Member-only story

Hierarchical Memory and Adaptive State Management for an AI Agent

17 min read · 4 days ago

Jack Luo Follow

Listen Share More

Author's note: to all friends and supporters, please click on this link to read the full story for free: https://thejackluo8.medium.com/hierarchical-memory-and-adaptive-state-management-for-an-ai-agent-f99a50562837?sk=4a1db8a2230524f1269b89fdf892c36a

To build a powerful AI assistant (sometimes called an agent), we need to carefully design how it remembers information, transitions between tasks, retrieves knowledge quickly, and orchestrates multi-step operations. We also want the system to be transparent (showing its plan or reasoning to the user) and customizable (able to adjust components like models or memory settings). Below, we break down each of these aspects in detail.

Memory Partitions: Personal, Project, and Task

An effective assistant should have multiple tiers of memory to handle different scopes of context. We can partition the agent's memory into: Personal memory, Project memory, and Task memory. This ensures the agent uses the right context at the right time without mixing up irrelevant details.

- Personal Memory (Global Context): This is long-term information about the user and the assistant's core configuration. It includes the *system's core directives and the user's general preferences* (for example, preferred tone or known user facts) that persist across all sessions. Personal memory is like the agent's permanent

knowledge base about "you" and itself. It prevents the AI from forgetting key facts be allows the AI to m m of assistants "forgetting everything" once a session ends. Personal memory might also store overarching goals or values and can be seen as analogous to an AI's long-term semantic memory.

- Project Memory (Context for Current Project/Workflow): Project memory is more specific to whatever project, topic, or workflow the user is currently engaged in. It holds information relevant to the *current domain or ongoing project*, such as project-specific documents, terminology, or user preferences for that particular context. This is semi-long-term: it persists throughout the project's life (which might span multiple sessions or tasks) but is not applied to unrelated projects. For example, if the user is working on a "Budget Report" project, the agent will keep relevant data (prior budgets, financial terms, user's reporting style) in project memory. Organizing memories by category or project helps the system quickly filter relevant information when needed. Advanced memory systems support grouping and filtering by metadata — essentially tagging memories by project or category — and by user or session, so the AI can retrieve content relevant to the current project without confusion.

- Task Memory (Working Memory): Task memory is the short-term, highly dynamic memory that pertains to the *current task or conversation step*. This is like the AI's working memory, which it uses to handle the immediate question or the next action. It might include the last few user queries, the intermediate conclusions drawn, or the current step in a multi-step solution. Task memory is typically simpler and more transient than the other layers — once the task is finished or the conversation moves on, this memory can be cleared or distilled into the higher-level memories if important. The task memory should be kept lean and focused (only the necessary details for the current step) to avoid cluttering the context window. In practice, this could be managed by the state module (see next section) which keeps track of the current step and relevant recent info. It aligns with how human working memory is limited and task-specific.

Each memory partition serves a distinct role, but they work together. For example, if a user asks a question, the agent might retrieve personal memory (to recall the user's background or earlier conversations) and project memory (to gather domain-

specific info) into the task working memory, and then answer using that combined context. By (e.g. not every past nt recall. Storing and searching memories with appropriate tags or vectors ensures the assistant can rapidly fetch what's needed — potentially using optimized indices or even graph-structured stores for efficiency. In fact, one can enhance retrieval by organizing knowledge in a graph or category hierarchy (more on this later), which, combined with embedding-based search, allows pinpointing relevant pieces with minimal noise.

Finally, keeping the task memory simpler means the agent's immediate context remains manageable. The agent might summarize or distill lengthy project context into concise notes for the task at hand. Complex details can be offloaded to long-term memory and only fetched when required, ensuring the prompt sent to the language model isn't overloaded. This also helps with token limitations and focuses the model on the pertinent details of the current step.

Goal-Driven State Transitions and Tool-Aware Processing

Designing state transitions means defining how the agent moves from one step or subtask to the next. A well-designed agent doesn't just react randomly; it maintains an internal state that includes what the user asked, what the current goal is, what sub-goals exist, and which tools or functions have been used or are available. The state module (as shown in the architecture diagram above) serves as the agent's "memory" of the ongoing process — it can be as simple as a record of the conversation so far and the next action to take, or as complex as a finite-state machine or workflow tree that the agent traverses.

