

Clinical Trials Multi-Agent System

- Isha Raju (CISC 691 – A06)

Table of Contents

| | |
|--|----------|
| GITHUB LINK..... | 1 |
| PURPOSE AND PROBLEM STATEMENT | 1 |
| AGENT OVERVIEW AND CAPABILITIES..... | 2 |
| TASK ASSIGNMENT AGENT | 3 |
| CLINICAL TRIALS FINDER AGENT..... | 3 |
| CLINICAL ASSISTANT AGENT (RARE DISEASE FOCUS) | 3 |
| DESIGN PATTERN, ARCHITECTURE, AND FRAMEWORK..... | 4 |
| DESIGN PATTERN: | 4 |
| FRAMEWORK & TOOLS INTEGRATION..... | 4 |
| WORKFLOW..... | 4 |
| WORKFLOW DIAGRAM – OVERALL | 5 |
| WORKFLOW DIAGRAM – CLINICAL TRIAL FINDER AGENT | 6 |
| WORKFLOW DIAGRAM - CLINICAL ASSISTANT AGENT | 6 |
| CHALLENGES AND SOLUTIONS..... | 6 |
| BENEFITS AND VALUE | 7 |
| REFLECTION AND FUTURE WORK..... | 7 |
| CURRENT STATUS: | 7 |
| HOW AI WAS USED IN THIS PROJECT..... | 8 |
| APPENDIX:..... | 9 |
| SOME SNIPPETS OF MY CONVERSATIONS WITH CHATGPT TO RESOLVE ISSUES TO RUN THE MODULES ERROR FREE. .. | 9 |

Github Link

<https://github.com/isharaju/Multiagent-clinical-system/tree/main>

Purpose and Problem Statement

The Clinical Trials Multi-Agent System was conceived to address pressing issues in modern healthcare access and medical research navigation. One of the primary problems in healthcare is the difficulty patients and caregivers face in identifying clinical

trials suitable for their specific conditions. Clinical trials represent critical opportunities for patients, particularly those suffering from rare or advanced diseases, to gain access to new treatments and potentially life-saving therapies. Despite the existence of registries like ClinicalTrials.gov, the complexity of these platforms often poses a barrier. Patients must sift through dense medical terminology, unstructured eligibility criteria, and poorly organized trial listings, all while attempting to match themselves against eligibility requirements without adequate clinical knowledge.

In addition to this, rare disease patients and their families face even more substantial challenges. Rare diseases, such as Spinocerebellar Ataxia (SCA), affect a small proportion of the population and often receive limited attention in mainstream medical research. The resources available are scattered across multiple journals, medical databases, and niche communities, making it nearly impossible for patients to stay informed about recent developments or emerging therapies. Clinicians too, constrained by time, find it impractical to continuously track updates for each individual patient's rare condition.

The proposed multi-agent system addresses these issues by leveraging artificial intelligence to create an accessible, intelligent interface that can:

- Automate the search for active, geographically feasible clinical trials.
- Parse complex eligibility criteria using advanced natural language processing.
- Provide evidence-based summaries of rare disease research, empowering patients and clinicians with up-to-date information.

By building a bridge between highly technical medical data and layperson comprehension, this system reimagines how technology can democratize access to healthcare knowledge.

Agent Overview and Capabilities

The Clinical Trials Multi-Agent System is composed of multiple specialized agents, each tailored to handle distinct functions within the healthcare information ecosystem. Rather than relying on a monolithic AI model, the system's architecture divides responsibilities across focused agents, ensuring higher efficiency, explainability, and adaptability.

At its core, the system uses a Task Assignment Agent to serve as a central coordinator. This agent interprets incoming user queries and determines whether the request relates to clinical trial discovery or rare disease information retrieval. Once the intent is classified, the query is forwarded to either the Clinical Trials Finder Agent or the Clinical Assistant Agent, both of which are optimized for their respective domains.

This modular design not only allows for clearer logical separation but also supports future scalability. For instance, additional agents for other healthcare services such as

insurance navigation, prescription support, or diagnostic assistance can be added without overhauling the core architecture.

Task Assignment Agent

The Task Assignment Agent is responsible for interpreting natural language input from users. For example, a query such as “Are there any ongoing clinical trials near me for Parkinson’s disease?” would be recognized as a clinical trial-related request and routed accordingly. Alternatively, a question like “What are the latest treatments for Spinocerebellar Ataxia?” would be identified as a research inquiry and directed to the Clinical Assistant Agent.

