Breaking the memory barrier

A production blueprint for stateful multi-agent systems on Google Cloud and MongoDB

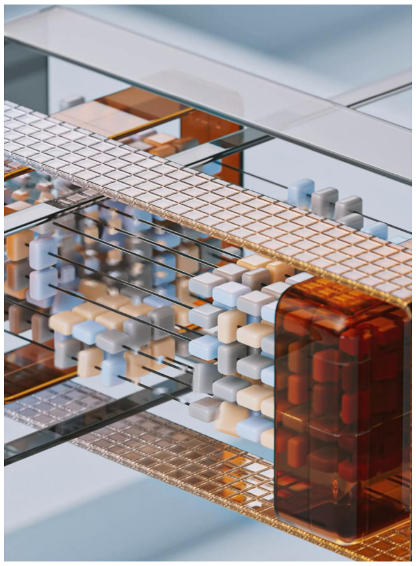

Table of contents

03 Introduction

07 The blueprint in action: Building a smart support agent team

10 Architecting the multi-tier memory with MongoDB Atlas

15 Implementation guide: Running the agent team

18 Advanced patterns for production

21 Scaling the system on Google Cloud and MongoDB Atlas

23 Conclusion

Authors

Pavel Duchovny
Lead Developer Advocate AI Mindshare Team at MongoDB

Christian Williams
Principal Architect AI/ML, Google Cloud Partner Engineering

Introduction: Beyond the amnesiac agent

The current wave of artificial intelligence is rapidly evolving beyond single-purpose, stateless “demo agents” into orchestrated, **stateful “agentic workflows”** that deliver tangible business value. You have likely built or interacted with an AI agent that performs a task impressively, only to forget the entire context of your conversation the next day. Or perhaps you have deployed a team of specialized agents that operate in silos, unable to share critical knowledge about a user or a task. This experience is common, and it highlights the primary obstacle preventing AI from reaching its full potential in the enterprise: the **memory barrier.**

The gap between a promising prototype and a robust production system often comes down to memory. This is not merely about logging conversation history. It is about creating a persistent, searchable, and shareable context that persists across sessions, scales across multiple agents, and enables the system to become more intelligent over time.

Most agent frameworks treat memory as an afterthought, providing basic chat history logs that are insufficient for real-world applications. Production systems demand agents that remember user preferences, learn from past interactions, and share context seamlessly across a team of specialized services to perform complex, multi-step tasks. Failing to address this challenge leads to fragmented user experiences, repeated work, and a fundamental limit on what agentic systems can achieve.

This article presents a **production-ready blueprint** that breaks the memory barrier. It demonstrates how to build sophisticated, stateful multi-agent systems by combining the power of Google Cloud’s open and scalable agent ecosystem with **MongoDB Atlas** as a unified data and memory platform. We will architect and build a multi-agent customer support system to illustrate these patterns, providing a practical guide for developers and architects tasked with creating the next generation of intelligent applications. This approach moves beyond simply “adding memory” and instead advocates for adopting a holistic, production-grade agentic architecture—one that solves the core challenges of orchestration, communication, and persistent, intelligent memory.

The modern agentic stack: A layered architecture

Building robust agentic systems, much like developing any complex software, requires a deliberate, layered architecture. This approach promotes modularity, scalability, and maintainability by separating distinct concerns. Our blueprint is founded on a three-layer stack that combines open standards for communication, powerful frameworks for agent development, and a unified platform for data and memory.

Layer 1: The communication fabric (A2A & MCP)

At the base of any multi-agent system is the protocol that governs how its components interact. Relying on open standards is crucial for interoperability and future-proofing.

A2A (Agent-to-Agent) Protocol: Developed by Google and now managed by the Linux Foundation, A2A is an open protocol designed for communication between agents, even across organizational or technological boundaries. It enables a team of specialized agents to collaborate on complex tasks without being tightly coupled or needing to expose their internal workings, such as proprietary logic or tools. A2A treats every interaction as a trackable task and allows agents to discover each other's capabilities dynamically through a standardized "Agent Card".

