AltoGPT Building Assistant - Technical Documentation

Overview

AltoGPT is an AI-powered building assistant system that combines multi-agent orchestration, real-time room sensor data analytics, and language model interfaces to deliver actionable, context-aware recommendations. It features a responsive frontend built with React, a FastAPI backend powered by LangGraph agents, and is containerized for modular deployment.

Frontend

Technology Stack

- React (Vite) for fast dev experience and component-based UI
- Tailwind CSS for responsive styling
- Axios for backend communication

Features

- User can input natural language queries (e.g., "What's the CO_2 in Room101?")
- Clear(with alert to confirm user selection) & Submit buttons
- Dynamic AI response section

Mockup

Backend

Stack

- FastAPI for high-speed, RESTful interface
- LangGraph + LangChain for multi-agent logic
- Pandas for structured sensor data querying

Purpose & Justification

- FastAPI provides async, production-ready backend routing
- LangGraph manages state transitions between agents, supporting extensibility and memory
- LangChain agents abstract logic for query understanding, data handling, and summarization

Agents in LangGraph

1. Keyword Agent (inside Orchestration)

- Extracts room, operation, and datetime (optional) from user queries.
- Falls back on chat_history for continuity.
- Uses structured prompt templates with LLM to ensure correct keyword classification.
- Handles time_specific operation by triggering a follow-up date/time parsing step.

2. Sensor Agent

- Loads CSV datasets using Pandas for CO_2, temperature, humidity, power, and presence.
- Supports 4 modes: `average`, `sum`, `latest`, `time_specific`.
- Performs fuzzy nearest-time matching for `time_specific`, with a threshold of ±1 hour.
- Flags status failure when data is out of range or unsupported.
- Returns a complete summary structure for IAQ, power, and occupancy.

3. Tool Agent

- Estimates daily energy cost from `avg_power_watts`.
- Flags warnings for high usage (threshold: $50/day).
- Injects alerts or fallback messages into the system state.

4. Rag Agent

- Implements a static retrieval-based knowledge system.
- Returns best-practice guidelines for CO_2, temperature, humidity, etc.
- Simulates external documents (e.g., ASHRAE) using a local dictionary.

5. Answer Agent

- Aggregates all structured sensor inputs and guidelines into a readable format.
- Appends alert-based feedback (e.g., high CO_2).
- Clearly indicates room status, power consumption, and occupancy.
- Separates analysis and guideline section for user clarity.
- Returns the final textual analysis to be displayed to the user

6. Chat Agent

- Gathers outputs from all prior agents.
- Formats the final user-facing response using LangChain prompt templating.
- Considers chat_history context.
- Flags errors for unknown room/operation/time.
- Handles failure states directly with fallback response.

7. Orchestration Logic

The orchestration is powered by LangGraph's conditional state logic:

- If `SensorAgent` fails → skip `ToolAgent` and `RAGAgent` → go to `ChatAgent`
- Otherwise: follow the full pipeline → ChatAgent finalizes the answer

Feature	How It Works	Why It's Important
Room Detection	LLM parses room from natural language	Flexible input instead of hardcoded selection
Operation Type	LLM classifies queries as average, latest, time-specific, or sum	Adapts response style to user need
Timestamp support	KeywordAgent detects phrases like "2 hours ago"	Enables historical querying
Chat Memory	Maintains recent history for continuity	Simulates natural back-and-forth interactions
Energy Tool	ToolAgent estimates energy usage + alerts	Actionable feedback ("Consider reducing HVAC")
IAQ Summaries	SensorAgent aggregates or retrieves data	Core to building analytics
RAG Explanation	RAGAgent returns standards	Justifies AI decisions with references

Triton Inference (Future Integration)

Use **NVIDIA Triton Inference Server** to serve a local language model like **Phi-2**.

```
# llm = HuggingFacePipeline.from_model_id(
#     model_id="microsoft/Phi-3-mini-4k-instruct",
#     # model_id="microsoft/phi-2",
#     task="text-generation",
#     device=0,
#     batch_size=1,
#     pipeline_kwargs={
#         "max_new_tokens": 50,
#         # "top_k": 50,
#         # "temperature": 0.1,
#     },
# )
```