This agent employs a combination of natural language understanding and context-aware reasoning powered by a language model. It ensures that ambiguous or mixed queries are disambiguated and routed correctly. By centralizing this decision-making process, the system ensures that users do not need to explicitly specify their intent.

Clinical Trials Finder Agent

The Clinical Trials Finder Agent is central to solving the problem of trial discovery. Once activated, it first extracts structured parameters from the user's query, such as age, gender, condition, and ZIP code. It then queries ClinicalTrials.gov for relevant studies using these filters.

However, raw trial data is rarely user-friendly. Eligibility criteria, typically written in dense medical language, are processed using the Mistral language model. This model reinterprets eligibility text into simplified summaries, clarifying whether a user is likely to qualify. Additionally, geolocation data is processed through geopy to calculate distances between the user's ZIP code and trial sites, enabling ranking by proximity.

The agent concludes its process by presenting a ranked, annotated list of trials including key information: study purpose, eligibility highlights, location, and contact details. This automation not only accelerates discovery but also reduces cognitive load for patients who would otherwise have to manually parse each trial entry.

Clinical Assistant Agent (Rare Disease Focus)

The Clinical Assistant Agent focuses on delivering targeted support for rare diseases. Spinocerebellar Ataxia (SCA) was chosen as the initial focus due to its complexity and limited mainstream resources. This agent connects to PubMed and other research databases, retrieving recent publications relevant to the query.

Once data is retrieved, the agent employs Retrieval-Augmented Generation (RAG) techniques to extract treatment options, highlight clinical findings, and provide easy-to-understand summaries. It also references reputable sources, ensuring credibility. Beyond static research retrieval, this agent can answer specific questions, such as “What physical therapy methods are recommended for SCA?” or “Are there any

promising drug trials reported this year?”

Moreover, the agent maintains a persistent memory of user interactions. For instance, if a user has previously asked about symptom management, the agent can contextualize future queries, leading to personalized and efficient responses.

Design Pattern, Architecture, and Framework

Design Pattern:

- Retrieval-Augmented Generation (RAG) combined with modular tool-using agents.
- LLM-driven reasoning enhances structured API queries and eligibility parsing.

Architecture:

Hybrid modular architecture:

- UI Layer: Streamlit.
- API Layer: FastAPI for communication and scalability.
- Orchestration Layer: LangChain and LangGraph for multi-agent coordination and memory.
- Reasoning Layer: Mistral LLM for local inference.
- Output Layer: Summarized results presented to the user via Streamlit.

Framework & Tools Integration

The integration of Streamlit, FastAPI, LangChain, LangGraph, and Mistral is pivotal:

- **Streamlit (UI):** User-facing interface for capturing inputs and displaying outputs.
- **FastAPI (API):** Middleware for handling requests/responses and linking the UI with backend agents.
- **LangChain:** Enables tool integration and retrieval-augmented generation workflows.
- **LangGraph:** Manages agent orchestration, memory retention, and decision routing.
- **Mistral LLM:** Local inference engine for private, on-device reasoning and summarization.
- **Geopy** for geolocation and distance ranking.
- **ChromaDB** for vector-based research retrieval.

This layered approach ensures modularity, scalability, and privacy-focused AI workflows.

Workflow

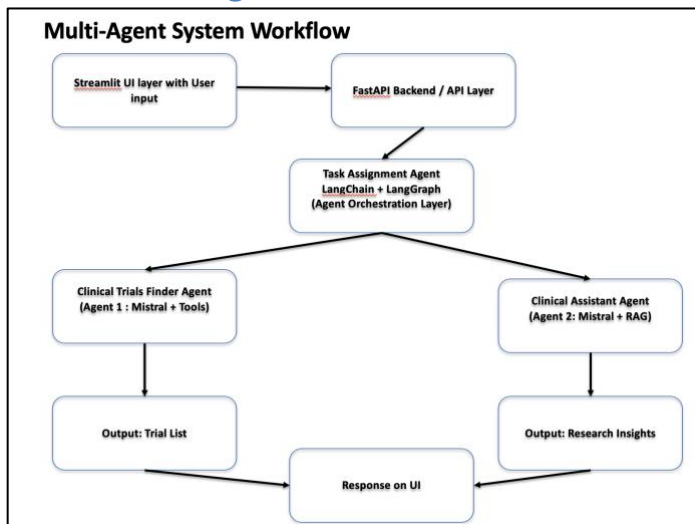
Input: User query (form fields or natural language).

