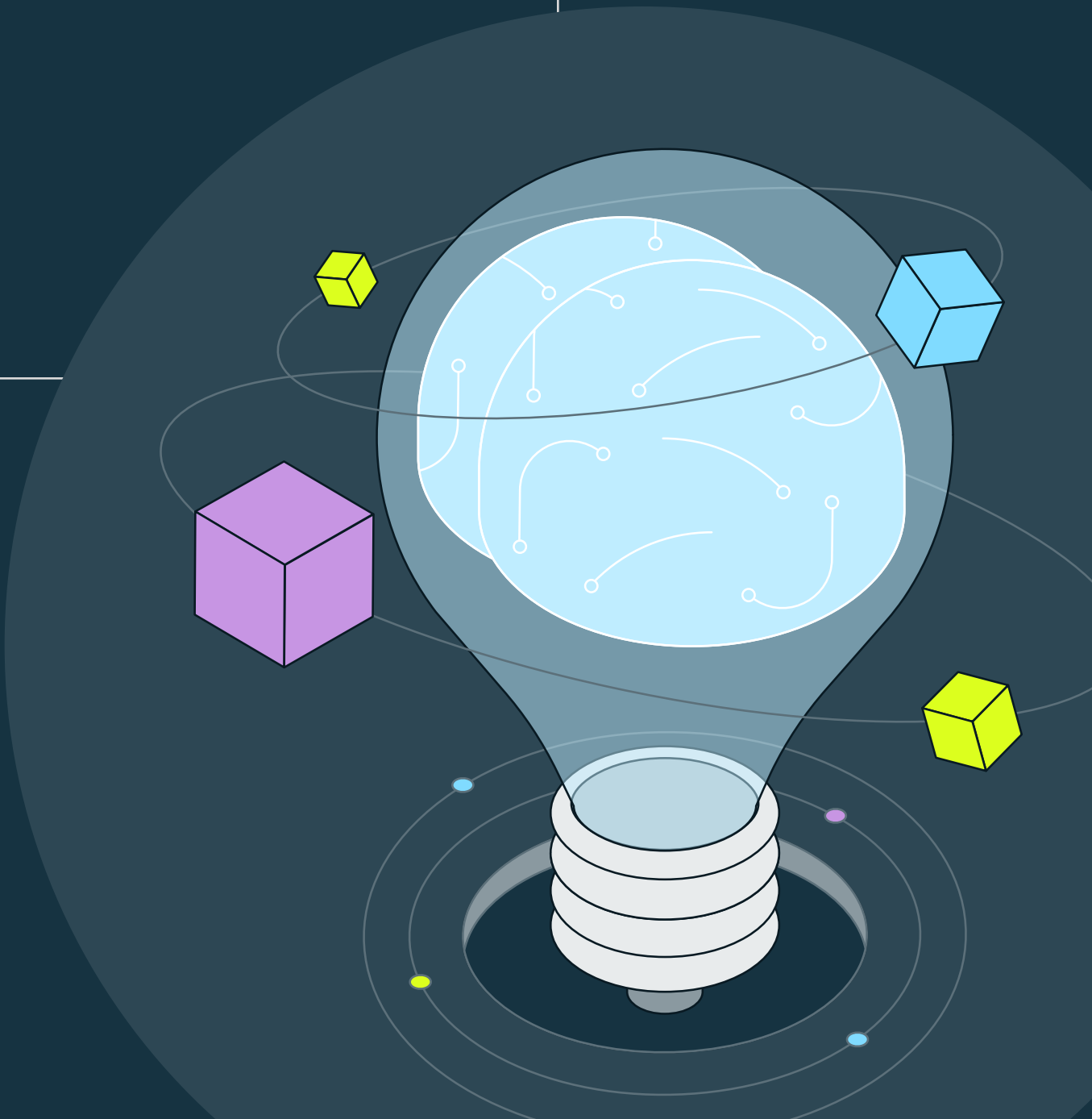
Redis

WHITEPAPER

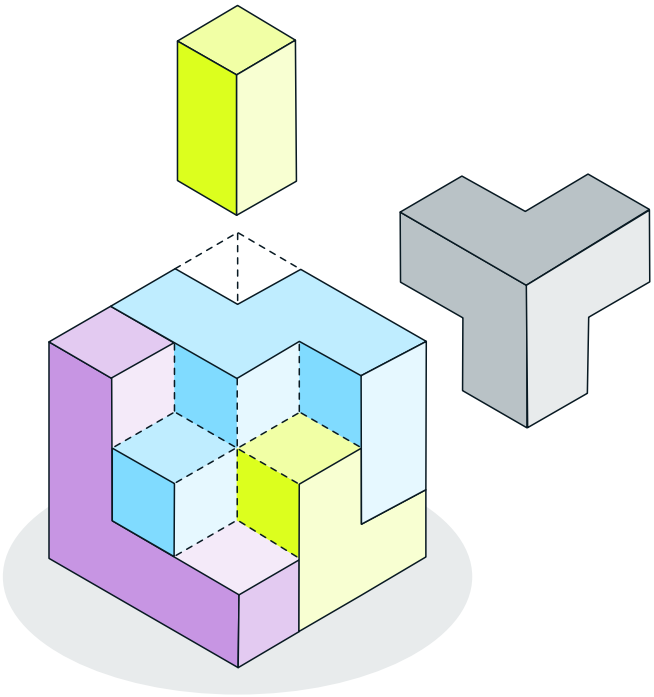
A developer's guide to agent memory



The role of AI agents is changing. Once limited to narrow tasks or reactive chat interfaces, they're now expected to operate with autonomy, manage complex workflows, and collaborate across tools or systems. Developers are now building agents that book travel, troubleshoot issues, conduct research, and coordinate tasks from start to finish—all without constant human input.

But while the vision for intelligent, self-directed agents is compelling, the reality still falls short. Most agents struggle with basic continuity. They forget what happened earlier in a conversation, lose track of progress mid-task, and fail to