Project Explanation

Overview

The **Insurance Agent** project is a full-stack AI-driven application that provides auto and home insurance quotes through a conversational interface. It combines a FastAPI backend, a React/Vite frontend, and a LangGraph-based workflow (the "brain") that orchestrates LLM reasoning, tool calls, and RAG search.

Repository Structure

```
insurance_agent/
│
├─ backend/                  # FastAPI server and AI logic
│   ├─ main.py               # API endpoints, session handling
│   ├─ memory.py             # Simple in-memory store for session data
│   ├─ provider.py           # Gemini provider abstraction
│   ├─ langgraph_agent.py    # LangGraph workflow definition
│   ├─ rag_system.py         # Chroma vector store + Gemini embeddings
│   ├─ document_analyzer.py # Gemini Vision document analysis
│   ├─ requirements.txt
│   ├─ test_*.py             # Unit & integration tests
│   └─ __init__.py
│
├─ frontend/                 # React UI built with Vite
│   ├─ src/
│   │   ├─ components/ChatInterface.jsx  # Chat UI component
│   │   ├─ App.jsx
│   │   └─ main.jsx
│   ├─ index.html
│   ├─ index.css           # Custom palette, Google Font "Inter"
│   ├─ package.json
│   └─ vite.config.js
│
└─ .gemini/brain/           # Artifacts generated during development
    ├─ implementation_plan.md
    ├─ task.md
    ├─ walkthrough.md
    └─ agent_architecture.md
```

Key Technologies

- **FastAPI** + **Uvicorn** – backend web framework and server.
- **React** + **Vite** – modern, fast frontend development.
- **Google Gemini 1.5 Flash** (`gemini-2.0-flash-exp`) – LLM for reasoning and vision.
- **LangGraph** – visual state-graph workflow (nodes, edges, conditional routing).
- **Chroma DB** + **Gemini embeddings** – vector store for Retrieval-Augmented Generation (RAG).
- **LangChain** – tool orchestration and message handling.

- **dotenv** – loads the `GEMINI_API_KEY` from `.env`.
- **Python-multipart** – handles file uploads for document analysis.
- **Testing** – `unittest` + FastAPI `TestClient` for API and LangGraph tests.

How It Works

1. **User** types a message in the React UI → POST `/api/chat`.
2. **FastAPI** stores the message in an in-memory session dict.
3. **(Future)** `main.py` forwards the session state to `agent_graph.invoke(state)`.
   - The graph runs **gather_info**, decides whether to **search_knowledge**, **calculate_quote**, or loop for more info.
   - Tool nodes call the premium-calculation functions.
4. If a knowledge query is detected, **RAG** (`rag_system.search_knowledge`) retrieves relevant docs from Chroma and injects them into the prompt.
5. **Gemini** generates the final answer, which is returned to the UI.
6. The UI displays the response; screenshots of the interaction are saved in the artifact folder for the walkthrough.

Testing & Verification

- Unit tests for memory, provider, and API endpoints.
- `test_langgraph_brain.py` validates the LangGraph state transitions.
- End-to-end UI tests (via browser sub-agent) captured screenshots for refusal and auto-insurance flows.

This file provides a concise, printable summary of the project for documentation or workshop hand-outs.