Flow:

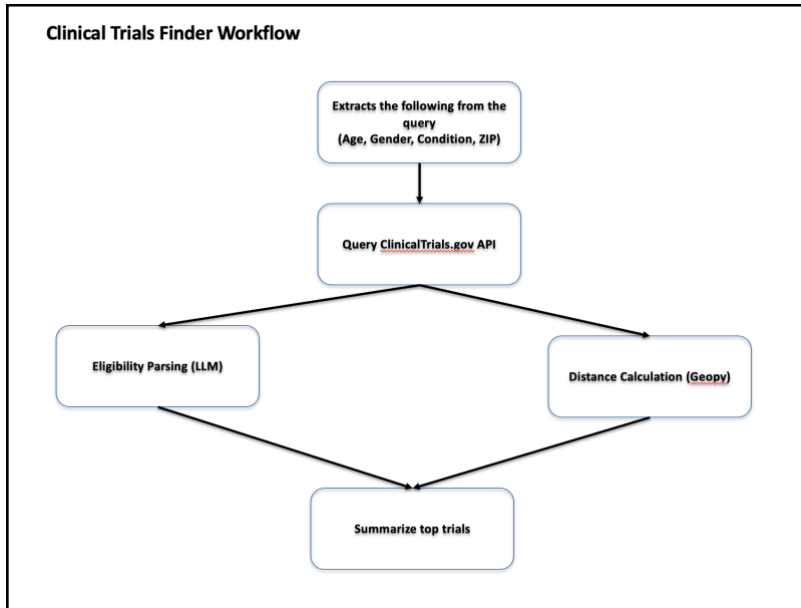
1. Task Assignment Agent classifies query type.
2. For clinical trials:
 - Query ClinicalTrials.gov.
 - Parse eligibility via LLM prompt chaining.
 - Calculate proximity using geopy.
 - Summarize top trials.
3. For rare disease research:
 - Retrieve PubMed papers using ChromaDB.
 - Summarize research and extract treatments.
 - Provide answers with citations.

Output: Patient-friendly trial list or research summaries displayed via UI.

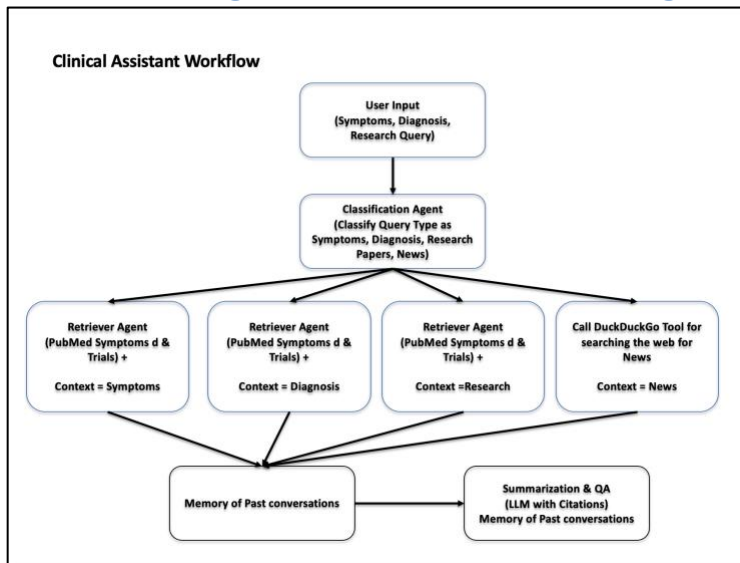
Workflow Diagram – Overall



Workflow Diagram – Clinical Trial Finder agent



Workflow Diagram - Clinical Assistant agent



Challenges and Solutions

| Challenge | Resolution |
|----------------------------------|---|
| Ambiguous trial eligibility text | Use prompt-chained LLM parsing for nuanced criteria extraction. |
| Memory-heavy local model loading | Optimize using quantized Mistral-7B and reduced inference token limits. |
| Sparse location data in trials | Add fallback logic to filter and prioritize valid geolocation entries. |

| | |
|-----------------------|--|
| Complex orchestration | Adopt LangGraph for structured stateful workflows with memory. |
|-----------------------|--|

Benefits and Value

- Clear separation of concerns across UI, API, orchestration, and reasoning layers.
- Enhanced patient experience through simplified interfaces and personalized outputs.
- Scalable architecture supporting additional agents and integrations.
- Local inference ensures privacy and compliance readiness (HIPAA).
- RAG-driven reasoning improves accuracy in medical contexts.