Goals and Constraints: The agent's state transitions should always be influenced by the overarching *goal* it's trying to achieve (e.g. answering a question, completing a multi-step task) and any *constraints* (time limits, user instructions like "don't use external sites", required accuracy, etc.). At each step, the agent evaluates "Where am I relative to the goal? What's the next best action given the constraints?". This goal-driven approach ensures the agent stays on track and makes decisions that bring it closer to completion. For instance, if the goal is to create a summary report, the agent's states might include: gathering data, analyzing data, drafting the report, and reviewing it. Constraints (like "data must be from 2021 or later" or "don't exceed 1000 words") will affect these states by, say, filtering which tools to use or how to format the output.

Tool Awareness: Modern AI agents can use various tools (search engines, calculators, [illegible] agents. The state l[illegible] *provides*. This means before transitioning to a state where a tool is invoked, the agent ensures it has the necessary input for that tool, and after the tool runs, the agent captures the tool's output into the state. For example, if one tool is a web search, the agent must first formulate a good query (possibly based on the current goal and context) and once search results are obtained, integrate those results into the working state (e.g. by summarizing or storing key findings). If multiple tools operate in parallel (more on parallelization later), the state must track each tool's status and results.

Notably, the agent should avoid redundant actions by considering what it already knows or has retrieved. If Tool A has fetched some data, the state can pass that data to Tool B instead of Tool B fetching the same data again. By making the state a central "blackboard" of information, different tools and subprocesses can share context. This is analogous to a team where members share notes — the *state* is the shared notes board where each tool/agent writes its output and reads any info it needs.

Decision Making and Transitions: With goals and tool outputs in mind, the agent makes decisions on what to do next. This could involve branching logic: *If* the needed information was found, move to a synthesis step; *if not*, perhaps try a different approach or ask the user for clarification. The state machine may have conditional transitions (like in a flowchart). For instance, *"If user has not provided an essential detail, go to a state where the agent asks a clarifying question."* In fact, an effective agent will proactively ask the user questions to fill knowledge gaps (much like the game 20-Questions or Akinator strategy to narrow down possibilities). This proactive questioning is a state transition triggered by recognizing an unknown in the goal context.

Modern AI orchestration frameworks embody this idea of goal-driven, tool-aware iteration. For example, the LLM Loop plugin demonstrates autonomous, goal-oriented task execution where the AI *persists* toward the goal by making multiple tool calls, checking results, and iterating until the task is done. In other words, unlike a single-turn prompt, the agent's controller keeps the conversation and tools moving through a sequence of states until the success criteria are met. Throughout this process, the state module is continuously updated — recording which sub-goals are completed and which remain.

In summary, state transitions should be adaptive (responsive to the evolving context and results g tool outputs int gent avoids aimless wandering and instead follows a rational workflow. This is critical for complex tasks that require reasoning through multiple steps or sources of information.

Graph-Based Hierarchical Retrieval for Fast Context

When the agent needs to recall or find information, especially from large knowledge bases, a graph-based hierarchical retrieval strategy can significantly speed up and improve relevance. Instead of treating all pieces of memory as a flat list of vectors, we introduce some structure — like a graph or hierarchy — to guide the retrieval process.

Why a Graph or Hierarchy? Human knowledge is interlinked and organized by topics; similarly, we can organize the agent's knowledge base into a graph of concepts or categories. For example, personal and project memories could be structured such that related information is connected: a "node" for a project might link to nodes for that project's requirements, timeline, and relevant people. A graph can capture relationships (e.g. *"Client X is associated with Project Y"* or *"Topic A is a subtopic of Topic B"*). This means if the current query is about Topic A, the system can quickly narrow the search space to that branch of the graph instead of scanning everything.

A hierarchical retrieval might work in two steps: first identify the most relevant *cluster or node* in the knowledge graph, then do a fine-grained vector search within that cluster for the specific details. This can be faster and more precise than searching over the entire memory every time. It also helps in cases where a keyword might be ambiguous across domains — the hierarchy provides context to disambiguate (e.g., the term "Java" under a "Programming" category vs "Java" under "Travel" for the island). By organizing information with metadata and categories (essentially a hierarchy), the system ensures queries get matched to the right context. In fact, a sophisticated memory system might use *hybrid search* (combining semantic vector search with filters for certain metadata or graph relationships). This hybrid approach yields relevant results and reduces the chance of pulling in unrelated information, thereby grounding the AI's answers in the correct context.

Graph Updates and Usage: As the agent acquires new information (through user input or dis ture. Suppose du ne." It can add that fact as a node or attribute linked to the Project Y node in the knowledge graph. Later, if the agent needs to recall deadlines, it can traverse the graph to find that info quickly. This dynamic graph update ensures the knowledge structure remains current. Over time, the graph becomes a rich network of the user's data and the agent's learnings, supporting more context-aware assistance.