account for user-specific preferences or goals. These limitations reduce trust, cause inefficiencies, raise inference costs, and often make the experience feel more like restarting than progressing.

The missing ingredient is memory.



What's agent memory?

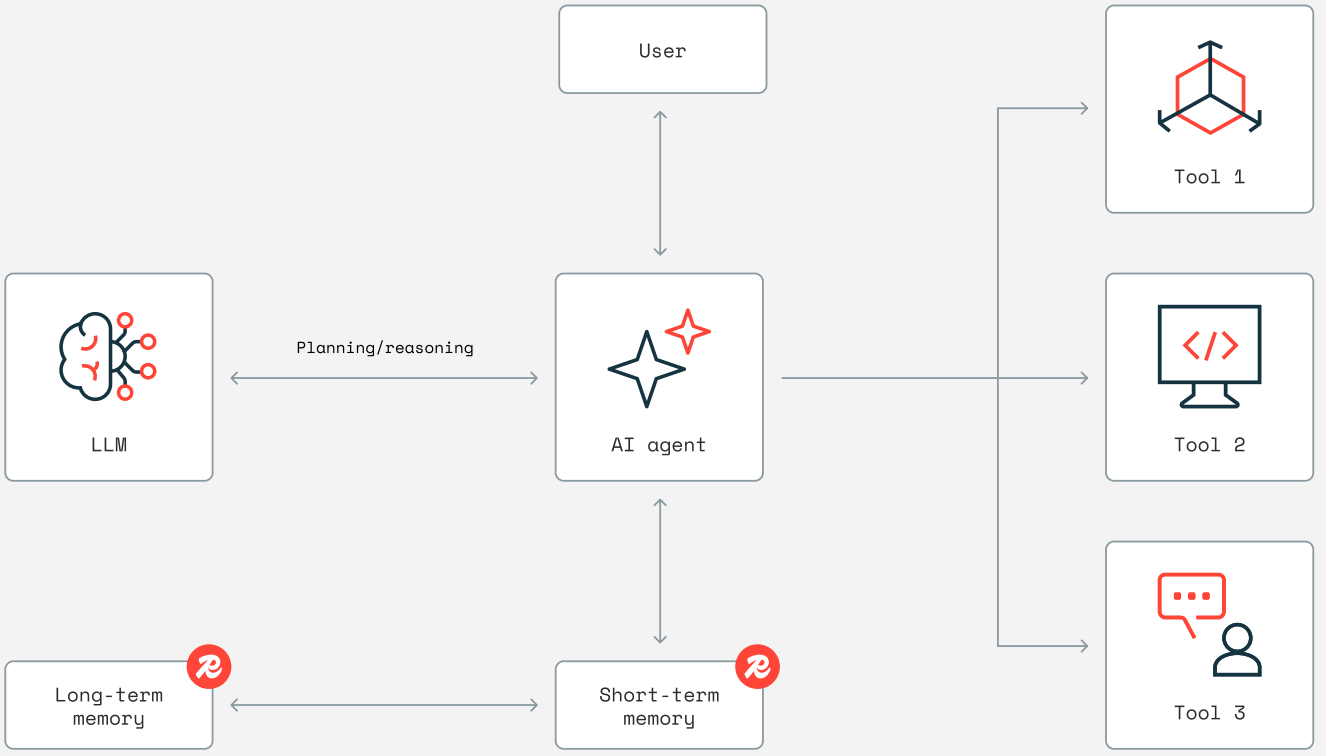
Imagine opening a travel assistant to book a flight for an upcoming trip. You used the same agent last month, so you expect it to remember that you prefer American Airlines, sit in an aisle seat, and usually fly out of JFK. But instead, it treats you like a first-time user, asking for your preferences all over again. You spend extra time re-entering information it should have remembered, and the interaction feels more like a form than a personalized assistant.