Reflection and Future Work

Future improvements will focus on:

- Implementing HIPAA-compliant data encryption and security.
- Expanding rare disease coverage and integrating wearable IoT health data.
- Introducing proactive notifications (e.g., email/SMS for trial updates).
- Adding reinforcement learning for feedback-driven trial ranking.
- Enhancing memory persistence for longitudinal patient engagement.

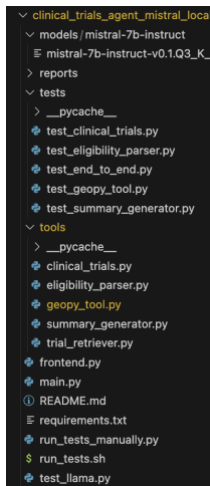
Current Status:

I have implemented the RAG pipeline (A04) of Agent 2.

And the below modules and tests of Agent 1:

```
llama_new_context_with_model: n_ctx      = 2048
llama_new_context_with_model: n_batch    = 512
llama_new_context_with_model: n_ubatch   = 512
llama_new_context_with_model: flash_attn = 0
llama_new_context_with_model: freq_base  = 10000.0
llama_new_context_with_model: freq_scale = 1
llama_kv_cache_init:      CPU KV buffer size = 256.00 MiB
llama_new_context_with_model: KV self size  = 256.00 MiB, K (f16): 128.00 MiB, V (f16): 128.00 MiB
llama_new_context_with_model: CPU output buffer size = 0.12 MiB
llama_new_context_with_model: CPU compute buffer size = 164.01 MiB
llama_new_context_with_model: graph nodes   = 1030
llama_new_context_with_model: graph splits  = 514
AVX = 1 | AVX_VNNI = 0 | AVX2 = 1 | AVX512 = 0 | AVX512_VBMI = 0 | AVX512_VNNI = 0 | AVX512_BF16 = 0 | FMA = 1 | NEON = 0 | SVE = 0 | ARM_FMA = 0 | F16 = 1 | FP16_VA = 0 | WASM_SIMD = 0 | BLAS = 1 | SSE3 = 1 | SSSE3 = 1 | VSX = 0 | MATMUL_INT8 = 0 | LLAMAFILE = 1 |
Model metadata: {'general.quantization_version': '2', 'tokenizer.ggml.unknown_token_id': '0', 'tokenizer.ggml.eos_token_id': '2', 'tokenizer.ggml.bos_token_id': '1', 'tokenizer.ggml.model': 'llama', 'llama.attention.head_count_kv': '8', 'llama.context_length': '32768', 'llama.attention.head_count': '3', 'llama.rope.freq_base': '10000.000000', 'llama.rope.dimension_count': '128', 'general.file_type': '12', 'llama.feed_forward_length': '14336', 'llama.embedding_length': '4096', 'llama.block_count': '32', 'general.architecture': 'llama', 'llama.attention.layer_norm_rms_epsilon': '0.000010', 'general.name': 'mistralai.mistral-7b-instruct-v0.1'}
Using fallback chat format: llama-2
[✓] Mocked trials query successful.
[✓] Trials query passed.
[✓] Eligibility parsing passed.
[✓] Distance ranking passed.
[✓] Summary generation passed.

[✓] ALL TESTS PASSED SUCCESSFULLY! [✓] [✓]
[✓] Passed: tests.test_end_to_end
```



The UI, integration and orchestration of all the components is in progress.

How AI Was Used in This Project

Artificial Intelligence was deeply integrated into every phase of this project, serving as a catalyst for creativity, decision-making, and problem-solving (I specifically used ChatGPT o1 model). From the earliest stages of conceptualization through to implementation and debugging, AI was not simply an auxiliary tool; it functioned as a co-pilot, guiding the development of the Clinical Trials Multi-Agent System.

The journey began with ideation. AI was invaluable in helping generate and refine ideas for the multi-agent framework. By engaging in iterative discussions with AI-driven systems, the scope of the project was clarified. AI provided insights into pressing challenges in healthcare, such as clinical trial accessibility and the fragmented nature of rare disease research. This process shaped the system's primary objectives and ensured that it addressed meaningful, real-world problems.

