prompt = (
        f"User question:\n{last_user_msg.content}\n\n"
        f"Retrieved context:\n{retrieval_context}\n\n"
        f"Available citations metadata:\n{citations}\n\n"
        "Now answer the question. At the end, add a 'Sources:' section listing each source "
        "in the form 'PDF: <file>, page <n>' or 'Excel: social_listening table'."
    )
```

Lines 189-195: Constructs prompt with:

- User question
- Retrieved context (PDF chunks or SQL results)
- Citations metadata
- Instructions to cite sources

```
    answer_text = chat_completion(
        ANSWER_SYSTEM_PROMPT,
        [{"role": "user", "content": prompt}],
    )
```

Lines 197-200: Calls LLM to generate answer.

```
    state["messages"].append(AIMessage(content=answer_text))
```

Line 201: Adds AI response to message history.

```
    return state
```

Line 202: Returns updated state.

```
def _log_analytics(state: OmniState, conversation_id: str) -> int:
```

Line 205: Logs query to analytics database. Returns query_id.

```
    engine = get_analytics_engine()
    now = datetime.utcnow().isoformat()
    routed_source = state.get("routed_source") or "unknown"
    duration_ms = max(0.0, (time.time() - state.get("start_time", time.time())) * 1000.0)
```

Lines 206-209: Prepares analytics data:

- Gets analytics DB connection
- Current timestamp in ISO format
- Routed source (defaults to "unknown")
- Calculates duration in milliseconds

```
    with engine.begin() as conn:
        result = conn.execute(
            text(
                """
                INSERT INTO queries (timestamp, conversation_id, user_query, routed_source,
                                     success, feedback, response_time_ms)
                VALUES (:ts, :cid, :q, :src, :success, :feedback, :rt)
                """
            ),
            dict(
                ts=now,
                cid=conversation_id,
                q=_get_last_user_text(state),
                src=routed_source,
                success=None,
                feedback=None,
```

```
                rt=duration_ms,
            ),
        )
```

Lines 211-229: Inserts query record with all metadata.

```
        query_id = getattr(result, "lastrowid", None)
        if not query_id:
            query_id = conn.execute(text("SELECT last_insert_rowid()")).scalar_one()
```

Lines 230-232: Gets inserted row ID (SQLite auto-increment). Tries attribute first, then SQL fallback.

```
    return int(query_id or 0)
```

Line 234: Returns query_id as integer.

```
def _get_last_user_text(state: OmniState) -> str:
    users = [m for m in state["messages"] if isinstance(m, HumanMessage)]
    return users[-1].content if users else ""
```

Lines 237-239: Helper to extract last user message text.

```
def build_graph():
```

Line 242: Builds and compiles the LangGraph workflow.

```
    graph = StateGraph(OmniState)
```

Line 243: Creates new state graph with OmniState schema.

```
    graph.add_node("router", _router_node)
    graph.add_node("pdf_retriever", _pdf_retriever_node)
    graph.add_node("excel_agent", _excel_agent_node)
    graph.add_node("answer", _answer_node)
```

Lines 245-248: Registers all node functions with names.

```
    graph.set_entry_point("router")
```

Line 250: Sets "router" as the starting node.

```
    def route_decision(state: OmniState):
        decision = state.get("routing_decision") or "pdf"
        if decision == "pdf":
            return "pdf_retriever"
        if decision == "excel":
            return "excel_agent"
        if decision == "both":
            return "pdf_retriever"
        return "pdf_retriever"
```

Lines 252-261: Conditional routing function:

- Gets routing decision from state
- Returns next node name based on decision
- "both" starts with PDF retriever

```
    graph.add_conditional_edges(
        "router",
        route_decision,
        {
            "pdf_retriever": "pdf_retriever",
            "excel_agent": "excel_agent",
        },
    )
```

Lines 263-270: Adds conditional edge from router:

- Calls `route_decision` function
- Maps return values to target nodes

```
    def after_pdf(state: OmniState):
        decision = state.get("routing_decision") or "pdf"
        if decision == "both":
            return "excel_agent"
        return "answer"
```

Lines 272-277: After PDF retriever, checks if "both" was requested:

- If yes, goes to Excel agent
- Otherwise goes to answer generator

```
    graph.add_conditional_edges(
        "pdf_retriever",
        after_pdf,
        {
            "excel_agent": "excel_agent",
            "answer": "answer",
        },
    )
```

Lines 279-286: Adds conditional edge from PDF retriever.