MCP (Model Context Protocol): MCP is a complementary open standard for agent-to-tool communication, adopted by industry leaders like Google. While A2A facilitates collaboration between agents, MCP standardizes how an agent securely accesses external tools, data sources, and services. This allows an agent to do things like query a database or access a user's local files in a secure, standardized way.

Layer 2: The agent development & orchestration engine (Google ADK & LangGraph)

With communication standards in place, the next layer provides the tools to build, orchestrate, and manage the agents themselves.

Google's Agent Development Kit (ADK): ADK is Google's open-source, code-first Python framework for building, testing, and deploying sophisticated agents and multi-agent systems. Designed to power enterprise-grade applications, ADK excels at creating modular systems of specialized agents that can be composed into hierarchies for complex coordination and delegation. It provides an integrated developer experience with a CLI and web UI for local testing, built-in evaluation frameworks, and flexible deployment options to services like Google Cloud Run, Google Kubernetes Engine (GKE), or the fully managed Vertex AI Agent Builder.

LangGraph: While ADK provides the framework for the overall multi-agent application, LangGraph is used to define the internal cognitive architecture of each individual agent. LangGraph, a library from LangChain, allows developers to build stateful, cyclical graphs that represent an agent's thought process. This makes it exceptionally well-suited for implementing patterns like ReAct (Reasoning and Acting), where an agent iteratively thinks, uses tools, and observes outcomes to solve a problem.

Layer 3: The unified memory & data platform (MongoDB Atlas)

The top layer is the foundation for statefulness, providing the system with its memory.

MongoDB Atlas: MongoDB Atlas serves as more than just a database; it is a comprehensive data platform for modern AI applications. Its core value proposition for agentic systems is the ability to manage operational data and vector embeddings within a single, unified platform. This eliminates the "synchronization tax"—the complexity, cost, and latency associated with maintaining separate databases for structured data and vector search. By using Atlas, developers ensure that an agent's memory is always consistent with the underlying business data, enabling real-time, context-aware responses. With native capabilities like Atlas Vector Search, it provides the full spectrum of memory services required by intelligent agents.

This layered stack provides a clear separation of concerns: A2A and MCP handle communication, ADK and LangGraph handle agent logic and orchestration, and MongoDB Atlas handles all facets of memory and data.

The blueprint in action: Building a smart support agent team

To demonstrate this architecture, we will build a customer support system for a fictional smart device company, “Aura Devices.” This system will consist of a team of specialized agents working together to resolve user issues efficiently.

The agent team

Our system is composed of several autonomous agents, each with a specific role:

- **Host agent:** Built with Google’s ADK, this agent serves as the front door to the system.

It runs the user interface, receives initial user queries, and acts as an orchestrator, routing requests to the appropriate specialized agent based on the nature of the inquiry.

- **Support agent:** This is the product expert. It has deep knowledge about all Aura Devices products, from their specifications to troubleshooting procedures. It handles all product-related questions.
- **Scheduling agent:** This agent is a calendar wizard. Its sole function is to book support appointments, manage available time slots, and interact with the scheduling system.
- **MCP service:** This is not an agent but a backend service that exposes scheduling tools (e.g., get_free_slots, schedule_meeting) via the Model Context Protocol (MCP). This demonstrates how A2A agents can securely integrate with and consume external tools and APIs.

Resolving the "opaque agent" paradox

A core principle of the A2A protocol is "opaque execution," meaning agents can collaborate without exposing their internal state, proprietary logic, or specific tool implementations. This enhances security and protects intellectual property. A potential question arises: how can agents share context if they are opaque?

Our architecture resolves this elegantly. The agents do not share their internal memory or thought processes directly. Instead, they interact with a shared, external, and arbitrated memory layer hosted on **MongoDB Atlas**. This is analogous to a human support team using a centralized Customer Relationship Management (CRM) system. Team members do not read each other's minds; they read from and write to the shared CRM, which serves as the single source of truth for customer interactions. This pattern allows for governed, auditable, and secure collaboration while respecting the autonomy and opacity of each agent.