As the project moved into the planning phase, AI offered structured, step-by-step guidance for designing the architecture and workflow. It provided advice on dividing the system into layers—such as the user interface, backend API, agent orchestration, and reasoning components—ensuring a clean separation of concerns and modularity. This foundation allowed for a well-organized, scalable design that could accommodate future enhancements and integrations.

Framework and tool selection was another area where AI played a crucial role. Through AI-assisted comparisons of different technologies, informed choices were made: FastAPI was selected for backend services due to its speed and simplicity, LangChain and LangGraph were chosen for orchestrating multi-agent interactions, Streamlit was adopted for building an intuitive and interactive UI, and Mistral was identified as the LLM best suited for local inference tasks. AI's input in this decision-making ensured a tech stack that was both

modern and well-aligned with the system's requirements.

AI's involvement extended significantly into the implementation phase. It was used extensively for debugging, whether that involved diagnosing errors in API integration, resolving environment configuration issues, or optimizing model inference for better performance. Rather than relying solely on manual trial and error, AI accelerated the debugging process by suggesting precise solutions and explaining error messages in detail, thereby improving both development speed and code quality.

Ultimately, AI served as an end-to-end collaborator, guiding creative brainstorming, technical design, framework selection, and hands-on coding support. This pervasive integration of AI transformed the project from a traditional development effort into a streamlined, highly iterative workflow that combined human oversight with AI-driven intelligence. By doing so, it not only shortened development time but also significantly improved the rigor, efficiency, and sophistication of the final system.

Appendix:

Some snippets of my conversations with ChatGPT to resolve issues to run the modules error free.

Challenges

1. For some reason my entire miniconda and environments got wiped out.
2. Reinstalled everything .
3. Used ChatGPT to design the architecture and workflow.
4. Used CHatGPT to provide a template starter code.
5. Created tests for all the .py files in the tools folder.
6. The tests failed in the first round.
7. There was an error with connecting to the Clinicaltrials gov API.

Added debugging logic as follows

1 AssertionError in `test_clinical_trials.py`

- The test expects `"StudyFieldsResponse"` in the response from `query_trials`.
- This means the API call likely failed, returned an error, or produced an unexpected result.

✅ Fix:

Add debug logging:

python

Copy Edit

```
def test_query_trials():
    data = query_trials("I am looking for Parkinson's trials in Boston")
    print("Response from API:", data) # <-- debug line
    assert "StudyFieldsResponse" in data, "Missing StudyFieldsResponse key"
```

If the API returns an error or empty response, check your network and whether `ClinicalTrials.gov` API endpoint is reachable.

Got a 403 error: Below is the response from Chatgpt.

This `403 Forbidden` error indicates that **the ClinicalTrials.gov API is blocking your request**. This typically happens due to one of these reasons:

✅ Why 403 Happens

1. Missing or improper request headers (e.g., `User-Agent`).

- Some APIs block requests that don't include a valid `User-Agent` (common when using `httpx` or `requests` without headers).

2. Temporary API restrictions or location-based blocking.

3. Rate limiting if you've sent multiple requests quickly.

The **ClinicalTrials.gov API is public and open** — it doesn't require authentication or API keys. The 403 Forbidden issue is typically related to:

✅ **Missing User-Agent header** (most common) ✅ Network/firewall restrictions (corporate networks sometimes block APIs) ✅ Temporary API rate-limiting

Updated `tools/clinical_trials.py` to include a **User-Agent header**:

Fix: Add Headers

Update `tools/clinical_trials.py` to include a **User-Agent** header:

```
python                                                                    Copy Edit

import httpx

def query_trials(input_text: str) -> str:
    import re
    condition = re.findall(r"for (.*) within", input_text)
    location = re.findall(r"in (.*) with", input_text)
    condition = condition[0] if condition else "cancer"
    location = location[0] if location else "Boston, MA"

    base_url = "https://clinicaltrials.gov/api/query/study_fields"
    params = {
        "expr": f"{condition} AND recruiting",
        "fields": "NCTId,BriefTitle,LocationCity,LocationState,OverallStatus,Eligibilit
        "min_rnk": 1,
        "max_rnk": 10,
        "fmt": "json"
    }

    headers = {
        "User-Agent": "ClinicalTrialsAgent/1.0 (contact: your_email@example.com)"
    }

    try:
        response = httpx.get(base_url, params=params, headers=headers, timeout=10)
        response.raise_for_status()
        return response.json()
    except Exception as e:
        return f"Error querying trials: {e}"
```

The API endpoint was old

Correct ClinicalTrials.gov API Endpoint

ClinicalTrials.gov recently **updated their API** as part of their modernization effort.