Agent memory solves this problem. It allows agents to persist knowledge over time, improving reasoning, personalization, and cost efficiency. With memory, agents can maintain task continuity, coordinate actions across multiple agents, and deliver faster, more context-aware responses. In short, agent memory refers to context that persists across interactions and is stored outside the LLM. Without memory, agents are just interfaces. With memory, they become true assistants.

The building blocks of agent memory

To act intelligently, an AI agent needs more than just a large language model. It needs memory systems that help it make sense of the present, understand the past, and retain important facts that guide future behavior. These systems equip the agent to carry context through a conversation, recall relevant history across sessions, and personalize interactions based on known information. Without memory, agents operate in isolation—each prompt is treated as new, with no awareness of what came before or what the user prefers.

That's why modern agents rely on three complementary types of memory: short-term memory to track recent context, long-term memory to recall meaningful history, and structured memory to store persistent facts like preferences, profiles, and task state. Together, these memory types allow agents to operate more like assistants and less like one-off tools.



How agents use memory

Redis makes three core types of memory possible—short-term (recent interactions), long-term (vector recall across sessions), and structured (user preferences and state). Together, they help agents behave more intelligently, personally, and persistently.

Short-term memory allows the agent to maintain continuity throughout a single interaction. It tracks recent user prompts, tool outputs, and conversation history, enabling the agent to respond contextually without reprocessing everything from scratch. This kind of memory is essential for back-and-forth exchanges, such as booking a flight or troubleshooting an issue, where the agent needs to reference what just occurred. Tools like RedisSaver is used to persist short-term memory in a session, while semantic caching tools like Redis LangCache can complement this by reusing previous completions for repeated or similar prompts.

Long-term memory helps the agent recall context across different sessions or tasks. This includes remembering previously booked trips, user preferences inferred over time, or summaries of past conversations. With long-term memory, an agent can connect dots between current requests and historical behavior. A Redis vector database makes this possible by storing embeddings of prior interactions and allowing fast, filtered retrieval based on semantic similarity and metadata. This is what enables an agent to recall, for instance, that the user prefers American Airlines for business travel and typically avoids red-eye flights.