```
    graph.add_edge("excel_agent", "answer")
    graph.add_edge("answer", END)
```

Lines 288-289: Adds fixed edges:

- Excel agent always goes to answer
- Answer always ends workflow

```
    return graph.compile()
```

Line 292: Compiles graph into executable workflow.

```
def run_omni_graph(conversation_id: str, history: List[Dict]) -> Dict:
```

Line 295: Main entry point. Runs the complete workflow.

```
    messages: List[BaseMessage] = []
    for m in history:
        if m["role"] == "user":
            messages.append(HumanMessage(content=m["content"]))
        else:
            messages.append(AIMessage(content=m["content"]))
```

Lines 300-305: Converts dict messages to LangChain message objects.

```
    workflow = build_graph()
    initial_state: OmniState = {
        "messages": messages,
        "routing_decision": None,
        "retrieval_context": None,
        "citations": [],
        "routed_source": None,
        "start_time": time.time(),
    }
```

Lines 307-315: Builds graph and creates initial state with:

- Conversation history
- Empty routing/context

- Current timestamp

```
    final_state = workflow.invoke(initial_state)
```
Line 317: Executes workflow with initial state.

```
    query_id = _log_analytics(final_state, conversation_id)
```
Line 318: Logs query to analytics, gets query_id.

```
    answer_msg = [m for m in final_state["messages"] if isinstance(m, AIMessage)][-1]
```
Line 320: Extracts last AI message (the answer).

