Multi Agent Assistant With 3 Types of Memory

Sherman Kettner[1]

University of Colorado Colorado Springs
1420 Austin Bluffs Pkwy, Colorado Springs, CO 80918 USA
skettner@uccs.edu

Abstract

This project investigates the integration of artificial intelligence (AI) agents with augmented reality (AR) smart glasses to assess their viability as intuitive, hands-free assistants for everyday tasks. To explore this potential, two prototypes were developed: a console application for testing local AI models and memory architecture, and a mobile application simulating deployment on AR smart glasses with cloud-based AI integration. The system features a modular, multi-agent design supported by a three-tiered memory model—instant, short-term, and long-term—intended to enable seamless context sharing between agents. Key challenges included inconsistent behavior from large language models (LLMs), difficulty in creating reusable short-term summaries, and hardware constraints when running local models in parallel. Results demonstrate that while cloud-based AI and instant/long-term memory function effectively, short-term memory remains a technical challenge. The project concludes that multi-agent systems on wearable platforms are feasible with appropriate memory models, error handling, and minimal tactile interfaces, laying the groundwork for future development of proactive AI assistants in AR environments.

Introduction

Augmented reality (AR) smart glasses and artificial intelligence (AI) agents are two rapidly advancing technologies with the potential to fundamentally reshape human–computer interaction. This project explores their integration, focusing on whether AI agents can effectively assist users with everyday tasks while requiring minimal user input. Minimizing interaction is essential—if the assistant demands frequent or complex input, smartphones remain the more practical alternative. Therefore, this research emphasizes the need for intuitive and lightweight AI interactions specifically tailored to wearable AR environments.

While current large language models (LLMs) offer impressive capabilities, they often operate as sophisticated search engines rather than proactive assistants. A truly effective assistant must go beyond question-answering—it must anticipate user needs, execute diverse tasks, and coordinate across multiple functions. To enable this, the project investigates a modular, multi-agent architecture in which a general-purpose AI agent delegates responsibilities to more specialized agents.

However, expanding functionality through modularity introduces new challenges, especially in memory management. Agents with limited or inconsistent access to shared memory struggle to operate cohesively. To address this, the project proposes and evaluates a three-tier memory system designed to support interoperability and context-sharing across agents.

Methodology

To explore the feasibility of AI-assisted AR smart glasses, this project was structured around the development of two prototype systems, each serving a distinct role. The first was a lightweight console application designed for rapid experimentation with memory architecture and multi-agent coordination using a local AI model. This environment served as a testbed for implementing shared memory models and enabling AI-to-AI communication. The second prototype was a mobile Android application that simulated deployment on AR smart glasses. This application focused on user interface (UI) design and incorporated lessons learned from the console version—particularly with respect to memory handling and agent coordination—while integrating a cloud-based AI.

System Design Principles

Several foundational principles guided both implementations:

- **Asynchronous Execution:** Components were designed to support non-blocking operations, allowing agents to read from and write to memory concurrently. In cases where perfect synchronization is not required, eventual consistency is tolerated; otherwise, locks or wait mechanisms are used.
- **Parallel Agent Processing:** Agents operate independently and in parallel, enabling scalability and responsiveness.
- **Agent Communication:** A primary goal was enabling structured delegation between agents. The system parses LLM output as function calls, using consistent interfaces or translation layers where needed to mediate between agents.

- **Text-Based Communication Format:** Because the architecture centers around language models, all inter-agent communication and memory interactions are string-based.

Memory Architecture

Inspired by human cognitive processes, the system organizes memory into three tiers:

- **Instant Memory:** Captures immediate conversational context and operates as a FIFO (First-In-First-Out) buffer.
- **Short-Term Memory:** Stores recent summaries and interaction highlights to maintain continuity across tasks. It uses a modified Least Commonly Used (LCU) or Least Recently Used (LRU) strategy.
- **Permanent Memory:** Provides persistent, long-term storage of all information.

These components are illustrated in Figure 1.

Although the original plan was to give all agents universal access to every memory type and enable unrestricted inter-agent calls, time constraints necessitated a more limited implementation. The resulting architecture is shown in Figure 2.

Development Tools and Environment

The project was implemented in C#, selected for its strong cross-platform capabilities and support for both desktop and mobile development. The Android application was developed using the .NET MAUI framework and tested on a virtual Google Pixel 7 device.