Structured memory is used to persist exact facts and user-specific data, like a preferred airline, loyalty program ID, or the state of an in-progress task. This form of memory is especially useful for personalization and decision-making. With RedisJSON, the agent can store and update structured records using predictable schemas and query them efficiently when constructing responses. Structured memory gives the agent a factual backbone to work from, keeping responses consistent even as the conversation evolves.

Together, these three types of memory make AI agents more useful, consistent, and capable of adapting to real-world workflows.

Key choices for managing long-term memory

A production-ready memory layer starts with a few key decisions:

1. Use the right type of memory for the task

Select a memory type based on both the structure of the data and how the agent needs to access it. Vector memory is flexible and ideal for retrieving unstructured content like prior conversations or tool outputs. Structured memory is better suited for storing explicit facts like user preferences or task state.

But context matters. For example, a frequent flyer number might first be used in short-term memory during a booking flow. This allows the agent to quickly fill out a form or confirm traveler details. Once the transaction is complete, that same information should be written to long-term memory so the agent can reuse it during future sessions without asking again. In many cases, agents will need both. A Redis vector database can store past flight searches or trip summaries, while RedisJSON holds persistent data like seating preferences or loyalty program IDs.

2. Size memory appropriately

Memory should be broken down into discrete units. Instead of saving an entire session, store individual interactions, booking attempts, or LLM responses as separate entries. This makes retrieval more precise and allows the agent to construct responses using only the most relevant context.

3. Tag memory with rich metadata

Use metadata to organize memory entries. Tag vector embeddings and structured records with labels such as user ID, task type (like “booking” or “support”), and timestamps. For example, tagging embeddings of American Airlines bookings made within the last 30 days gives agents the ability to filter results quickly and retrieve only the most relevant history. This tagging can be automated in several ways: some teams generate metadata directly using model outputs, others apply tagging via a memory orchestration layer, and some memory servers apply post-processing rules to add or update tags based on tool calls or behavior. Regardless of the method, consistent tagging is essential for powering semantic and filtered retrieval in production.

4. Set clear update triggers

Memory should be updated at meaningful checkpoints: after a tool call, when a form is submitted, or at the end of a conversation segment. For instance, when a customer selects a new travel preference, that value should be written immediately to structured memory so the agent can incorporate it going forward.

5. Combine retrieval strategies

Effective agents use hybrid search, blending semantic similarity from vector embeddings with metadata filters or keyword matches. This helps them recall relevant memories even when phrased differently than they were originally stored. Some systems also apply reranking to refine retrieval results based on recency, user preferences, or task relevance.

6. Maintain and prune memory

Short-lived memory items like temporary itineraries should expire automatically using TTLs. Long-term memories can be managed with custom logic, such as deleting records that haven’t been accessed in a given time window. This keeps the memory system efficient and prevents stale information from polluting future interactions.