Code deep dive: Agent discovery with A2A

The host agent dynamically discovers and connects to the other agents using their A2A AgentCard. The AgentCard acts as an AI's business card, advertising its name, capabilities, and endpoint URL. This enables a plug-and-play architecture where new agents can be added to the system with minimal configuration changes.

The following Python code shows how the host agent initializes its connections to remote agents by fetching their cards:

```python
async def _async_init_components(self, remote_agent_addresses: List[str]):
    """
    Initializes connections to remote agents by fetching their A2A AgentCards.
    """
    async with httpx.AsyncClient(timeout=30) as client:
        for address in remote_agent_addresses:
            card_resolver = A2ACardResolver(client, address)
            try:
                card = await card_resolver.get_agent_card()
                remote_connection = RemoteAgentConnections(
                    agent_card=card,
                    agent_url=address
                )
                self.remote_agent_connections[card.name] = remote_connection
            except Exception as e:
                print(f"ERROR: Failed to initialize connection for {address}: 
{e}")
```

Each specialized agent, like the support agent, defines and exposes its own card:

```python
agent_card = AgentCard(
    name="Support Agent",
    description="Handles user support queries and product information for Aura 
Devices.",
    url=f"http://{host}:{port}/",
    version="1.0.0",
    defaultInputModes=["text"],
    defaultOutputModes=["text"],
    capabilities=AgentCapabilities(streaming=True),
)
```

This mechanism for self-description and discovery is fundamental to building flexible and scalable multi-agent systems.

Architecting the multi-tier memory with MongoDB Atlas

Production-grade agent memory is not a single, monolithic block. Different types of information require different storage and retrieval strategies. Our blueprint implements a sophisticated, three-tier memory architecture using specific LangChain-MongoDB integrations, all running on the unified **MongoDB Atlas** platform. This multi-tiered approach ensures that the right type of memory is used for the right purpose, optimizing for performance, scalability, and intelligence.

The following table outlines the three tiers of memory, their purpose, and the components used to implement them.

Memory tier	LangChain/ MongoDB component	Purpose & function	Use case example
Tier 1: State persistence	MongoDBSaver (Checkpointer)	Saves the complete, serializable state of a LangGraph conversation thread. Enables fault tolerance and long-running, stateful session resumption.	A user returns after a week; the conversation resumes exactly where it left off, with all context intact.
Tier 2: Semantic memory	MongoDBStore with Atlas Vector Search	Stores information as vector embeddings for semantic search. Allows agents to find contextually relevant memories, not just keyword matches.	A user mentions they “dislike straps that pinch.” The agent later recalls this preference when recommending new watch bands.
Tier 3: Structured data	Standard MongoDB collections	Stores operational business data (e.g., user profiles, product catalogs, appointment slots) for reliable, transactional access via standard queries.	The Scheduling Agent queries a collection to find and book an available support slot, ensuring no double-bookings.

Tier 1 deep dive: Conversation checkpoints (MongoDBSaver)

The foundation of a stateful agent is the ability to persist its execution state. For agents built with LangGraph, the MongoDBSaver provides this capability out of the box. It is a checkpointer that serializes and saves the entire state of an agent's graph—including the message history, intermediate steps, and current node—to a MongoDB collection at the end of each step. This is critical for two reasons:

1. Fault tolerance: If an agent process crashes, it can be restarted and resume from the last saved checkpoint, preventing loss of work.
2. Long-running sessions: It allows conversations to be paused and resumed days or weeks later, as the complete context is durably stored.

Initializing the checkpointer is straightforward:

```python
# In an agent's initialization code
from langchain_mongodb.checkpoint import MongoDBSaver
from pymongo import MongoClient

# Establish connection to MongoDB Atlas
client = MongoClient(os.environ["MONGODB_URI"])

# Instantiate the checkpointer

checkpointer = MongoDBSaver.from_conn_string(
    conn_string=os.environ["MONGODB_URI"],
    db_name="agent_memory",
    collection_name="a2a_thread_checkpoints"
)
```

Tier 2 deep dive: Semantic memory (Atlas Vector Search)

This tier is where the "intelligence" of the memory system resides. While checkpoints save the raw state, semantic memory stores distilled knowledge in a way that agents can reason about. This is achieved by converting important pieces of information into vector embeddings—numerical representations that capture semantic meaning—and storing them in MongoDB.