It's worth noting that graph-based memory doesn't replace embeddings but complements them. The agent might still encode each node or piece of info as an embedding for semantic search, but the graph provides an additional layer of fast routing. In practice, systems like knowledge graphs or even simple taxonomies allow a quick narrowing-down of relevant data before the expensive vector similarity search is done. This results in faster retrieval times and often better precision, especially as the knowledge store grows large. By maintaining such a hierarchy, the agent achieves a kind of *structured memory* akin to how a library has categories and an index on top of the books.

Multi-Step Prompting: Parallelization vs. Serialization

Complex tasks often require the AI to perform multiple reasoning or action steps. Rather than one giant prompt that tries to do everything, it's more effective to break the process into multiple prompts or sub-tasks. This is known as *multi-step prompting* or a multi-hop approach. Within this, we can leverage both parallelization (doing some steps at the same time) and serialization (doing steps one after another in sequence) as appropriate.

Sequential (Serialized) Steps: Some tasks have natural dependencies: you must do Step A before Step B. For example, first summarize a document, then translate the summary. The agent should first prompt to get the summary, then use the result in a new prompt for translation. Sequential prompting is straightforward and ensures each step's output feeds into the next. This is how the agent gradually builds towards the final answer, refining or gathering pieces of the solution step by step.

Parallel Steps: In other cases, parts of a task can be done independently. For instance, if researching a topic that has multiple facets, the agent could formulate several queries (facets) and search for each simultaneously, rather than one by one. By running subtasks in parallel, we can significantly speed up the overall process.

The agent's coordination logic would then wait for all parallel tasks to finish and collect thei

ta about *economy, h* data-fetching prompts for each topic in parallel, then merge the findings. Many advanced research systems allow tuning this "breadth" — how many parallel searches or branches to explore at once — versus the "depth" of sequential iteration. In fact, the *Deep Research* framework explicitly lets you configure the number of parallel subqueries per iteration (breadth) and how many iterative cycles (depth) to run. This means the developer can decide to fan out wide (many parallel tasks) and/or go deep (multiple rounds of refinement), depending on the complexity of the problem.

Orchestration: Implementing multi-step prompting requires an orchestrator (often the agent module or a coordinator agent) to keep track of which prompts to send out, and when. It might use a workflow tree or plan (discussed more in the next section) to know what subtasks exist. For parallel tasks, threads or asynchronous calls can be used; for sequential, it will await one result then move on. The key is that the agent can juggle multiple contexts. Usually, the state will maintain a queue or graph of these sub-tasks. When parallel tasks complete, their outputs are inserted into the state (perhaps under their respective node in a workflow graph), and any task waiting on those results can then proceed.

It's also possible to combine approaches: e.g., do two steps in sequence, then branch into parallel tasks, then converge, etc. For example, first prompt might be "Plan how to tackle this problem," which returns a list of sub-tasks. Then the agent could execute those sub-tasks in parallel if they're independent (like "gather data from source X" and "gather data from source Y"). After that, a final prompt consolidates everything. Many research and engineering efforts with LLMs follow this idea of *plan-execute* and even *reflect*, where the AI plans a course of action, executes possibly parallel steps, and then reviews the combined results for the final answer.

By intelligently mixing serialization and parallelization, the agent improves both efficiency and thoroughness. Parallelization cuts down wait time when multiple actions can happen simultaneously, while serialization ensures that dependent logic and reasoning are done in the correct order. The result is a more robust and faster workflow, especially for complex tasks that naturally break into sub-components.

Node-Based Workflow Planning and Transparency

To manage complex multi-step processes, it's beneficial to represent the plan or workflow a Each node represents ons or dependencies. Building the workflow in a node-based format has two major advantages: internal clarity for the agent and transparency for the user.

Planning with Nodes: When a new complex query or task comes in, the agent should first create a plan — effectively a graph of nodes — before diving into execution. This plan might be generated by the LLM itself (prompted to outline steps) or by a specialized planning module. For example, given a request *"Analyze the quarterly sales and generate a report with insights"*, the agent can break this into nodes: (1) Understand the request (if clarification needed, ask user), (2) Collect sales data for the quarter, (3) Analyze data for trends, (4) Generate insights and recommendations, (5) Compile the report. These nodes can be drawn as a chain or a tree (if some steps can branch out). Each node may have additional detail, like which tool or function to use, and what the input/output should be.