Why Phi-2

- Open source
- Lightweight model with good performance for on-device inference
- Good model at lower resource requirements (6-12 B)
- Not done as too low GPU → high latency during inference

Why Triton

- Fast ONNX runtime for GPU-accelerated LLM
- Production-ready with HTTP/gRPC interfaces
- Compatible with LangChain and FastAPI

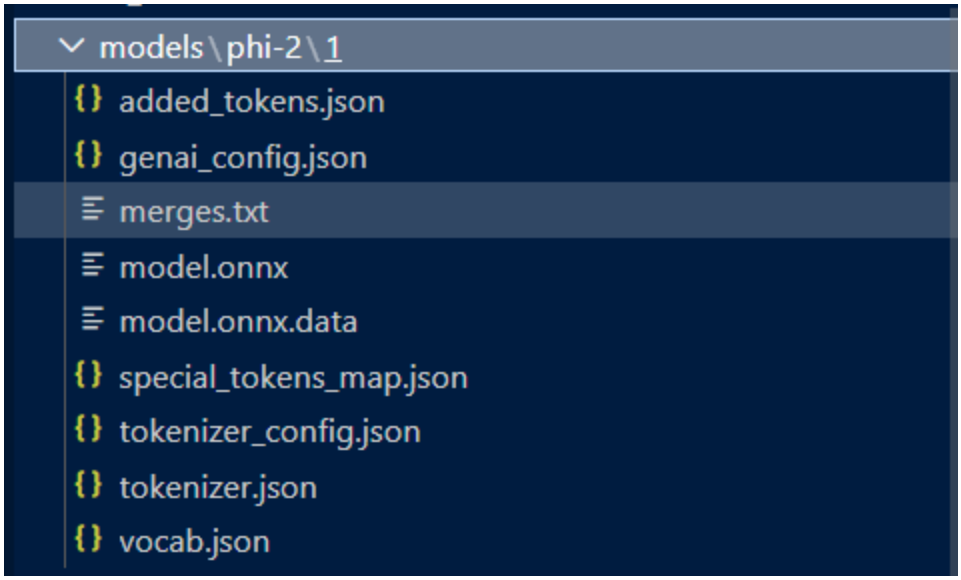
Integration Plan

1. Export model to ANXX using optimum

   In separate Agent

   (See ll_triton.py in code)

2. Create a model repository for Triton



3. Run Triton via Docker with GPU

```
# triton:
#   build:
#     context: ./triton
#     args:
#       HUGGINGFACE_HUB_TOKEN: ${HUGGINGFACE_HUB_TOKEN}
#   runtime: nvidia
#   deploy:
#     resources:
#       reservations:
#         devices:
#           - driver: nvidia
#             capabilities: [gpu]
#             count: 1
#   volumes:
#     - ./.env:/opt/.env
#     - ./hf_cache:/hf_cache
#     - ./models:/models
#   ports:
#     - "8000:8000"  # HTTP
#     - "8001:8001"  # gRPC
#     - "8002:8002"  # Metrics
```

4. Replace llm.invoke with Triton HTTP call from the backend

   In Orchestration.py

```
# TRITON_URL = os.environ.get("TRITON_URL", "http://localhost:8000")

# llm = TritonPhi2Client(triton_url=TRITON_URL)
```

Why Triton was not implemented

- No support for Docker Desktop in Windows

The current model is Gemini

- Open source
- Low latency
- Can load faster with low GPU availability

Databricks

Why use?

- Easily scalable
- Schema aware tables
- RAG at Scale due to Unity Catalogue
- Model Hosting and tracking with MLFlow
- CI/CD ready due to notebook jobs

Integration Plan

- Upload the csv as Delta table
- Replace load function with Databricks SQL
- Add Databricks integration via token
- Migrate LLM calls to MLFlow

Not Used as?