Why choose Redis for AI agent memory

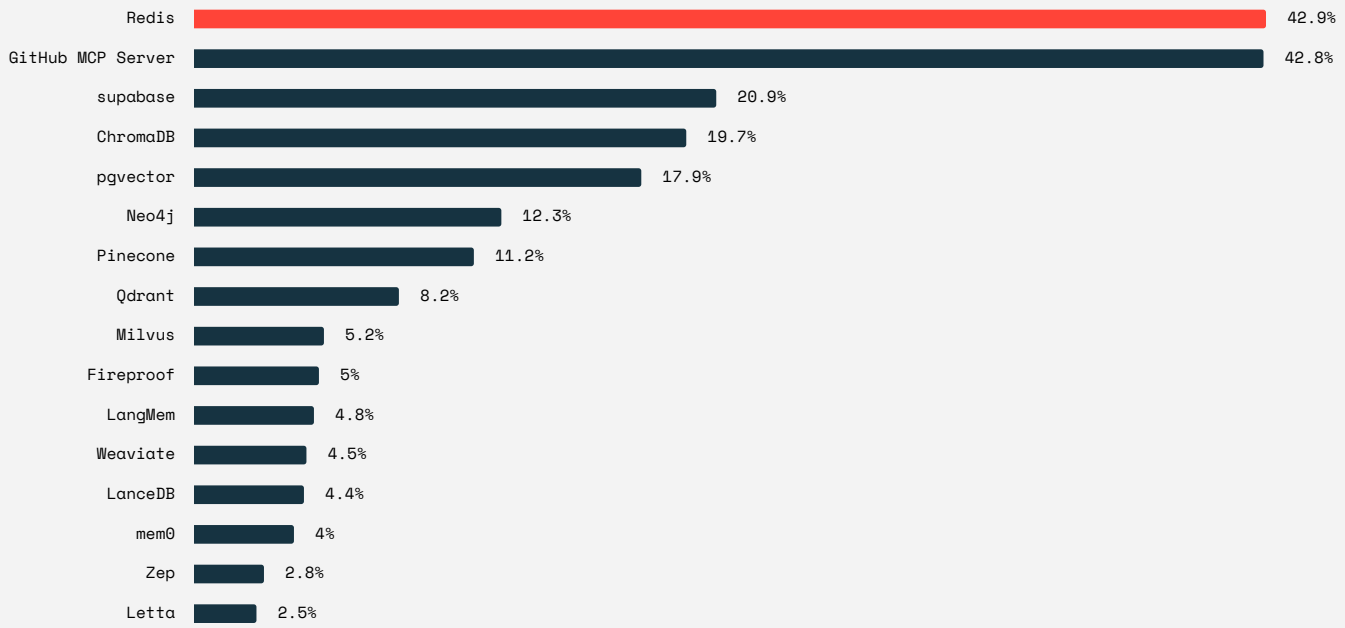
Redis is a unified, low-latency memory layer that supports all three forms of agent memory. With sub-millisecond reads and writes, Redis can keep up with the real-time needs of LLM agents.

Redis also fits seamlessly into popular AI frameworks. It integrates with LangChain, LangGraph, and OpenAI, making it easy to adopt whether you're building an agent from scratch or scaling one into production. Core Redis capabilities include:

- LangCache, a fully-managed semantic caching service for AI apps and agents that makes AI apps faster and more accurate. Instead of calling the model again for similar inputs, the agent can reuse prior completions. For example, if a user recently searched for "flights to San Francisco on Monday with American Airlines," that result can be returned instantly.
- Vector database, a high-performance vector database that supports approximate nearest neighbor (ANN) search with metadata filtering. This makes it possible to recall the most similar past interactions, like all American Airlines flight queries with a preferred time window.
- RedisJSON, a structured document store that supports flexible schema and indexing. This makes it easy to persist user profiles and state, such as frequent flyer details or selected seat preferences.
- Hybrid query capabilities, where Redis combines vector, text, and metadata search to deliver context that is both semantically relevant and filterable.
- Agent memory server patterns, where Redis operates as a shared memory backend for multi-agent apps, making it possible for agents and tools to coordinate on shared user context.

By offering low latency and unified support for all memory types, Redis removes the need for separate systems and simplifies the agent architecture.

AI agent data storage tools



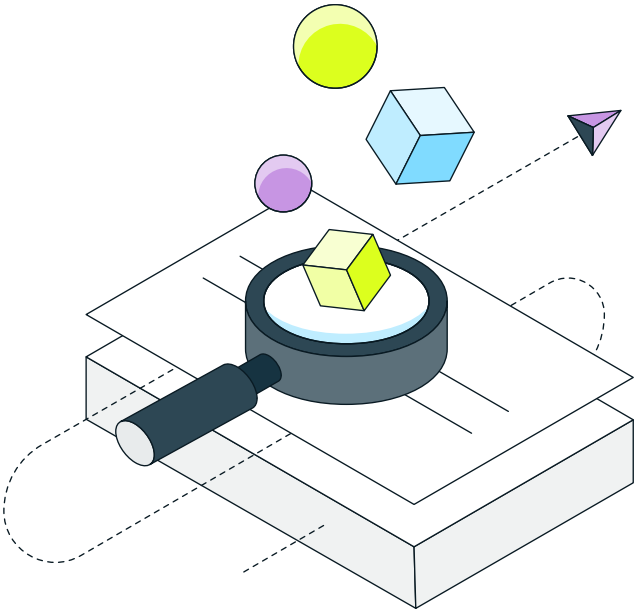
A recent Stack Overflow survey found that Redis is the best tool for AI agent memory and data storage. 43% of developers reported using Redis for agents, ahead of vector-native options like ChromaDB and pgvector.