With **Atlas Vector Search**, agents can then perform similarity searches to retrieve memories based on contextual relevance, not just keyword matches. For example, a query about "sore wrists" can retrieve a past user comment about "straps that pinch," because their vector representations are close in high-dimensional space.

The **VertexAIEmbeddings** class from LangChain's Google integrations provides a simple interface for this, backed by a MongoDB collection with a configured Vector Search index.

Python
```python
# In an agent's initialization code

import os
from pymongo import MongoClient
from langchain_google_vertexai import VertexAIEmbeddings
from langchain_mongodb.vectorstores import MongoDBAtlasVectorSearch

# 1. Establish connection to MongoDB Atlas using the connection string
client = MongoClient(os.environ["MONGODB_URI"])

# 2. Define the collection that will store the vector embeddings
collection = client["agent_memory"]["a2a_semantic_memory"]

# 3. Instantiate the embedding model from Google Vertex AI
embedding_model = VertexAIEmbeddings(
    model_name="gemini-embedding-001",
    project=os.environ["GCLOUD_PROJECT"]
)

# 4. Instantiate the MongoDBAtlasVectorSearch class
vector_store = MongoDBAtlasVectorSearch(
    collection=collection,
    embedding=embedding_model,
    # The 'index_name' must match the name of the Vector Search Index
    # you created beforehand in the MongoDB Atlas UI.
    index_name="default_vector_index"
)

# The 'vector_store' object is now ready to be used.
# For example, it can be passed to an agent or used directly
# to add documents or perform similarity searches.
```

Tier 3 deep dive: Structured business data

Not all data belongs in a vector index. Critical business data—such as user profiles, product catalogs, inventory levels, and appointment schedules—requires the reliability and transactional guarantees of a standard database. MongoDB's flexible document model is ideal for storing this structured and semi-structured data.

In our demo, the MCP Service for scheduling interacts directly with a standard MongoDB collection to manage appointment slots. This ensures that when the Scheduling Agent books a meeting, it does so via a reliable, atomic operation that prevents conflicts like double-booking.

By leveraging these three tiers on a single platform, the architecture provides a comprehensive memory solution that is robust, intelligent, and scalable.

Python
```python
# Example tool within the MCP Service
@mcp.tool
async def schedule_meeting(request: ScheduleMeetingRequest) ->
MeetingSlotResponse:
    """
    Books a meeting slot in the calendar if it is available.
    """
    meetings_collection = db["meetings"]

    # Atomically find and update an available slot to prevent race conditions
    result = meetings_collection.find_one_and_update(
        {"time": request.time, "booked": False},
        {"$set": {"booked": True, "user_id": request.user_id}}
    )

    if result:
        return MeetingSlotResponse(success=True, message="Appointment
scheduled.")
    else:
        return MeetingSlotResponse(success=False, message="Slot is no longer
available.")
```

Implementation guide: Running the agent team

This section provides a step-by-step guide to running the multi-agent system on a local machine. The architecture is designed for the cloud, but this allows for rapid development and testing.

Prerequisites

- Python 3.10+
- A MongoDB Atlas account (the free M0 tier is sufficient for this demo).
- A Google Cloud account with the **Vertex AI API enabled.**
- The uv package manager (recommended) or pip.
- **Google Cloud CLI:** You must authenticate your local environment. Run the following command in your terminal and follow the prompts:

Shell
```
gcloud auth application-default login
```

Step 1: Clone and install dependencies

First, clone the project repository and install the required Python packages.

Shell
```
git clone <repository_url>
cd a2a-agents-mongodb
uv sync  # or pip install -r requirements.txt
```

Step 2: Configure environment variables

Create .env files in each of the agent directories (host_agent/, support_agent/, scheduling_agent/) and populate them with your credentials.

