With state, LLMs are increasingly becoming databases - So we need to incorporate catalogs to engineer accountability

Arunit Baidya

While LLMs and databases serve fundamentally different roles, recent practices and advancements in LLM systems closely resemble those present in database systems. Modern LLMs have diverged from stateless token predictors to stateful LLM systems that retain and manage state through mechanisms like durable logs, retrieval-based augmentation, and user session histories. Once you add state, LLM systems start to store, retrieve, update, index and manage information across multiple data stores, not just generate text.

In production systems, this goes well beyond simple store and fetch operations. SOTA LLM architectures tend to use indexing which combines vector embeddings, metadata based filtering, logs of past queries, and keyword similarity to generate the most relevant results for queries. Further, LLM systems cache extracted data, and use freshness policies to optimize result accuracy against system latency and recomputation expense. Additionally, runtime orchestrators choose chunking policies and retrieval depths to optimize for token, latency, and compute costs. At that point, LLM systems can no longer be reduced to just static probabilistic transformers, but systems that actively manage state.

Once we notice this, the analogy to database systems is undeniable. LLM systems read and write shared data structures over time, and the surrounding stack decides what the model considers as contextual input for a given request. Since RAG pipelines choose which fragments of the knowledge store even enter the context window, they end up affecting the semantic content of the answers themselves, because they influence which facts, flavors, rules, and constraints the model works with. So, it follows that the questions and standards for databases can broadly be applied to LLMs as well. Thus, we must ask what exactly do our answers depend on, what can see or modify states, and how can we tie stateful dependencies which created an output to question accountability on the wider system?

For mature databases, these answers are built into the system. For example, catalog tables in PostgreSQL track relations, data types, functions, triggers, and permissions. You can ask the database to identify objects that depend on a particular column or show which indexes support a given query. Even the final physical query plan selected for execution is

exposed to the user. LLM systems don't have a close equivalent. When a prompt is processed, the users and even model developers are not aware of what embedding indexes and datasets trained on were relied on to provide output. Of course, LLM developers have some understanding of how configurations and internal state influence a model's output, but if you press them on which specific artifacts and environment variables shaped an individual response, no human can track that to a level of detail that is actually useful.

Hence, the need for a stack-global catalog. I believe a close LLM equivalent of database catalogs should identify relations, interactions, and dependencies between the major pieces of the LLM stack as first-class components and objects. In absence of a concrete catalog design, I can narrow the role of a catalog to two critical expectations: being able to identify dependencies between instances of components and artifacts, and versioning components and artifacts over time to identify their active period. A "graph" representation of dependencies between components and artifacts would make the impact of dependencies on each other apparent. Knowing which versions were active at a time make assessing relevant "snapshots" of the system accessible. Essentially, rather than a new component, a catalog is the map of the system's state and wiring itself.

With a catalog implementing the above mentioned expectations, the accountability benefits in principle are clear. When a problematic response is flagged, developers can see the snapshot of the system catalog at that time and see which artifacts from context were relied on, which embedding indexes were used, and what system policies/states active at that time shaped interactions. With a graph representation, identifying any potential bad actor allows us to localize the possible investigation scope and achieve a more granular analysis. Further, over time, the catalog would let us detect recurring failure patterns tied to specific datasets, indexes, prompts, or model versions, so that model limitations, configuration errors, and harmful or misleading stateful data can be addressed proactively rather than reactively.

Having made all these demands of a LLM systems "catalog", and explaining the need and benefit, I do realize that building the catalog is not trivial. Building a catalog firstly requires developers to identify which objects and artifacts should be treated as catalog entries and tracked consistently across the stack. Secondly, overheads in LLM systems, compute costs, and memory costs could balloon depending on the implementation choices. Finally, due to the all encompassing nature of the catalog, it may even become a security concern if it exposes too much about how different users' data flows through the system. However, with the growing reliance of LLMs, compounded by seemingly unbounded increase in size,

complexity, and utility expected of LLMs, I believe that it is necessary to implement catalogs, and get a better sense of accountability with LLM systems, for the sanity of users and developers alike.