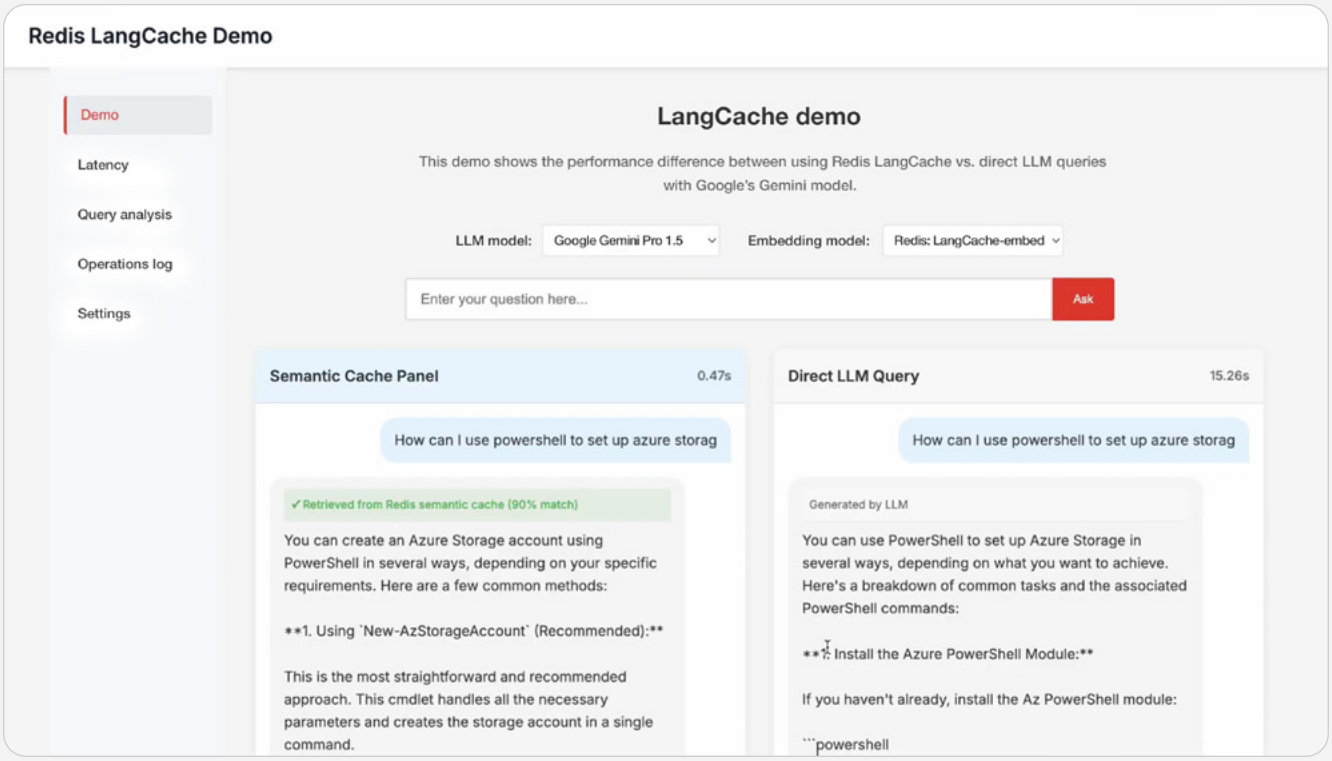
Benefits of agent memory with Redis

With Redis, agents can recall what they've seen and what users have said, making them more responsive and more intelligent.