Shell
```
# In host_agent/.env, support_agent/.env, and scheduling_agent/.env
MONGODB_URI="mongodb+srv://<username>:<password>@<cluster-url>/"
GCLOUD_PROJECT="your-google-cloud-project-id"

# In scheduling_agent/.env, add the MCP server URL
MCP_SERVER_URL="http://localhost:8001/mcp"

# In host_agent/.env, add the URLs for the specialized agents
SUPPORT_AGENT_URL="http://localhost:8001"
SCHEDULING_AGENT_URL="http://localhost:8002"
```

Step 3: Start the MCP server

The MCP server exposes the scheduling tools. Navigate to its directory and start the server.

```
Shell
cd mcp_server
uv run main.py
```

This will start a server on port 8000, exposing tools like schedule_meeting.

Step 4: Launch the specialized agents

Each agent runs as an independent process. Open separate terminal windows for each one.

```
Shell
# Terminal 1: Support Agent
cd support_agent
uv run main.py --port 8001

# Terminal 2: Scheduling Agent
cd scheduling_agent
uv run main.py --port 8002
```

Step 5: Launch the host agent and UI

Finally, start the host agent, which also serves the web-based user interface.

```
Shell
# Terminal 3: Host Agent
cd host_agent
uv run app.py --port 8080
```

Navigate to http://localhost:8080 in your browser. You can now interact with the agent team. Try asking questions that require collaboration, such as:

- *"Tell me about the Aura Pro watch."* (Handled by the support agent)
- *"That sounds great. Can you schedule a demo for me tomorrow at 3 pm?"* (Handled by Scheduling Agent)
- "\ (The system will store this fact and recall it later)

As you interact, you can observe the agent logs in each terminal to see the A2A communication and memory operations in action.

Advanced patterns for production

Moving the system from a local demo to a scalable, enterprise-grade deployment requires adopting specific design patterns for managing memory and ensuring robust operation. These patterns address key concerns of multi-tenancy, governance, and long-term data lifecycle.

Pattern 1: Intelligent memory management with LangMem

In a simple implementation, a developer might write explicit logic to save information to memory. However, a more autonomous and scalable approach is to empower the agent to manage its own memory. The LangMem library, designed to work with LangChain and LangGraph, provides tools for this purpose. By equipping an agent with langmem's create_manage_memory_tool, the agent can use its reasoning capabilities to decide what information is important enough to commit to long-term semantic memory and when to do so. This aligns with the broader trend of increasing agent autonomy and reduces the burden on developers to anticipate every piece of information that might be valuable later.

Pattern 2: Memory namespacing and isolation

In a multi-tenant application serving many users, it is critical to keep memories isolated. langmem and MongoDBStore support the concept of namespaces, which act as partitions for memory. By using a hierarchical namespace strategy, memories can be cleanly organized and secured. For example, a namespace could be structured as ("user_data", user_id, "preferences"). This ensures that one user's agent cannot access another user's memories, a fundamental requirement for privacy and security.

Pattern 3: Selective memory sharing and governance

While isolation is important, collaboration often requires sharing. This pattern involves creating different namespaces for different scopes of memory:

Agent-Private Memory: Each agent might have a private namespace, such as ("agent_private", agent_id, "reasoning_traces"), to store internal thoughts or logs that should not be shared.

User-Shared Memory: A team of agents serving a single user could share a namespace like ("user_context", user_id) to access common information, such as the user's product ownership or past support tickets.

This selective sharing enables effective collaboration while maintaining control over data access, a key aspect of AI governance.

Pattern 4: Memory lifecycle management

Not all memories should be stored forever. Over time, some information becomes stale or irrelevant. A robust production system must include a strategy for memory lifecycle management. This can be implemented by adding metadata to each memory entry stored in MongoDB, such as: created_at, last_accessed_at, access_count, and relevance_score.