AI Integration: Local vs. Cloud Models

Both local and cloud-based LLMs were evaluated to understand their trade-offs in an AR context.

Local AI The "Llama 3.2 3B Instruct Q8" model was deployed locally using the `Jan` framework. Alternative tools, such as ChatRTX, were tested but discarded due to inadequate API support.

Advantages:

- Ensures complete data privacy by avoiding external communication.
- Offers low latency after initialization.
- Functions offline, making it suitable for environments with poor connectivity.

Disadvantages:

- Requires significant local hardware resources (RAM, CPU/GPU).
- Large storage footprint due to model size.
- Slower initialization and response time compared to cloud-based solutions.

Cloud AI The Gemini API was selected for online inference, providing access to advanced language capabilities without the need for local deployment.

Advantages:

- Access to high-performance models maintained by the provider.
- Faster, scalable inference without local resource constraints.
- Easier integration and updates.

Disadvantages:

- Requires stable internet connectivity.
- Involves potential privacy risks due to external data transmission.
- May incur financial or rate-limit constraints.

Long-Term Memory Storage

To implement persistent memory, the system uses PostgreSQL as its database backend, chosen for its stability and extensibility. Database management and inspection were performed using PGAdmin 4. This setup enables reliable long-term storage of agent knowledge and interaction histories across sessions.

User Interface Considerations

Based on firsthand experience with AR devices, one of the most significant usability barriers is a cumbersome interface. To address this, the prototype adopts a minimalist tactile input system augmented with voice-to-text for complex tasks. The tactile interface includes just two controls: "Next" and "Select." This design supports efficient menu navigation with minimal cognitive load—similar in concept to navigating a desktop interface using only the "Tab" and "Enter" keys, but optimized for simplicity and physical wearability.

Software Design

Console Application: Local AI and Memory Architecture

The project began with the development of a console-based application to test local AI integration and prototype the memory system. Initial efforts focused on establishing API communication with a locally hosted language model via the `Jan` framework. This process encountered challenges related to request formatting and response encoding. Ultimately, successful interaction was achieved using the following C request structure:

```
1  var aiRequest = new AIRequest
2  {
3      Messages = new List<Message>
4      {
5          new Message { Role = "user",
       Content = userInput }
6      },
7      Model = "llama3.2-3b-instruct",
8      Stream = false,
9      MaxTokens = 256,
10     MaxCompletionTokens = 256,
11     Temperature = 0.7,
```

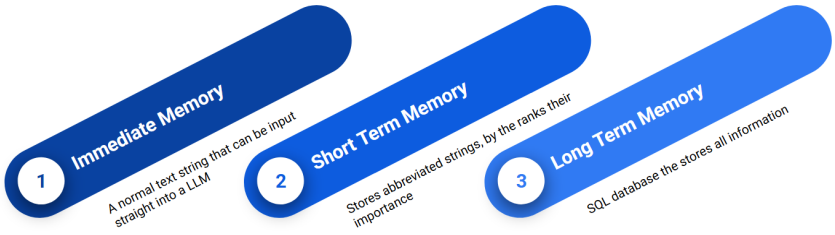

Figure 1: Three-tiered AI memory model

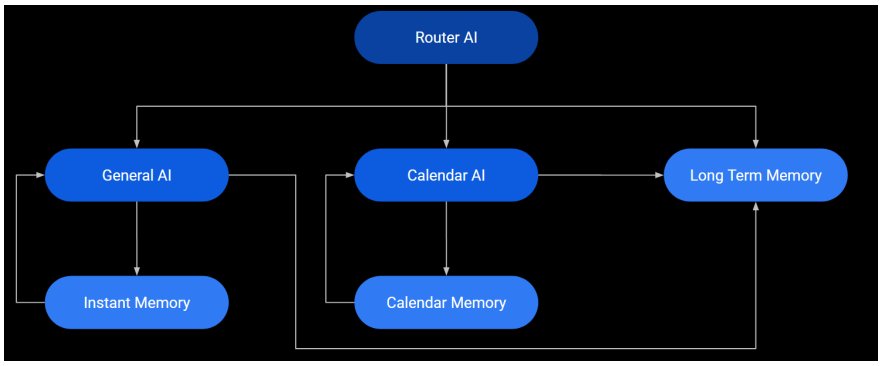

Figure 2: System architecture for agent and memory interaction

```
    TopP = 0.9,
    N = 1
};
```