Parallel and Conditional Nodes: In the node-based plan, we can mark certain nodes that can run in parallel (as discussed earlier). We can also include conditional branches — for instance, a node like "If data is insufficient, ask user for more info" might branch off if needed. This resembles a flowchart with decision diamonds (yes/no branches) and action boxes. The planning process should think of these possibilities: *what if a step fails or yields an unexpected result?* — maybe include a node to handle errors or to verify outputs (the earlier diagram mentioned a "Citation Verify" step as a kind of verification node to ensure information accuracy).

Showing the Plan to the User: Once this workflow graph is created (even if partially), the assistant can present it to the user for approval or awareness. Transparency is key for trust, and it also engages the user in the loop. The user might see something like: "Here's my plan: [Step 1: Do X, Step 2: Do Y, …]. Shall I proceed?" This gives the user a chance to correct any misunderstandings early. For example, the user might respond, "Actually, don't spend time collecting data from source B; I already have that," which saves the agent from a wrong subtask. By showing a node-based outline of the intended workflow, the agent operates more like a collaborator than a mysterious box.

Even during execution, the agent can update the user at milestones. After finishing a major node, it could report progress: "Completed data collection. Next, I will

analyze trends." This concept was hinted in the design notes as *"Milestone AI: Update user on prog* orm the user and m ates ensure the process stays aligned with user expectations and can be adjusted if needed.

Internal Node Tracking: Internally, representing each step as a node (possibly in a data structure or an internal graph) helps the agent keep track of complex workflows. It knows which nodes are done, which are pending, and which resulted in what data. If something goes wrong or if new info comes to light (e.g., the user adds a new requirement mid-task), the agent can modify the graph — add a node, skip nodes, or reorder if necessary. This flexible graph-based plan can evolve, and the state management becomes easier when each piece of work is compartmentalized in a node. It's similar to how project managers track tasks in a project: each task (node) has a status and dependencies; the agent here is doing the same for cognitive tasks.

In summary, node-based workflow planning gives structure to the agent's reasoning. It enables parallel and conditional logic clearly, provides a mechanism for user transparency and feedback, and allows dynamic adjustments. By the time the final answer is delivered, the user will have essentially seen the work that went into it, which not only builds trust but also creates an opportunity for the user to learn or to correct the agent's course if it was going astray.

Continuous Memory Updates and System Customization

Building a smart agent is not a one-and-done affair — it involves continuously updating its knowledge and allowing customizations to adapt to different needs and contexts.

Graph and Vector Memory Updates: As the agent completes tasks or gains new insights, it should feed important pieces of information back into its memory stores. This happens on two fronts:

- *Vector Memory Update:* If the agent just summarized a document or found an answer through searching, it can embed that result and add it to the vector database (with appropriate tags, e.g., "Summary of Q3 report" under project XYZ). Later, when a related question comes up, the agent can quickly retrieve this summary instead of recalculating it. This learning process makes the assistant more efficient over time — it's essentially building a knowledge base through usage. Storing the right level of detail is important: perhaps save the

final answer and key evidence, rather than every trivial exchange, to keep the memor*chanism* can be URL or reference, so it can be verified later for accuracy (preventing the agent from confidently perpetuating an earlier mistake).

- *Graph Knowledge Update:* Similarly, any new entities, relationships or categories discovered can be added to the knowledge graph. If the agent learns that "Project Y's deadline is moved to November", it can update the Project Y node's "deadline" property. If a new subtopic emerges (say the user starts a new sub-project), the graph can get a new node and link it appropriately. Over time, the graph grows, but because it's structured, it remains navigable. The combination of updated vectors and an updated graph means the agent's memory keeps getting richer and remains well-organized.

Periodic pruning or summarization might also be employed: e.g., after a project concludes, the agent might summarize key outcomes and archive detailed logs, so that the active memory stays relevant and concise. The episodic memory (session transcripts) can be distilled into a summary and stored in long-term memory, rather than keeping entire raw chat logs forever.

Customizing the System ("Customize Everything"): A robust agent platform should allow customization of each major component to suit different scenarios or preferences:

- *Memory Settings:* Adjust how much to remember, how aggressively to retrieve, or what indexing techniques to use. For instance, one might configure a bias towards recent information (recency bias) if up-to-date data is crucial, or include more past context if historical consistency is needed. One can also configure grouping of memories (like by project or type) or set rules like "never forget user's name and core preferences".

- *Model and Tool Selection:* The model router in the architecture can be configured to choose different AI models depending on the task. Perhaps use a faster, cheaper model for simple or preliminary tasks and a more powerful (but slower) model for complex reasoning or final answers. This is analogous to choosing the right tool for each job. In fact, frameworks exist that allow developers to explicitly set which LLM to use for which type of sub-task. You might, for

example, use a code-specialized model for coding tasks and a language-
special … s (OpenAI, Anthro … y depending on context. The user or developer can thus fine-tune the balance between performance and expense.