Agents can respond faster, using cached results from LangCache instead of waiting for a fresh model completion. They become more coherent, using long-term vector memory to reference past interactions and user history. They reduce cost, avoiding redundant API calls and expensive token usage by leveraging cache and memory instead of context stuffing. And they feel more personalized, because they can incorporate structured memory that reflects your preferences over time.

For instance, your AI agent might greet you with, "Welcome back. Ready to book another American Airlines trip to Chicago?" before you've even prompted it.





LangCache speeds up response times by more than 30x

This image shows a side-by-side comparison of LangCache vs. a direct LLM query. The LangCache response is served in just **0.47** seconds from Redis' semantic cache, while the direct LLM call takes **15.26** seconds, highlighting how caching can dramatically cut latency for repeated or semantically similar prompts.

Best practices for production-ready memory

Use Redis effectively in agent memory systems with these key recommendations:

Use TTLs for short-lived data:

Clear out ephemeral memories, like session transcripts, after they're no longer relevant.

Store embeddings with rich metadata:

For example, tag each vector with booking type, date, and destination. This allows hybrid filtering during retrieval.

Cache repeatable queries using LangCache:

Many agents see repetitive queries like "best flights from JFK to LAX." These can be served directly from cache.

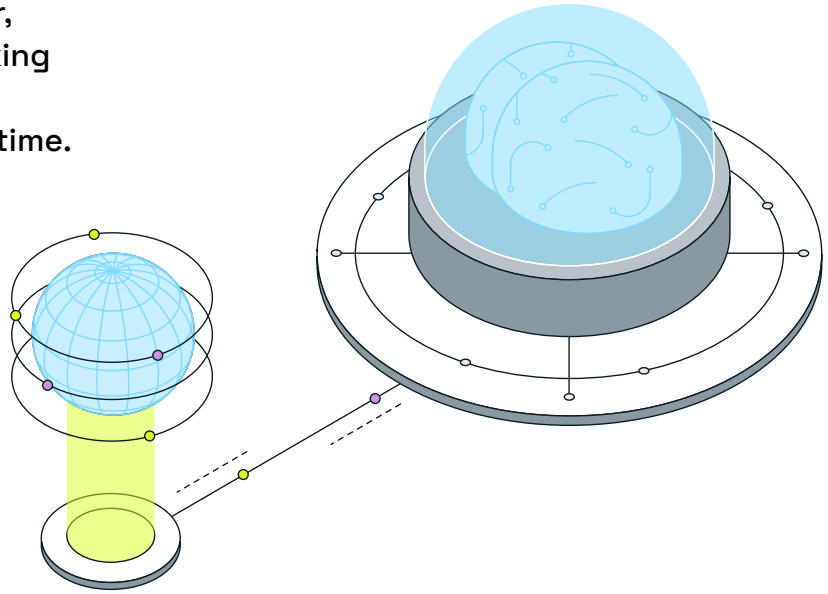
Structure memory writes to avoid overwrite conflicts:

Use user IDs or session tokens to segment memory.

Monitor memory usage:

Use Redis Insight or keyspace statistics to track eviction rates, key volume, and query latency. This helps ensure performance at scale.

Redis combines high performance with a clean, flexible model for managing memory. It supports agents that can remember, adapt, and act—whether they're booking flights, coordinating tasks, or simply delivering smarter experiences every time.



Example: Running agent memory with LangGraph & Redis

Follow along in this Github notebook.

This walkthrough uses LangGraph to build a travel agent that stores and retrieves memories in real time. It covers:

Short-term memory: Conversation checkpointing

A RedisSaver saves conversation history at each node in the graph:

```
from langgraph.checkpoint.redis import
RedisSaver

redis_saver = RedisSaver(redis_
client=redis_client)
redis_saver.setup()
```

Each turn in the conversation is saved automatically. The agent also manages conversation size via a summarizer node:

```
def summarize_conversation(state, config):
    # summarize after MESSAGE_
SUMMARIZATION_THRESHOLD
    # ... call LLM to produce a summary ...
    state["messages"] = [summary_message,
state["messages"][-1]]
    return state
```