The payload was serialized and encoded as follows:

```
string payload = JsonConvert.
    SerializeObject(aiRequest,
    serializerSettings);
var content = new StringContent(payload,
     Encoding.UTF8, "application/json");
```

The request was then sent, and the response parsed:

```
HttpResponseMessage response =
    httpClient.PostAsync(apiUrl, content)
    .Result;
string rawResponse = response.Content.
    ReadAsStringAsync().Result;
```

Memory Implementation Once AI interaction was established, memory functionality was integrated using the three-tiered model.

Short-Term Memory: Short-term memory was implemented using a combination of a `Dictionary` to track importance and a `SortedSet` to maintain order based on usage. Each memory item was stored as a tuple consisting of a string and a double representing importance. Core methods included `AddNewEntry`, `RemoveEntry`, `GetFinalString`, and `GetTopStringsDescending`.

Similarity detection was implemented by prompting the LLM to compare new entries against existing memory strings. Prompts used included:

```
Summarize this: {input} down to 200
    characters. Only output this summary.
```

and for similarity detection:

```
Is this string: "{input}"
similar to any of these strings:
{formattedList}

Return each similar string and its
    similarity score out of 100,
separated by '|', and wrap percentages
    in %%.
```

This similarity feedback was parsed and used to decide whether to merge, replace, or retain memory entries. While functional, this approach remains imperfect and is a candidate for future refinement.

Instant Memory: Instant memory was designed as a simple fixed-length FIFO buffer, maintaining the most recent user and AI messages for context-aware responses.

Permanent Memory: Long-term memory was handled via a PostgreSQL database. Entries were stored asynchronously using the following fire-and-forget pattern:

```
public void StoreMemoryFireAndForget(
    string content, string type)
{
    Task.Run(async () =>
    {
        try
        {
            await StoreMemoryAsync(
```

```
    content, type);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error storing memory asynchronously: {ex.Message}");
        }
    });
}
```

This design enables concurrent memory updates across multiple agents without blocking operations, facilitating scalable context retention.

Android Application: Online AI and Interface

The Android application was developed using the .NET MAUI framework and tested on a simulated Google Pixel 7. Unlike the console version, this prototype connected to an online language model through the Gemini API. Integration was straightforward using the provider's standard documentation.

All core memory modules from the console version were reused. The primary enhancement was the addition of a XAML-based user interface, featuring four navigable options:

- **Home:** Allows the user to choose between the Router AI, General AI, Long-Term Memory, and Calendar AI.
- **General AI:** Enables interaction with a stateless LLM. To simulate memory, the app injects short-term memory content into every API call.
- **Calendar AI:** Supports adding and searching calendar events.
- **Long-Term Memory:** Enables querying of stored memory entries in the PostgreSQL database.

Navigation was controlled via a minimalist two-button interface: `Next` and `Select`. This approach was chosen to reflect the limited tactile input space available on AR glasses while maintaining ease of use. Navigation logic was implemented as follows:

```
        private void OnNextClicked(object sender, EventArgs e)
        {
            if (Options.Count == 0)
                return;

            currentIndex = (currentIndex + 1) % Options.Count;
            var item = Options[currentIndex];

            OptionsView.SelectedItem = item;
            // scroll it into viewcentered
            OptionsView.ScrollTo(item, position: ScrollToPosition.Center, animate: true);
        }

        private async void OnSelectClicked(object sender, EventArgs e)
        {
            var selected = OptionsView.SelectedItem?.ToString();
            if (currentPage.SelectAction != null)
            {
                await currentPage.SelectAction(selected);
                // and if you want to reset view when you re-populate:
                if (Options.Count > 0)
                    OptionsView.ScrollTo(Options[0], position: ScrollToPosition.Start, animate: false);
            }
        }
```

Additionally, the Router AI module attempts to classify user input and redirect the request to the most appropriate subagent, streamlining the overall interaction process.