```
    return {
        "answer": answer_msg.content,
        "routed_source": final_state.get("routed_source"),
        "citations": final_state.get("citations", []),
        "query_id": query_id,
    }
```
Lines 321-326: Returns dict with answer, metadata, and query_id for feedback.

Detailed Code Analysis: Remaining Backend Files

backend/llm.py - Gemini LLM Integration

Purpose: Wraps Google Gemini API with proper authentication and message formatting.

Key Lines:

- **Line 3**: `from dotenv import load_dotenv` - Loads environment variables from .env file
- **Line 5**: `load_dotenv()` - Actually loads the .env file into environment
- **Line 8**: `GEMINI_MODEL_NAME = os.getenv("GEMINI_MODEL_NAME", "gemini-2.5-flash")` - Gets model name from env, defaults to gemini-2.5-flash
- **Line 11-18**: `_get_client()` - Creates Gemini client:
  - Checks for API key, raises error if missing
  - Configures genai with API key
  - If system_instruction provided, creates model with it (Gemini's way of setting system prompts)
  - Returns configured GenerativeModel
- **Line 21-40**: `chat_completion()` - Main LLM call function:
  - Creates client with system prompt
  - Converts messages: "assistant" → "model" (Gemini's role name)
  - Calls `generate_content()` with message history
  - Handles errors and empty responses

backend/db.py - Database Management

Purpose: Manages SQLite database connections and schema.

Key Lines:

- **Line 10**: `BASE_DIR = Path(__file__).resolve().parent.parent` - Gets project root directory
- **Line 11**: `DATA_DIR = BASE_DIR / "Data"` - Path to Data/ folder
- **Line 12**: `DB_DIR = BASE_DIR / "db"` - Path to db/ folder
- **Line 13**: `DB_DIR.mkdir(exist_ok=True)` - Creates db/ folder if it doesn't exist
- **Line 16-17**: Defines paths to SQLite database files
- **Line 20-21**: `get_excel_engine()` - Creates SQLAlchemy engine for Excel database
- **Line 24-25**: `get_analytics_engine()` - Creates SQLAlchemy engine for analytics database
- **Line 28-46**: `init_analytics_schema()` - Creates queries table with all columns (id, timestamp, conversation_id, user_query, routed_source, success, feedback, response_time_ms)

backend/pdf_ingestion.py - PDF Processing

Purpose: Handles PDF ingestion into ChromaDB vector store.

Key Lines:

- **Line 19-25**: `get_pdf_client()` - Creates persistent ChromaDB client pointing to chroma_pdfs/ directory
- **Line 31-33**: `get_pdf_collection()` - Gets or creates "pdf_docs" collection
- **Line 36-71**: `ingest_pdfs()` - Main ingestion function:
  - Creates text splitter (chunk_size=1200, overlap=200)
  - For each PDF: loads with PyPDFLoader, splits into chunks
  - Extracts metadata (file_name, page number)
  - Generates UUID for each chunk
  - Adds to ChromaDB collection
  - Returns total chunk count
- **Line 74-77**: `pdf_semantic_search()` - Queries ChromaDB collection with user query, returns top k results

backend/excel_ingestion.py - Excel/CSV Processing

Purpose: Handles CSV ingestion into SQLite.

Key Lines:

- **Line 11-23**: `ingest_social_listening()` - Main ingestion function:
  - Reads CSV with pandas
  - Gets SQLite engine
  - Uses `df.to_sql()` to write to database (if_exists="replace" means overwrite on re-ingest)
  - Returns row count
- **Line 26-33**: `run_structured_query()` - Placeholder function (not used, SQL execution happens in graph.py)

backend/models.py - Pydantic Data Models

Purpose: Defines request/response schemas for API validation.

Key Models:

- **IngestionResponse** (lines 8-10): `pdf_chunks: int, excel_rows: int`
- **ChatMessage** (lines 13-15): `role: str, content: str`
- **ChatRequest** (lines 18-20): `conversation_id: str, messages: List[ChatMessage]`
- **ChatResponse** (lines 23-27): `answer: str, routed_source: Optional[str], citations: List[Dict], query_id: Optional[int]`
- **FeedbackRequest** (lines 30-32): `query_id: int, feedback: int`
- **AnalyticsSummary** (lines 35-39): `total_queries: int, by_source: Dict[str, int], avg_response_time_ms: float, feedback_summary: Dict[str, int]`

Detailed Code Analysis: frontend/app.py

File Purpose

`app.py` is the complete Streamlit frontend application. It provides the chat interface, analytics dashboard, and feedback collection UI.

Line-by-Line Explanation

```python
import os
import uuid
import requests
import streamlit as st
import pandas as pd
import altair as alt
```

Lines 1-7: Imports:

- `os` : Environment variables
- `uuid` : Generate conversation IDs
- `requests` : HTTP client for API calls
- `streamlit` : UI framework
- `pandas` : Data manipulation for charts
- `altair` : Chart library

```python
BACKEND_URL = os.getenv("OMNISOURCE_BACKEND_URL", "http://localhost:8000")
```

Line 10: Gets backend URL from env, defaults to localhost:8000.

```python
def ensure_conversation_id():
    if "conversation_id" not in st.session_state:
        st.session_state["conversation_id"] = str(uuid.uuid4())
    if "messages" not in st.session_state:
        st.session_state["messages"] = []
```

Lines 13-17: Initializes session state:

- `conversation_id` : UUID for tracking conversation
- `messages` : List of message dicts

```python
def render_chat():
```

Line 20: Main chat rendering function.

```python
    st.subheader("OmniSource Chatbot")
    ensure_conversation_id()
```

Lines 21-22: Sets header, ensures session state initialized.

```python
    for m in st.session_state["messages"]:
        role = "user" if m["role"] == "user" else "assistant"
        with st.chat_message(role):
            st.markdown(m["content"])
```

Lines 24-28: Renders chat history:

- Iterates through stored messages
- Determines role for Streamlit chat_message widget
- Displays message content as markdown

```python
    user_input = st.chat_input("Ask about your data...")
```

Line 30: Creates chat input widget at bottom of screen.