The **new base endpoint** is:

```
bash                                                                    Copy Edit

https://clinicaltrials.gov/api/v2/studies
```

The new endpoint worked fine on the browser and I could see a json file. However it did not work with my code. So updated my code for it

Ah, this explains it:

- The **API call works fine in the browser** and returns JSON (✅), but in your code it's returning `403 Forbidden`.
- This means **the request from Python is being rejected by the ClinicalTrials.gov API, likely due to how the query is formatted or missing headers.**

🔍 What's Happening

The test query string is sending:

```
rust
```

📄 Copy 🗑️ Edit

```
query.term=I am looking for Parkinson's trials in Boston
```

This natural language format is **too verbose** and contains special characters ('), which may trigger a block (or misinterpretation) on the API side.

1 Sanitize Input

Instead of passing the full natural language query, **extract just the condition** ("Parkinson's") and optionally the location.

Example fix in `query_trials()`:

```
python Copy Edit

def query_trials(input_text: str) -> dict:
    import re
    # Extract condition (simplify NLP for now)
    match = re.search(r"(Parkinson's|Alzheimer's|cancer|diabetes)", input_text, re.IGNORECASE)
    condition = match.group(1) if match else "cancer"

    base_url = "https://clinicaltrials.gov/api/v2/studies"
    params = {
        "format": "json",
        "filter.overallStatus": "RECRUITING",
        "query.term": condition,
        "pageSize": 10,
    }
    headers = {"User-Agent": "ClinicalTrialsAgent/1.0 (contact: your_email@example.com)"}

    try:
        response = httpx.get(base_url, params=params, headers=headers, timeout=10)
        response.raise_for_status()
        return response.json()
    except Exception as e:
        return f"Error querying trials: {e}"
```

This was resolved. And then I got an error for `eligibility_criteria`.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

✓ Passed: `tests.test_clinical_trials`

▶ Running: `tests.test_eligibility_parser`

A module that was compiled using NumPy 1.x cannot be run in NumPy 2.3.2 as it may crash. To support both 1.x and 2.x versions of NumPy, modules must be compiled with NumPy 2.0. Some module may need to rebuild instead e.g. with 'pybind11>=2.12'.


If you are a user of the module, the easiest solution will be to downgrade to 'numpy<2' or try to upgrade the affected module. We expect that some modules will need time to support NumPy 2.

/Users/isha/Desktop/HU_courses/Harrisburg/CISC_691_Next_GEN_AI/projects/clinical_trials_agent_mistral_local/models/mistral-7b-instruct/mistral-7b-instruct-v0.1.Q3_K_M.gguf

```
'ReformerTokenizer', 'RemBertTokenizer', 'RetriBertTokenizer',  
'RobertaTokenizer', 'RoFormerTokenizer', 'SeamlessM4TTokenizer',  
'SqueezeBertTokenizer', 'T5Tokenizer', 'UdopTokenizer',  
'WhisperTokenizer', 'XLMRobertaTokenizer', 'XLNetTokenizer',  
'SplinterTokenizer', 'XGLMTokenizer', 'LlamaTokenizer',  
'CodeLlamaTokenizer', 'GemmaTokenizer', 'Phi3Tokenizer']  
✖ Test failed: tests.test_eligibility_parser  
(env_a06) Ishas-MacBook-Pro:clinical_trials_agent_mistral_local  
isha$
```

You're now facing a **tokenizer conversion error** because your **local Mistral model directory** (`./models/mistral-7b-instruct`) is missing the `tokenizer.model` file that Hugging Face needs to load it properly.

Root Cause

- The Mistral model you downloaded is likely in **GGUF or another format** (e.g., llama.cpp) not directly compatible with Hugging Face `AutoTokenizer`.
- Hugging Face is attempting to **convert it using** `tiktoken`, but it fails because the necessary `tokenizer.model` (or vocab files) is not present. 