Memory Table Design

The database schema for persistent memory was defined as follows:

```
CREATE TABLE ai_memory (
    id SERIAL PRIMARY KEY,
    type TEXT NOT NULL,
    file_data BYTEA NOT NULL,
    created_at TIMESTAMPTZ DEFAULT NOW()
);
```

For calendar events, the following schema was implemented:

```
CREATE TABLE calendar_events (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    description TEXT,
    event_time TIMESTAMP NOT NULL
);
```

These tables support long-term storage of AI knowledge and user-defined tasks, enabling stateful behavior across sessions.

Results

The console application successfully validated the core functionality of all three memory architectures: instant, short-term, and long-term. Both instant and long-term memory performed reliably, supporting consistent data storage and retrieval across multiple sessions. In contrast, short-term memory posed challenges—particularly in terms of relevance ranking and task continuity—due to limitations in AI behavior and prompt interpretation. These challenges are discussed further in the following section.

The console prototype also confirmed the feasibility of multi-agent coordination and local AI integration. However, performance testing revealed a critical bottleneck: simultaneous execution of multiple AI agents on consumer-grade hardware was not sustainable, even on high-end machines.

This limitation underscored the need to shift AI workloads to cloud infrastructure for scalable deployment.

The Android application achieved its primary design goals. The simplified two-button navigation interface proved intuitive and responsive, validating the proposed UI model for AR smart glasses. Additionally, the integration of cloud-based AI eliminated the hardware constraints encountered in the console version, enabling smoother operation and improved scalability.

All memory modules from the console application were successfully ported to the mobile platform with only minor modifications. Instant and long-term memory continued to operate as expected, while short-term memory retained its behavioral limitations, reaffirming the need for improved memory summarization and relevance filtering.

Representative screenshots from the final Android prototype are shown in Figure 3, Figure 4, Figure 5, and Figure 6.

Discussion

A central goal of this project was to evaluate the feasibility of different memory architectures in AI-assisted systems. This section reflects on the lessons learned from designing, implementing, and testing the three proposed memory types: instant, short-term, and long-term.

Instant Memory

Instant memory functioned as expected and proved valuable in specific contexts. However, its utility was somewhat limited. For example, when interacting with a local AI model, the model itself retained conversational history internally, rendering an external instant memory layer redundant. As a result, instant memory is most useful in scenarios involving stateless models—such as cloud-based APIs—or when context must be transferred between different AI clients. In such cases, it provides a lightweight mechanism for preserving short-term interaction history.

Short-Term Memory

Short-term memory was both the most ambitious and the most challenging component of the project. Despite significant development effort, a fully functional and reliable implementation was not achieved.

Several obstacles emerged. First, large language models (LLMs) exhibit unpredictability in their output formatting. This made parsing similarity comparisons generated by the AI unreliable, as even minor deviations could cause downstream failures. Attempts to use the LLM for identifying string similarity frequently produced inconsistent or unusable results.

Second, summarization for memory reuse proved more difficult than anticipated. Current LLMs are not designed to generate and interpret summaries in a structured way. For example, when the user introduced themselves as ”Sherman,” the AI summarized this as simply ”Sherman.” Later, when referencing the summary, the AI incorrectly interpreted ”Sherman” as its own name. This highlights a broader limitation: current LLMs lack persistent identity and contextual grounding, making it difficult to create summaries that are meaningful across sessions.