```python
    if user_input:
        st.session_state["messages"].append({"role": "user", "content": user_input})
        with st.chat_message("user"):
            st.markdown(user_input)
```

Lines 31-34: When user submits:

- Adds user message to session state
- Displays it in chat UI

```python
        with st.chat_message("assistant"):
            with st.spinner("Thinking..."):
                resp = requests.post(
                    f"{BACKEND_URL}/chat",
                    json={
                        "conversation_id": st.session_state["conversation_id"],
                        "messages": st.session_state["messages"],
                    },
                    timeout=120,
                )
```

Lines 36-45: Shows assistant message area with spinner:

- POSTs to /chat endpoint
- Sends conversation_id and full message history
- 120 second timeout

```python
                resp.raise_for_status()
                data = resp.json()
                answer = data["answer"]
                st.session_state["last_query_id"] = data.get("query_id")
                st.markdown(answer)
                st.session_state["messages"].append(
                    {"role": "assistant", "content": answer}
                )
```

Lines 46-54: Processes response:

- Checks HTTP status
- Parses JSON
- Extracts answer and query_id
- Displays answer
- Saves to message history

```python
    if st.session_state["messages"]:
        last_msg = st.session_state["messages"][-1]
        qid = st.session_state.get("last_query_id")
        if last_msg["role"] == "assistant" and qid is not None:
            col1, col2 = st.columns(2)
            with col1:
                if st.button("Helpful", key=f"up_{qid}"):
                    try:
                        resp = requests.post(
                            f"{BACKEND_URL}/feedback",
                            json={"query_id": qid, "feedback": 1},
                            timeout=10,
                        )
                        resp.raise_for_status()
                        st.success("Thanks for the feedback.")
                    except Exception as e:
                        st.error(f"Failed to submit feedback: {e}")
            with col2:
                if st.button("Not helpful", key=f"down_{qid}"):
                    try:
```

```
            resp = requests.post(
                f"{BACKEND_URL}/feedback",
                json={"query_id": qid, "feedback": -1},
                timeout=10,
            )
            resp.raise_for_status()
            st.info("Feedback recorded.")
        except Exception as e:
            st.error(f"Failed to submit feedback: {e}")
```

Lines 56-85: Feedback buttons:

- Only shows if last message is assistant and query_id exists
- Two columns for two buttons
- On click: POSTs to /feedback endpoint
- Shows success/error message

```
def render_analytics():
```

Line 88: Analytics dashboard rendering function.

```
    st.subheader("Analytics Dashboard")
    resp = requests.get(f"{BACKEND_URL}/analytics/summary", timeout=30)
    if not resp.ok:
        st.error("Failed to load analytics summary.")
        return
    data = resp.json()
```

Lines 89-94: Fetches analytics data from backend.

```
    st.metric("Total Queries", data["total_queries"])
    st.metric("Avg Response Time (ms)", round(data["avg_response_time_ms"], 1))
```

Lines 96-97: Displays key metrics.

```
    if data["total_queries"] == 0:
        st.info("No queries yet. Start chatting to see analytics here.")
        return
```

Lines 99-101: Early return if no data.

```
    st.markdown("### Query Source Usage")
    if data["by_source"]:
        source_df = pd.DataFrame(
            [
                {"Source": src.capitalize(), "Count": count}
                for src, count in data["by_source"].items()
            ]
        )
        source_chart = (
            alt.Chart(source_df)
            .mark_bar()
            .encode(
                x=alt.X("Source", sort="-y", title="Primary routed source"),
                y=alt.Y("Count", title="Number of queries"),
                tooltip=["Source", "Count"],
            )
            .properties(height=300)
        )
        st.altair_chart(source_chart, use_container_width=True)
```

Lines 103-122: Source usage chart:

- Creates DataFrame from source counts
- Builds Altair bar chart
- Sorts by count descending
- Adds tooltips