JavaScript

```
// Run this command in the MongoDB shell (mongosh)
db.a2a_semantic_memory.createIndex(
  // Field to index for TTL: MUST be a date type
  { "created_at": 1 },
  {
    // Documents will expire after this many seconds
    "expireAfterSeconds": 2592000, // 30 days

    // The TTL rule ONLY applies to documents matching this filter
    "partialFilterExpression": {
      "relevance_score": { "$lt": 0.5 }
    },
    "name": "memory_ttl_policy_index"
  }
)```
```

MongoDB's TTL Index is perfect for auto-deleting documents from persistent memory. With the data stored correctly, you can apply the index. The command below tells MongoDB to look for documents where relevance_score is less than 0.5 and delete them 2,592,000 seconds (30 days) after the time stored in their created_at field.

With this metadata, background processes can be run to periodically "clean up" the memory:

- **Archiving:** Move old, infrequently accessed memories to cheaper storage.
- **Summarization:** Consolidate multiple related episodic memories into a single, more abstract semantic memory.
- **Decay:** Lower the relevance of memories that have not been accessed in a long time, making them less likely to be retrieved.

This ensures the memory system remains performant and cost-effective as it scales.

Scaling the system on Google Cloud and MongoDB Atlas

The architecture presented is designed for scale from day one, leveraging the strengths of both Google Cloud and MongoDB Atlas.

Scaling the agents on Google Cloud

The containerized agents can be deployed to Google Cloud's scalable compute platforms:

Google Cloud Run: For a serverless, fully managed environment, deploying each agent as a separate Cloud Run service provides automatic scaling based on request volume, from zero to thousands of instances. This is ideal for applications with variable or unpredictable traffic.

Google Kubernetes Engine (GKE): For applications requiring more control over the infrastructure, agents can be deployed to a GKE cluster. GKE provides powerful orchestration capabilities for managing complex, containerized workloads at scale.

Vertex AI Agent Builder: As the application matures, the agents can be deployed to Vertex AI Agent Engine. This is a fully managed, agent-optimized runtime designed specifically for deploying, managing, and scaling agentic systems built with frameworks like ADK.

Scaling the memory on MongoDB Atlas

The data and memory layer scales seamlessly on MongoDB Atlas, which is architected for high-throughput, distributed workloads:

Sharding: As the volume of structured data (Tier 3) and conversation checkpoints (Tier 1) grows, the MongoDB collections can be sharded. Sharding distributes the data and workload across multiple nodes, enabling horizontal scalability to handle millions of users and conversations.

Search Nodes: Atlas Vector Search workloads can be isolated onto dedicated Search Nodes. This allows you to scale your vector search capabilities (Tier 2) independently from your core operational database workload. You can add more Search Nodes as your semantic search traffic increases, ensuring consistently low-latency performance without impacting other parts of the application.

The combination of scalable compute from Google Cloud and scalable, unified data services from MongoDB Atlas provides a robust, end-to-end solution for building and operating agentic systems at any scale.

Conclusion: The future is stateful and collaborative

The era of amnesiac, standalone AI agents is drawing to a close. The future of AI applications lies in building collaborative, stateful agentic systems that can learn, remember, and engage in complex, long-running tasks. The primary obstacle to realizing this future has been the memory barrier, but as we have demonstrated, this barrier can be broken.

The blueprint detailed in this article provides a clear, production-ready path forward. It is built on a foundation of open standards (A2A, MCP), powerful development frameworks (Google ADK, LangGraph), and a unified data platform (MongoDB Atlas). The core architectural pattern—a team of specialized, opaque agents collaborating through a sophisticated, multi-tiered external memory layer—offers a solution that is modular, scalable, and intelligent. By separating concerns into distinct layers for communication, orchestration, and memory, this architecture provides the flexibility and robustness required for enterprise-grade applications.

This approach overcomes the limitations of simplistic memory models, enabling agents to build a rich, contextual understanding of users and tasks over time. It bridges the gap between promising demos and powerful production systems, allowing developers to build AI that delivers genuinely helpful, personalized, and persistent experiences. The age of goldfish-memory agents is over. Welcome to the era of AI with a memory that learns, adapts, and endures.