Original (Hugging Face) Code in `eligibility_parser.py`

python

Copy Edit

```
from transformers import pipeline, AutoTokenizer, AutoModelForCausalLM

# Load local Mistral model
model_path = "./models/mistral-7b-instruct"
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoModelForCausalLM.from_pretrained(model_path)
llm = pipeline("text-generation", model=model, tokenizer=tokenizer)
```

Replace With llama.cpp Loader

python

Copy Edit

```
from llama_cpp import Llama

# Load local Mistral model (GGUF format with llama.cpp)
llm = Llama(model_path="./models/mistral-7b-instruct/mistral-7b-instruct.Q4_K_M.gguf")

def generate_llm_response(prompt: str, max_tokens: int = 256) -> str:
    response = llm(prompt, max_tokens=max_tokens, echo=False)
    return response["choices"][0]["text"].strip()
```

For an error

This error is happening because `llama.cpp` with Metal acceleration is not compatible with your macOS Metal version or GPU setup. Specifically:

- The errors like `host_name attribute requires Metal language standard macos-metal2.2 or higher` indicate that your system's Metal API version is too low (older macOS or Intel GPU).
- On Intel-based Macs, Metal support is limited compared to Apple Silicon (M1/M2).
- The final error `ggml_metal_init: error: metal library is nil` confirms that llama.cpp failed to initialize Metal.

✓ How to Fix

You have two main options:

Option 1: Disable Metal (Run on CPU Only)

You can tell llama.cpp to **fall back to CPU**:

```
python                                                                    Copy Edit

from llama_cpp import Llama

llm = Llama(
    model_path="./models/mistral-7b-instruct/mistral-7b-instruct-v0.1.Q3_K_M.gguf",
    n_gpu_layers=0 # Force CPU mode, disables Metal
)
```

This did not work

Ah, if you've already forced `n_gpu_layers=0` and **still see Metal errors**, it means that the **llama.cpp backend is still attempting to initialize Metal** despite GPU being disabled. This is a known issue in some `llama-cpp-python` builds where the Metal backend is linked in by default.

✅ Solution: Force CPU-only build of llama-cpp-python

You need to reinstall `llama-cpp-python` without Metal support:

```
bash

pip uninstall llama-cpp-python
CMAKE_ARGS="-DLLAMA_METAL=OFF" FORCE_CMAKE=1 pip install llama-cpp-python --no-cache-dir
```

This explicitly disables the Metal backend and ensures CPU-only inference.

`python -c "import llama_cpp; print(llvm_cpp.__file__)"`
Should fail (no llama_cpp found)

```
ggml_metal_init: error: metal library is nil
ggml_backend_metal_device_init: error: failed to allocate context
llama_init_from_model: failed to initialize the context: failed to initialize Metal backend
Traceback (most recent call last):
  File "/Users/ishah/Desktop/HU_courses/Harrisburg/CISC_691_Next_GEN_AI/projects/clinical_trials_agent_mistral_local/test_llama.py", line 5, in <module>
    llm = Llama(
          ~~~~~
  File "/Users/ishah/miniconda3/envs/env_a06/lib/python3.12/site-packages/llama_cpp/llama.py", line 395, in __init__
    internals.LlamaContext(
  File "/Users/ishah/miniconda3/envs/env_a06/lib/python3.12/site-packages/llama_cpp/internals.py", line 263, in __init__
    raise ValueError("Failed to create llama_context")
ValueError: Failed to create llama_context
% (env a06) Ishahs-MacBook-Pro:clinical_trials_agent_mistral_local isha$
```

🔥 Alternative (Safer): Install Official CPU-Only Prebuilt Wheel

Instead of compiling locally, you can install the CPU-only wheel directly:

```
bash

pip install llama-cpp-python==0.2.90 --no-cache-dir --force-reinstall
```