```
    st.markdown("### Feedback Overview")
    fb_up = data["feedback_summary"].get("up", 0)
    fb_down = data["feedback_summary"].get("down", 0)
    if fb_up == 0 and fb_down == 0:
        st.write("No feedback provided yet.")
    else:
        fb_df = pd.DataFrame(
            [
                {"Feedback": "Helpful", "Count": fb_up},
                {"Feedback": "Not helpful", "Count": fb_down},
            ]
        )
        fb_chart = (
            alt.Chart(fb_df)
            .mark_bar()
            .encode(
                x=alt.X("Feedback", title="User feedback"),
                y=alt.Y("Count", title="Number of responses"),
                color=alt.Color("Feedback", legend=None),
                tooltip=["Feedback", "Count"],
            )
            .properties(height=300)
        )
        st.altair_chart(fb_chart, use_container_width=True)
```

Lines 124-150: Feedback chart:

- Gets up/down counts
- Shows message if no feedback
- Otherwise creates bar chart with colors

```
def main():
    st.set_page_config(page_title="OmniSource Chatbot")
    st.title("OmniSource Multi-Source Analytics Assistant Chatbot")
    tab_chat, tab_analytics = st.tabs(["Chat", "Analytics"])
    with tab_chat:
        render_chat()
    with tab_analytics:
        render_analytics()
```

Lines 153-161: Main function:

- Sets page config
- Creates title
- Creates two tabs
- Renders each tab's content

```
if __name__ == "__main__":
    main()
```

Lines 164-165: Entry point when run directly.

Summary and Key Design Decisions

Architecture Decisions

1. **Separate Storage for PDFs and Excel**:
   - PDFs → ChromaDB (vector database for semantic search)
   - Excel → SQLite (relational database for structured queries)
   - This separation allows optimal retrieval strategies for each data type
2. **LangGraph for Agent Orchestration**:
   - Graph-based approach provides clear control flow
   - State machine pattern ensures data consistency
   - Easy to add new agent nodes or modify routing logic
3. **Source Routing Intelligence**:
   - LLM-based router analyzes query intent
   - Supports three routing modes: pdf, excel, both
   - "Both" mode allows combining quantitative and narrative data
4. **Citation System**:
   - Every answer includes source references
   - PDF citations: file name + page number
   - Excel citations: table name
   - Enables traceability and verification
5. **Analytics and Feedback Loop**:
   - Every query logged with metadata
   - User feedback collected and aggregated
   - Dashboard provides insights into system performance

Data Flow Summary

```
User Query
    ↓
[Router Agent] → Decides: excel/pdf/both
    ↓
[Retrieval Agents] → Fetch relevant data
    ├→ PDF: Semantic search in ChromaDB
    └→ Excel: SQL query in SQLite
    ↓
[Answer Generator] → Synthesizes response with citations
    ↓
[Analytics Logger] → Records query metadata
    ↓
Response to User
```

Security Considerations

1. **SQL Injection Prevention**:
   - Only SELECT queries allowed
   - SQL validation before execution
   - Parameterized queries via SQLAlchemy
2. **API Key Management**:
   - Environment variables via .env file
   - Never hardcoded in source code
   - Loaded at runtime
3. **CORS Configuration**:
   - Currently allows all origins (development)
   - Should be restricted in production

Performance Optimizations

1. **PDF Chunking**:
   - Chunk size: 1200 characters
   - Overlap: 200 characters (maintains context)
   - Limits retrieval to top 5 chunks
2. **SQL Result Limiting**:
   - Excel results capped at 50 rows
   - Prevents token limit issues
   - Maintains response quality
3. **Connection Pooling**:
   - SQLAlchemy engines manage connections
   - Reuses connections efficiently

Future Enhancement Opportunities

1. **Caching Layer**: Cache frequent queries to reduce LLM calls
2. **Streaming Responses**: Stream answer generation for better UX
3. **Multi-turn Context**: Improve conversation context handling
4. **Advanced Analytics**: Add more detailed metrics and trends
5. **Source Quality Scoring**: Rank sources by relevance/quality
6. **User Authentication**: Add user accounts and query history
7. **Export Functionality**: Allow exporting conversations and analytics

---