- *Depth of Reasoning:* The system could expose knobs for how exhaustive the agent should be. For a quick answer, maybe limit to one or two steps of reasoning. For a deep analysis, allow the agent to do many research cycles and use multiple tools. As noted earlier, you can tweak the "depth" of multi-hop reasoning and the "breadth" of parallel exploration to match the problem at hand or the user's preference. This is an important customization for users — some might say "just give me a quick answer now" while others might prefer "take your time and be thorough, I can wait."
- *Safety and Validation:* One might customize how strict the agent is in filtering content or checking its outputs. The architecture diagram referenced a *"Validation & Safety Check Module"*. This could include adjustable settings for things like offensive content filters, or a toggle for "verify facts with an external source before finalizing an answer." Depending on the application (casual conversation vs. medical or legal advice, for example), the requirements for caution and verification differ.
- *User Interaction Style:* Some users might want verbose explanations and transparent thought processes, while others just want the answer. So the system can allow toggling the verbosity of the responses or whether the agent shows its planning nodes by default or only on request. Customization might also cover language style, level of detail, use of technical jargon, etc., which ties back to personal memory of user preferences.

Enabling such customization turns the AI agent from a one-size-fits-all solution into a flexible platform that can adapt to each user or task. This is important because an agent might be used for various purposes — from a coding assistant to a research analyst to a personal organizer — each may benefit from different configurations.

In conclusion, by partitioning memory into personal/project/task scopes, the agent retains relevant knowledge at appropriate time frames without overload. Through a goal-driven state machine that accounts for tool outputs and decision points, it navigates complex tasks methodically. A graph-structured memory with fast

retrieval ensures it finds information efficiently. Utilizing multi-step prompting with both parall problem-solving. Re ith the user adds transparency and fosters collaboration. Finally, continuously learning (updating memory) and allowing custom tweaks ensure the system improves over time and can be tailored to specific needs.

All these elements together compose a sophisticated AI assistant framework: one that remembers, plans, learns, and cooperates with the user — ultimately leading to more accurate results and a better user experience. By designing the system with these principles, we get an AI that is not just reactive, but proactively helpful, reliable, and aligned with the user's goals.

Memory | Agentic Ai | Deep Learning

Follow

Written by Jack Luo

9 followers · 12 following

ai & robotics student exploring the intersection between humanity and innovation. Writes about personal experiences, philosophy, and whatever is interesting

No responses yet

Bgerby

What are your thoughts?

More from

Jack Luo

A complete guide to getting a 400 SAT score:

By Jack

Jun 3, 2021

Jack Luo

Improving nt Learning

In this article, I would like to purpose improvements in web agents in the following areas. I will first talk about how large language...

4d ago 100

Jack Luo

The AI Projects No One's Building (But Should Be)

A Conversation About the Future of Creative Technology

19h ago

Jack Luo

200 second cat dream

I saw an Instagram story of a cat for adoption. the tabby cat in the photo looked gentle in that way that makes you think your life might...

Nov 3 1

See all from Jack Luo

Recommended from Medium

In AI Software Engineer by Joe Njenga

Anthropic Just Solved AI Agent Bloat—150K Tokens Down to 2K (Code Execution With MCP)

Anthropic just released smartest way to build scalable AI agents, cutting token use by 98%, shift from tool calling to MCP code execution

3d ago 346 23

In Level Up Coding by Fareed Khan

Building a Training Architecture for Self-Improving AI Agents

RL Algorithms, Policy Modeling, Distributed Training and more.

5d ago

Welcome back. You are signed into your member account **bg••••@jaxondigital.com**.

Sam Vaknin

How Wikipedia is Poisoning AI

By: Sam Vaknin, Brussels Morning

3d ago 50

CodeOrbit

I Built a RAG System for 100,000 Documents—Here's the Architecture

My productio[illegible]s.

Nov 1

Jonkoolx

Here's a polished draft for "Why Boredom Is the Gateway to Creativity":

Why Boredom Is the Gateway to Creativity

Aug 21 92 1

In Towards AI by Teja Kusireddy

We Spent $47,000 Running AI Agents in Production. Here's What Nobody Tells You

Multi-agent … …ropic's Model Context Protocol (MCP) are revolutionary. But…

Welcome back. You are signed into your member account **bg••••@jaxondigital.com**.

Oct 16 3.4K 114

See more recommendations