Then test again with the Python snippet.

```
$ run_tests.sh  eligibility_parser.py  test_llama.py x
clinical_trials_agent_mistral_local > test_llama.py > ...
1  from llama_cpp import Llama
2
3  print("✅ Import worked - CPU only mode")
4
5  llm = Llama(
6      model_path="./models/mistral-7b-instruct/mistral-7b-instruct-v0.1.Q3_K_M.gguf",
7      n_gpu_layers=0,
8      n_threads=8
9  )
10 print("✅ Llama initialized successfully in CPU mode!")
11
```

```

llm_load_print_meta: EOS token      = 2 '</s>'
llm_load_print_meta: UNK token      = 0 '<unk>'
llm_load_print_meta: LF token       = 13 '<0x0A>'
llm_load_print_meta: max token length = 48
llm_load_tensors: ggml ctx size = 0.14 MiB
llm_load_tensors: offloading 0 repeating layers to GPU
llm_load_tensors: offloaded 0/33 layers to GPU
llm_load_tensors: CPU buffer size = 3355.27 MiB
.....
llama_new_context_with_model: n_ctx = 512
llama_new_context_with_model: n_batch = 512
llama_new_context_with_model: n_ubatch = 512
llama_new_context_with_model: flash_attn = 0
llama_new_context_with_model: freq_base = 10000.0
llama_new_context_with_model: freq_scale = 1
llama_kv_cache_init: CPU KV buffer size = 64.00 MiB
llama_new_context_with_model: KV self size = 64.00 MiB, K (f16): 32.00 MiB, V (f16): 32.00 MiB
llama_new_context_with_model: CPU output buffer size = 0.12 MiB
llama_new_context_with_model: CPU compute buffer size = 81.01 MiB
llama_new_context_with_model: graph nodes = 1030
llama_new_context_with_model: graph splits = 514
AVX = 1 | AVX_VNNI = 0 | AVX2 = 1 | AVX512 = 0 | AVX512_VBMI = 0 | AVX512_VNNI = 0 | AVX512_BF16 = 0 | FMA = 1 | NEON = 0 | SVE = 0 | ARM_FMA = 0 | F16C = 1 | FP16_VA = 0 | WASM_SIMD = 0 | BLAS = 1 | SSE3 = 1 | SSSE3 = 1 | VSX = 0 | MATMUL_INT8 = 0 | LLAMAFILE = 1 |
Model metadata: {'general.quantization_version': '2', 'tokenizer.ggml.unknown_token_id': '0', 'tokenizer.ggml.eos_token_id': '2', 'tokenizer.ggml.bos_token_id': '1', 'tokenizer.ggml.model': 'llama', 'llama.attention.head_count_kv': '8', 'llama.context_length': '32768', 'llama.attention.head_count': '32', 'llama.rope.freq_base': '10000.000000', 'llama.rope.dimension_count': '128', 'general.file_type': '12', 'llama.feed_forward_length': '14336', 'llama.embedding_length': '4096', 'llama.block_count': '32', 'general.architecture': 'llama', 'llama.attention.layer_norm_rms_epsilon': '0.000010', 'general.name': 'mistralai_mistral-7b-instruct-v0.1'}
Using fallback chat format: llama-2
✅ llama initialized successfully in CPU mode!

```

Finally Worked!

```

✅ llama initialized successfully!
✅ Passed: tests.test_eligibility_parser
=====
▶ Running: tests.test_geopy_tool
✅ Passed: tests.test_geopy_tool
=====

```

```

from tools.clinical_trials import query_trials
from tools.eligibility_parser import check_eligibility
from tools.geopy_tool import rank_by_distance
from tools.summary_generator import generate_summary

```

```

def test_end_to_end():
    # Inputs
    condition = "Parkinson's"
    location = "New York, NY"
    patient = {"age": 60, "gender": "male", "location": location}

    # Step 1: Query trials (returns dict with 'studies')
    trials_data = query_trials(f"condition:{condition} AND location:{location}")
    assert trials_data and isinstance(trials_data, dict), "Query trials should return a valid dict."
    assert "studies" in trials_data, "API response must contain 'studies'."

    # Extract list of studies
    trials = trials_data["studies"]

    # Step 2: Check eligibility
    eligibility = check_eligibility(trials, patient)
    assert isinstance(eligibility, list), "Eligibility check should return a list."
    assert all("eligible" in e for e in eligibility), "Each result must have an 'eligible' field."

    # Step 3: Rank by distance
    ranked_trials = rank_by_distance(eligibility, patient["location"])

```

```
assert isinstance(ranked_trials, list), "Ranked trials should be a list."
```

```
# Step 4: Generate summary
summary = generate_summary(ranked_trials)
assert isinstance(summary, str) and len(summary) > 0, "Summary should be a non-empty string."
```

```
print("Eligibility Result:", eligibility)
print("Ranked Trials:", ranked_trials)
print("Generated Summary:", summary)
```

✅ Option 2: Fix Live API Calls

If you still want real API tests:

- Add realistic `User-Agent` and backoff:

```
python                                                                    Copy Edit

headers = {
    "User-Agent": "Mozilla/5.0 (compatible; ClinicalTrialsAgent/1.0; +https://example.c
}
```

- Add a short `time.sleep(1)` between requests (API rate-limiting).
- Or cache results locally during tests.

🔥 Recommended Approach

Mock for tests, use **real API in production runs**.

This keeps your CI/CD fast and avoids unpredictable API failures (403, downtime).