SLIDE 1

As most of you already know, Large Language Models are constrained by **fixed context windows**. Whether it's 4K, 16K, or even 100K tokens — if the input is too long, older parts of the conversation get truncated, leading to loss of memory and coherence.

In enterprise settings, this becomes especially problematic — because client conversations, audit logs, or case threads are often large, multi-turn, and full of nested meaning.

The second challenge Is that LLMs aren’t naturally built to traverse nested structures like JSON data in APIs. They understand text, but not hierarchies unless explicitly guided.  
**How can we make LLMs remember and reason better — across long histories and structured information — without changing the model itself?**”

**✅ Language models like GPT, Claude, and LLaMA are fundamentally capable of semantic reasoning.  
They understand meanings, synonyms, intent, and even latent relationships between concepts — all through learned embeddings and their internal attention mechanisms.**

**But — and this is key — that semantic reasoning is largely passive. It enables them to understand language well, but not to act on it strategically in external systems.**

**❌ LLMs don’t naturally know how to traverse external APIs, navigate nested JSON structures, or explore live data schemas based on a goal.**

**❌ Likewise, when dealing with long histories — like ongoing chats or logs — LLMs don’t know which parts are most important for the current query.  
They’ll either take in too much (and get confused), or truncate blindly based on token limits, losing valuable information.**

So this calls for a 2-part solution.

The first part is called **Dynamic Context Compression**:

* It intelligently breaks a long chat history into manageable chunks
* Then it **ranks those chunks by semantic relevance** to the current question using TF-IDF
* And it uses the LLM to **summarize less-relevant parts**, so we can stay within the token limit without losing memory.

The second part is called **Semantic Navigation**:

* When given a complex JSON or API response, the system prompts the LLM to **choose which key to explore next**, based on the user’s intent.
* It doesn’t rely on schema knowledge or hardcoded logic.
* Instead, it uses **semantic inference** to traverse hierarchies — dynamically — until it finds the right value.

These two components together allow LLMs to act as intelligent, memory-aware, structure-savvy agents — while keeping everything model-agnostic and extensible.”

SLIDE 2

**SEMANTIC NAVIGATION WORKFLOW**

“Let me walk you through the **Semantic Navigation workflow**, because this is where things get interesting.

1. The user enters a query — for example: *“What is the capital of the country with the highest population?”*
2. The system passes this query and the current JSON structure to the LLM.
3. The LLM analyzes the **available keys**, and semantically chooses the most relevant one — not by string-matching, but by understanding latent meaning.
   * For example, it may correctly infer that ‘main\_office\_location’ is equivalent to ‘capital’.
4. If the value is itself another dict or a list, the system **recursively prompts** the LLM to choose again.
5. This continues until the final value is reached — and the system returns the answer *with a traceable path* of how it got there.

So we’re not just getting answers — we’re getting a reasoning trail.”

SLIDE 3

Now let’s zoom out and look at the bigger picture.

This approach is **domain-agnostic** — it can be used in finance to explore transaction logs, in supply chains to analyze JSON API flows, or in HR to track policy Q&A logs.

The UI is simple — natural language. No query language. No schema awareness.

And the model remains untouched — we’ve added value entirely at the **architecture level**, so it works with GPT, Claude, LLaMA, or any model we deploy internally.

That’s the key takeaway here:  
✅ We’ve taken what LLMs already do — semantic understanding —  
🔁 And turned it into a system that adds **semantic planning, navigation, memory compression, and traceability**.

This is the kind of modular, composable innovation that will power the next wave of **enterprise copilots** — and we’ve built the blueprint for it.”

CODE

**🗂️ 1. app.py — The Frontend Controller**

“This is the main **Streamlit app**. It ties everything together into a usable interface.  
Users can enter a query, adjust the compression budget, and either run a standard LLM query or trigger semantic navigation.  
It reads from and writes to sample\_chat.json, maintaining multi-turn history.  
It also integrates with the external RestCountries API and sends individual country data into the Semantic Navigator for reasoning.”

**🧠 2. compressor.py — Dynamic Context Compression Engine**

“This file handles **dynamic context compression**.  
It breaks the full chat history into token-sized chunks, ranks them based on how relevant they are to the current query using TF-IDF, and summarizes the lower-priority ones using the LLM.  
This ensures we always stay within the model’s token limit — without losing key context.”

**🔌 3. llama\_api.py — LLM Request Handler**

“This is the **API interface** to our local LLaMA model served via Ollama.  
It sends prompts, receives responses, and handles generation parameters like max tokens and temperature.  
It keeps the entire solution model-agnostic and deployable locally — which is critical for privacy-focused or regulated clients.”

**🧾 4. sample\_chat.json — Memory Log**

“This is the **chat memory file**. Every query and response is stored here.  
It allows the system to simulate memory across turns and ensures that context compression always operates on the full historical thread.  
It's what makes our LLM ‘aware’ of past interactions in a long conversation.”

**🧭 5. semantic\_navigator.py — Semantic JSON Explorer**

“This is the **core semantic navigation module**.  
It takes any nested JSON and, at each level, prompts the LLM to pick the most semantically relevant key — not through string matching, but by reasoning about meaning.  
It recursively follows that path, building a trace as it goes, until it reaches a final value.  
This gives the LLM a way to explore unknown data structures — like APIs or documents — step by step.”