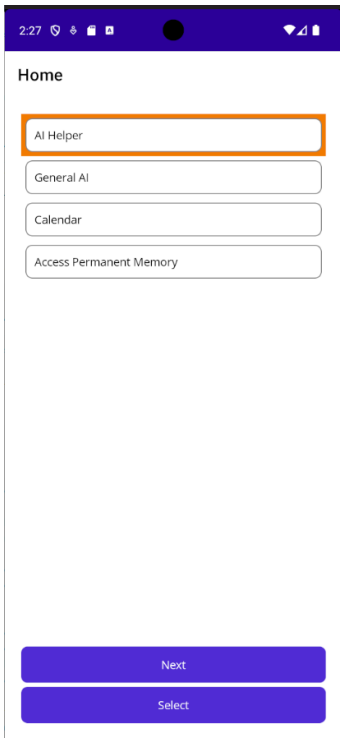

Figure 3: Home screen interface showing available AI modules

Figure 4: Calendar AI interface for event creation and search

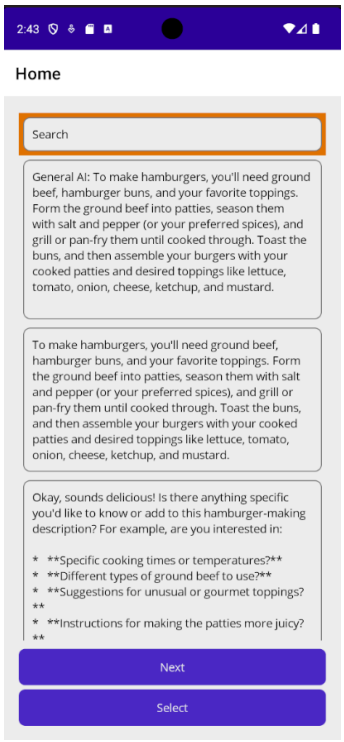

Figure 5: Long-term memory search functionality

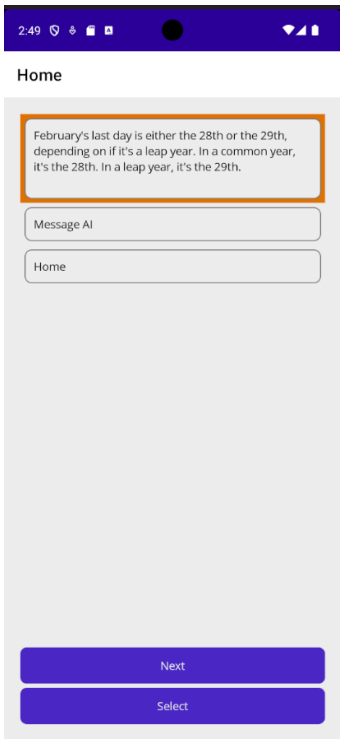

Figure 6: General AI chat interface with short-term memory injection

These issues could potentially be addressed by training models specifically for short-term memory use—capable of both generating structured summaries and reasoning over them in future interactions. In the meantime, alternative approaches—such as non-AI similarity metrics (e.g., Levenshtein distance) or improved error handling and response validation—could provide more reliable results.

Long-Term Memory

Long-term memory performed reliably, as expected from a SQL-based implementation. The PostgreSQL database accurately stored content with associated timestamps and metadata, providing a solid foundation for persistent knowledge storage. However, a key limitation remains: no AI agents currently retrieve or use this stored information. As such, while the system effectively logs past interactions, it does not yet enable historical recall or reflection.

Addressing this gap would be a significant step forward. Integrating long-term memory into agent behavior—so that relevant knowledge can inform responses—would enhance personalization, continuity, and usefulness over time.

General Takeaways

Several broader insights emerged during this project. One key takeaway is the importance of robust error handling when integrating AI-generated outputs into traditional codebases. When AI responses influence memory updates or program control flow, even minor formatting deviations can result in system failures. Therefore, incorporating validation, fallback logic, and clear assumptions is critical.

Another important lesson involves concurrency. While parallelism is conceptually straightforward, practical implementation—particularly in a way that yields real performance benefits—is nontrivial. Although limited progress was made toward concurrent agent execution, the final codebase remains largely single-threaded. Based on this experience, a staged approach—starting with a stable single-threaded implementation before refactoring for parallelism—is likely to be more effective. This allows for better architectural clarity and more controlled debugging before introducing the complexity of asynchronous interactions.

Conclusion

This project explored the integration of AI agents with augmented reality (AR) smart glasses, focusing on memory architecture, agent coordination, and user interface design. Two prototype systems—a console application using a local AI model and an Android application using a cloud-based model—were developed to test different design strategies and memory implementations.

Three types of memory were proposed and evaluated: instant, short-term, and long-term. Instant and long-term memory systems functioned reliably across platforms, while short-term memory presented challenges due to the limitations of current LLMs in generating and reusing structured summaries. These findings suggest that effective short-term memory integration will require either enhanced prompting strategies or future models designed specifically for memory-aware interaction.

In addition, the project demonstrated the feasibility of using a minimal two-button tactile interface for navigating AI interactions, a promising design direction for wearable AR devices. However, limitations in hardware capacity and the unpredictability of AI outputs highlighted the need for robust error handling and the advantages of offloading computation to cloud infrastructure.

Overall, this project provides a foundational architecture for multi-agent AI systems operating on AR platforms and identifies key areas for future development—including memory refinement, agent interoperability, and long-term knowledge utilization.