Long-term memory: Index, store, and retrieve

1. Define a Redis vector database index schema:

```
from redisvl.index import SearchIndex
from redisvl.schema.schema import
IndexSchema
```

```
memory_schema = IndexSchema.from_dict({
    "index": {"name": "agent_memories",
"prefix": "memory:", "storage_type":
"json"},
    "fields": [
        {"name":"content", "type":"text"},
        {"name":"memory_type",
"type":"tag"},
        {"name":"user_id", "type":"tag"},
        {"name":"embedding",
"type":"vector", "attrs":{

"algorithm":"flat","dims":1536,"distance_
metric":"cosine","datatype":"float32"
        }},
    ],
})
long_term_memory_index =
SearchIndex(schema=memory_schema, redis_
client=redis_client, overwrite=True)
```

2. Store memory entries avoiding duplicates:

```
def store_memory(content, memory_type,
user_id):
    if not similar_memory_exists(content,
memory_type, user_id):
        embedding = openai_embed.
embed(content)
        long_term_memory_index.load([{
            "user_id": user_id,
            "content": content,
            "memory_type": memory_type.
value,
            "embedding": embedding,
            "memory_id": str(ulid.ULID()),
            "created_at": datetime.now().
isoformat(),
        }])
```

3. Retrieve relevant memories with hybrid filters:

```
from redisvl.query import VectorRangeQuery

def retrieve_memories(query, user_id):
    vector_query = VectorRangeQuery(
        vector=openai_embed.embed(query),
        num_results=5,
        vector_field_name="embedding"
    )
    vector_query.set_filter(f"@user_
id:{{{user_id}}}")
    results = long_term_memory_index.
query(vector_query)
    return [doc["content"] for doc in
results]
```

Before each agent response, the graph augments the prompt with relevant memories:

```
def retrieve_relevant_memories(state,
config):
    query = state["messages"][-1].content
    mems = retrieve_memories(query,
config["user_id"])
    if mems:
        memory_context = "\nRelevant
memories:\n" + "\n".join(f"- {m}" for m in
mems)
        state["messages"][-1] =
HumanMessage(content=query + memory_
context)
    return state
```

What each memory layer supports:

Aspect	Impact
Short-term memory	Maintains session context using RedisSaver
Long-term memory	Builds up user history with a Redis vector database
Hybrid retrieval	Combines semantic and filter-based searches
Efficiency	Avoids redundant context and reduces LLM calls
User delight	Agent remembers preferences across sessions (e.g., "American airlines, aisle seat")

These code snippets demonstrate how to build a memory layer that is scalable, efficient, and user-focused, turning a simple travel bot into a context-aware assistant.

Why agent memory matters

Memory is what makes agent behavior intelligent. Without it, even the most advanced language models can't deliver meaningful help. They treat each prompt in isolation, forget user preferences, and fail to build continuity across sessions. As a result, users are left with interactions that feel robotic and impersonal—more like talking to a script than a smart assistant.

Redis is an ideal fit for building agentic memory. With native support for vectors, structured JSON, and hybrid querying—delivered at sub-millisecond speed—it streamlines what would otherwise take multiple disconnected systems.

LangCache speeds up responses by caching completions. The Redis vector database handles fast semantic recall with metadata filtering.

RedisJSON persists structured user data that agents can query and update in real time.

The value of agent memory shows up in production. It helps reduce costs by skipping unnecessary LLM calls, improves responsiveness, cuts repetition, and keeps context alive across touchpoints. Most importantly, it helps agents deliver experiences that feel thoughtful and personalized.

As AI agents take on roles in travel booking, customer support, and enterprise tools, scalable memory becomes essential. It is core infrastructure. Developers who treat memory as a foundational part of the stack, rather than an afterthought, will be best positioned to build agents that actually earn the title "assistant."

Whatever your next step, Redis is ready.

- Redis Agent Memory Server: Both conversational context and long-term memory.
- Redis AI docs: Quickstarts and tutorials to get you up and running fast.
- Redis Cloud: The easiest way to deploy Redis—try it free on AWS, Azure, or GCP.

Let's build the future of AI—together.