## What I want :

Let’s say we want to use an LLM and want to make it know every single thing we do . The problem here is the context limit of every llm models.

**Solution I thought :**

(Everything here is totally based on my little knowledge and some skills that I have . A person cannot think accurately out of his knowledge . I don’t have the knowledge of what solution the industry is following . So I might do mistake or leave some things behind)

**My approach :** For every solution , I will find problems in it and create a solution for that as well , and hopefully I finally find something near perfection .

**Approach :**

We keep the first and last text blocks , and rest in between goes into rag VECTOR STORE.

## Problems I found :

**1. The "Useless Anchor" Problem**

In many real-world interactions, the **first message** is often low-value (e.g., "Hi," or "Are you there?"). If you strictly anchor the first message, you waste precious context window space on a greeting while the actual mission or constraints—which might appear in the second or third message—are moved to RAG.

- **Result:** The LLM stays polite but forgets the actual rules you set at the start of the session.

**2. Semantic Fragmentation**

Conversations are rarely linear. A user might mention a variable in message 4, a logic constraint in message 10, and then try to use them together in message 20.

- **The Issue:** Because message 4 and 10 are now in RAG, the LLM only "sees" them if the RAG retrieval is triggered by a high-similarity match.
- **The Risk:** If the user’s current query doesn't perfectly mirror the phrasing used in message 4, the LLM won't retrieve it, leading to "I don't know what variable you're talking about" errors.

**3. Poor Embedding Quality for Short Messages**

RAG works best on "chunked" documents with rich context.

- **The Problem:** Chat messages are often short and rely on the messages immediately preceding them to make sense.

- **Example:** If a middle message says "Actually, let's use the other one," and you store that in RAG, the vector embedding for that sentence is almost meaningless without the surrounding messages. It becomes "orphan data" that is hard to retrieve accurately.

**4. Increased Latency and Complexity**

Adding RAG to a standard chat loop changes the architecture from a simple API call to a multi-step pipeline:

1. **Step 1:** Embed the current user query.
2. **Step 2:** Query the Vector Database.
3. **Step 3:** Re-rank the results (optional but recommended).
4. **Step 4:** Construct the final prompt with Anchors + RAG results.
5. **Step 5:** Finally, call the LLM.

- **The Cons:** This can add anywhere from **200ms to 1s of latency** per turn and increases the chance of a failure point in your application.

**5. Transition "Jank"**

When a message moves from the "Live" context (the sliding window at the end) into the "Archive" (RAG), the LLM's behavior might suddenly change.

- **The Experience:** One moment the LLM is following a specific nuance perfectly; the next moment, that nuance has "rotated" out of the active window, and the RAG fails to pull it back in with 100% fidelity. This creates an inconsistent user experience.

## SOLUTIONS I THOUGHT:

### To solve 1:

Let’s use a second, cheap, smaller LLM that will define if a message contains any data we need to store or not. With this, we avoid saving "hi" or any other useless text.

### To solve 2:

For every message saved in RAG—for every block of user and AI responses—we will keep proper labeling, such as Message 1, 2, 3, and so on.

### To solve 3:

Let’s keep summarized data within the text block. For example, if after a lot of text the user says "change the color to pink," instead of saving this as a single data point, we keep a list-wise summary of all previous texts with it. This way, any block of text has enough information and context.

## To solve 4:

Yeah, we can't do much about the speed; it will be a bit slow. However, maybe we can fine-tune an LLM just for this task so it performs faster? We can do parallel work, that will reduce the timing .

## To solve 5:

I think this is solved by Solve 3, as every block of text now includes the necessary context.

## PROBLEMS FOUND ON THE SOLVE :

**AS FOR SOLVE 3 ,** This increases **Storage Cost** and **Embedding Latency**, as processing more text for every single message saved.

**Solution :**
We will avoid generating a summary of a summary and will not store the summary in every row. Instead, we will generate and retain it once. When a query is processed and returns data, the summary is treated as a constant, though it updates with each iteration.
—

## Another Problem :

One problem is that , when I hardcode the context , over time , it can become large.

## Solution :

Consider the following hypothetical scenario: We are developing a website with the assistance of an LLM. In the front end, suppose we require a button to be colored blue. We instruct the LLM to implement this, and it does so. This request is consequently added to the hardcoded context summary list. However, since this requirement is resolved in the code, it is unlikely to be needed again.

Therefore, queries of this nature should not be retained in the summary list. Consequently, for every query submitted by the user, it is designated as a task for the LLM and remains in the summary list even after resolution. We track the iteration count from the moment the query was initially posed; if it is not subsequently referenced, it is removed from the summary and forwarded to the RAG system.

Another issue here is that, let's say there was a text where one said his name or business name , now with our previous logic , it will forget this too.

So , we use :
Tiered Summary Architecture

Instead of one big block, split "hardcoded" context into three distinct layers with different expiration rules:

The North Star (Static): The core project mission and immutable rules (e.g., "Always use Python 3.11," "Target: 180marketing clients"). This never changes or grows.

The Active Pulse (Dynamic): A high-level bullet list of current unresolved tasks or open variables.

The static part always stays there . The dynamic part stays till the time it is resolved .

—-----
IMPLEMENTATION ONGOING..

**Nazmul Hasan**
**AI Automation Engineer**
**7 Feb 2026**