- No Linux-based environment
- No cloud GPU access

Deployment and CI/CD

Preferred Setup

- Use Docker Compose to run:
- React frontend (port 3000)
- FastAPI backend (port 8000)
- Triton inference (port 8001) (optional)

CI/CD

Use of Github Action workflow to:

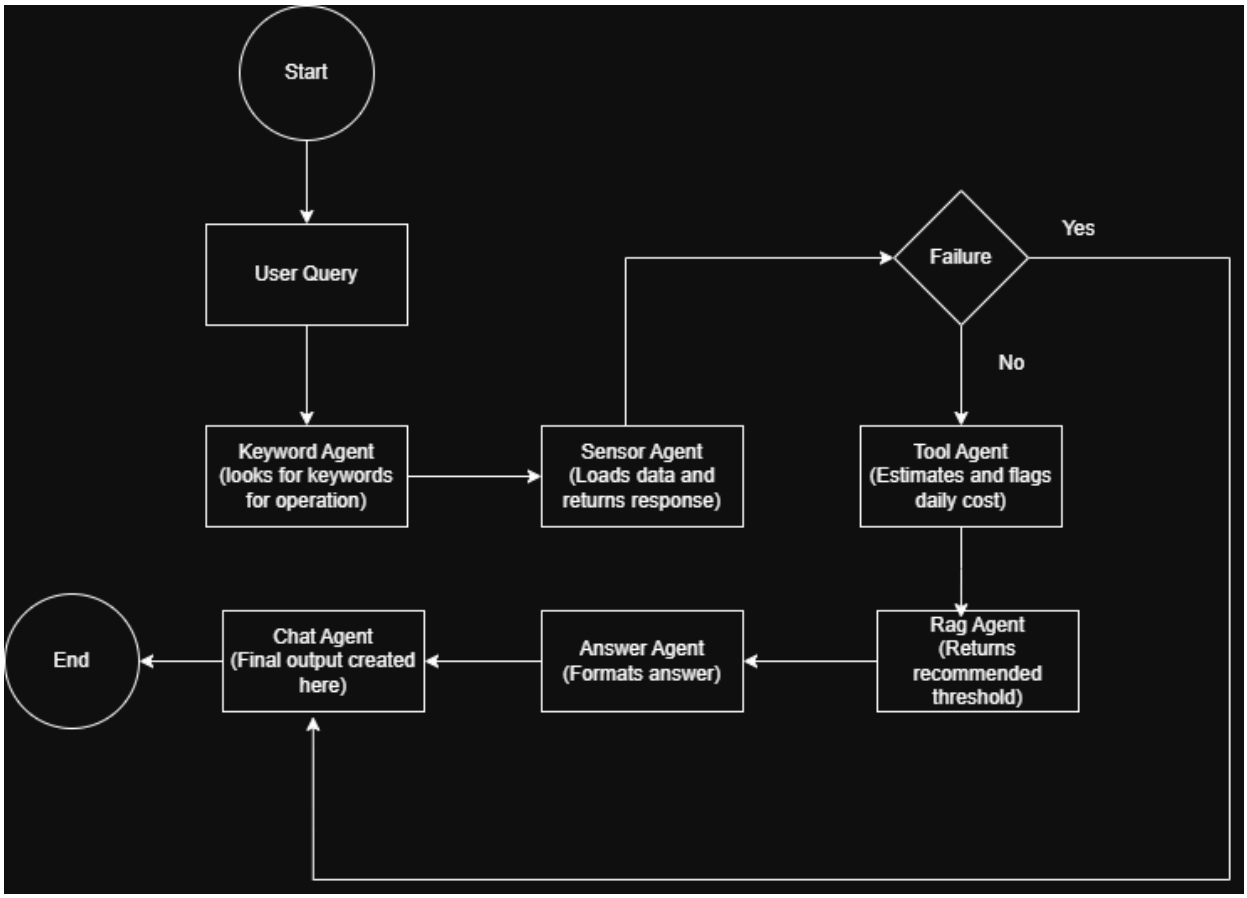
- Install packages
- Docker build and pushing of the image to container registry
- Deployment via cloud

Not implemented as:

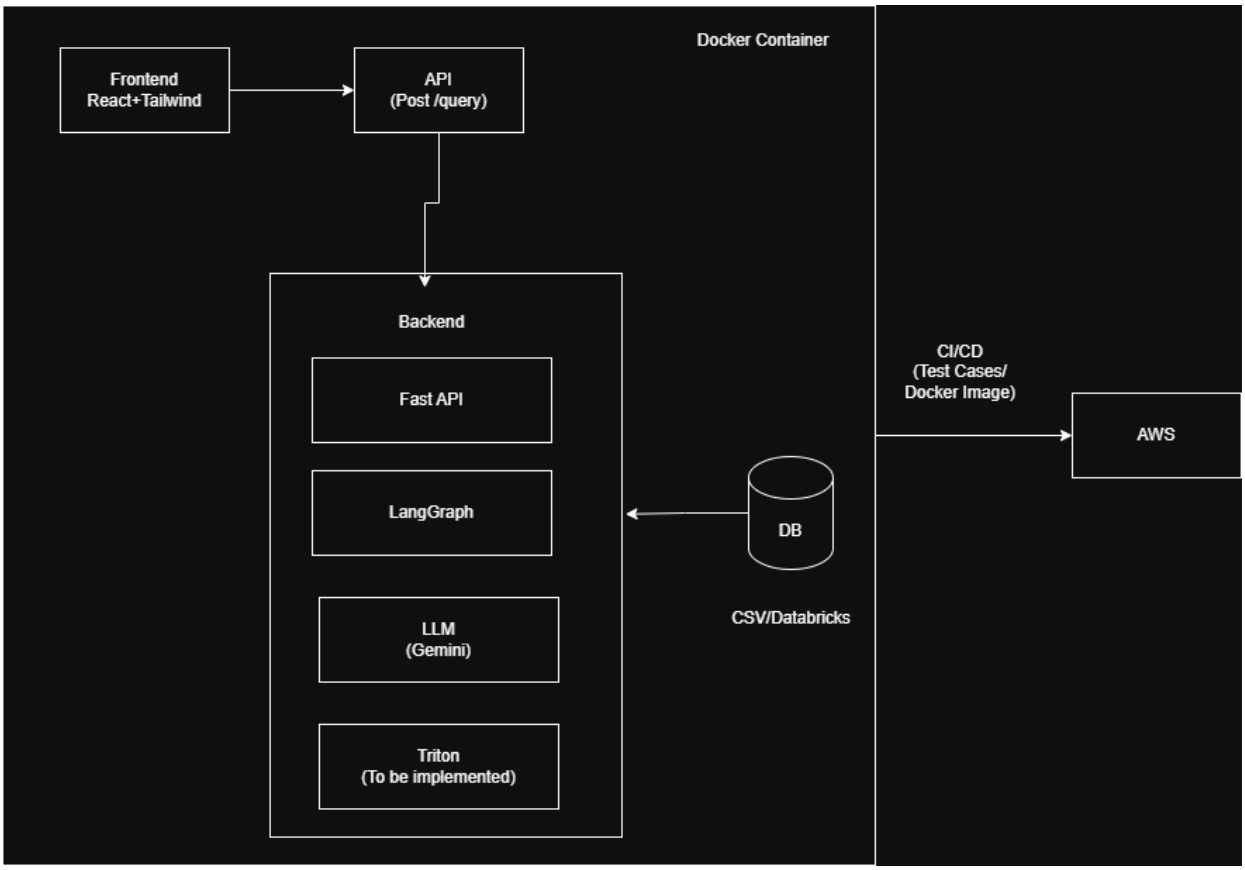
- No GPU-enabled workflow action in github in free subscription

Visual Diagram

Agent Flow Diagram



Overall Architecture



Conclusion

AltoGPT is a scalable and modular AI assistant framework tailored for building data analytics. While GPU inference and cloud deployment are future goals, the current implementation demonstrates:

- Natural language interaction with sensor systems
- Modular agents coordinating via LangGraph
- Prompt design via Langchain
- Custom tool logic for energy insights

With GPU + cloud integration, this system is production-ready for smart building management and beyond.

Env Files

```
GEMINI_API_KEY=AIzaSyBkE3dgJ_6jFQyvR88X68fYV_vDL4VYu9I
HUGGINGFACE_HUB_TOKEN=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
```