ChatGPT

Memory and Context Strategies for Multi-Agent Coding Systems

Introduction

Multi-agent coding assistants (e.g. separate frontend, backend, QA, and orchestrator agents) require robust memory mechanisms to share knowledge and maintain context over time. Large language models (LLMs) like Anthropic's Claude have a fixed context window, so we must design external memory analogous to human **working**, **short-term**, and **long-term memory** to extend their capabilities [1]. In a human brain, **working memory** holds a small amount of information we're actively thinking about, **short-term memory** retains recent information temporarily, and **long-term memory** is a durable store of knowledge. By mimicking this, we can make coding agents more reliable and efficient in remembering project details and past decisions. Modern AI frameworks and research (including Anthropic's Model Context Protocol, Retrieval-Augmented Generation, knowledge graphs, vector databases, etc.) already provide building blocks for such a memory architecture. Below, we dive deep into existing approaches and how they can be applied in a Claude Code MCP environment to improve context handling.

Human-Inspired Memory Layers in an AI System

Working Memory: In our context, working memory corresponds to the LLM's immediate prompt context – the information the model is actively "holding in mind" when generating a response [1]. This includes the current user query or task and any very recent or high-focus data (like the specific code snippet being edited). It's analogous to a coder focusing on a few lines of code or a single function at a time. Working memory is very limited in capacity (bounded by the model's token window), but it contains extremely detailed information relevant **right now**.

Short-Term Memory: Short-term memory for an AI agent spans the relevant context of the current session or task – recent dialogue turns, instructions, or intermediate results that are still in use. It ensures continuity within a conversation or coding session [2] [3]. For example, if the frontend agent has been discussing a new page design with the orchestrator, the key points from that discussion should remain in short-term memory so the agent can refer back to them a few steps later. Short-term memory in LLM-based systems often takes the form of the recent chat history or a summary of it. Unlike working memory, which is the live prompt, short-term memory could be stored externally and fetched as needed (for instance, maintaining a **rolling summary or a window of the last N interactions**). The content in short-term memory may decay or be trimmed as it grows (similar to how humans gradually forget details of a conversation unless reinforced) [2]. Managing this is crucial – too little context and the agent forgets important details, too much and it may run out of prompt space or get irrelevant information.

Long-Term Memory: Long-term memory is a persistent knowledge base that the agents can draw upon even if it hasn't been mentioned recently in the conversation [4] [3]. This includes the entire codebase, documentation, accumulated facts about the project, prior design decisions, and any knowledge that persists across sessions. In humans, long-term memory holds knowledge for days or years; in an AI, this

would be an external storage that isn't subject to the context window limit – for example, a database of facts, a vector store of embedded documents, or a knowledge graph. Long-term memory does **not** sit in the prompt at all times; instead, the system must retrieve relevant pieces from it when needed. A key challenge is **recall**: knowing which parts of the vast long-term store to pull into short-term memory for a given task. That's where techniques like semantic search or knowledge graph queries come in (more on these below). Successful long-term memory integration means the frontend dev agent can "remember" something like a utility function written weeks ago, or a decision made in a planning meeting, by querying the knowledge base, rather than us having to manually feed that into the prompt each time.

Crucially, these three layers should work together. You can imagine the orchestrator agent acting a bit like a human hippocampus – when a specialized agent is tasked with something, the orchestrator (or the agent's own logic) fetches from **long-term memory** the most relevant facts or code and loads those into the agent's **short-term context**, from which the agent then draws into the **working memory** (prompt) while generating code. By structuring memory this way, we hope to achieve both **breadth** (access to lots of knowledge) and **focus** (using only what's needed at any moment). Recent research indeed advocates such human-like memory structures for LLM agents – e.g. using a *central working memory hub* with an episodic buffer to carry information across dialogue episodes [5]. While specific implementations vary, the consensus is that treating context as a *managed resource* (with retrieval, summarization, and forgetting mechanisms) is far more effective than relying on the raw LLM context alone.

Challenges in Multi-Agent Context Management

Designing memory for a single chatbot is tricky; doing so for multiple collaborating agents adds another layer of complexity. In our scenario (frontend dev, backend dev, QA, and orchestrator agents), each agent has its own scope of work but they operate on a shared codebase and project. Some challenges include:

- **Limited Context Windows:** Each agent (backed by an LLM) can only "see" a limited amount of text at once. They cannot load the entire code repository or all conversation history into their working memory. This necessitates careful selection of what context to provide. Context engineering – deciding what information goes into each LLM call – becomes critical [6]. For example, the frontend agent doesn't need the backend database schema in memory when fixing a CSS issue, but absolutely needs API contract details when wiring up data hooks. Feeding too much irrelevance can confuse the model or exceed token limits, while feeding too little causes it to hallucinate or duplicate work.

- **Fragmented Knowledge Across Agents:** Each agent sees only parts of the whole picture. The frontend agent might not know what the backend agent discussed with the orchestrator unless there's a mechanism to share that knowledge. If not managed, this can lead to inconsistency (the frontend implements something the backend wasn't expecting) or redundant work. The memory system must allow **knowledge sharing** so that, for instance, the QA agent is aware of design decisions made earlier, or the backend agent can access the frontend's interface schema. The orchestrator can facilitate this by acting as a communication hub and by updating the shared long-term memory whenever an agent discovers or creates new information.

- **Temporal Continuity:** Multi-agent projects are long-running. An agent might be invoked days or weeks apart. Long-term memory must persist across these gaps. For instance, if the project had a discussion a month ago about choosing a tech stack, and now a new agent instance spins up (say a DevOps agent to deploy something), it should be able to recall those past decisions from long-term

memory rather than treating it as a fresh problem. This is where *persistent storage* (databases, files, knowledge bases) is essential – the agents should not rely solely on ephemeral conversation history.

- **Complex Queries and Reasoning:** Sometimes the needed context isn't a single document, but relationships between pieces of information. For example, "When the frontend asks for data X, which backend service provides it, and which database does that service use?" Answering this might require multi-hop reasoning through the knowledge base (page → API → microservice → database schema). The memory architecture should support such queries, possibly by using structured data or graph-based knowledge to capture relationships.

- **Avoiding Information Overload:** While we want completeness, we must avoid overloading the agents with too much context. Not only do tokens cost compute, but irrelevant or excessive information can degrade the quality of responses. As a best practice, many agent frameworks advise retrieving **just enough** context that is relevant to the query or task [7]. It's better to have targeted memory retrieval (even if it requires multiple steps) than to dump a huge chunk of the codebase "just in case." This ties into tools like search queries, filters, and summarizers to trim context intelligently.

- **Consistency and Updates:** The knowledge base (long-term memory) itself needs maintenance. Code is updated, decisions change, new information comes in. The memory system should allow updates – e.g. when the backend agent refactors an API, the stored knowledge about that API's parameters should be updated or marked stale. If using a knowledge graph or database, this means editing or versioning nodes; if using a vector store, perhaps re-embedding updated documents. A stale memory can be as harmful as no memory, as it might mislead agents.

Overall, the goal is to ensure each agent always has *the right context at the right time*. Techniques like retrieval-augmented generation, summarization, and knowledge graphs directly tackle these issues. Next, we'll explore those techniques and how they contribute to working, short-term, and long-term memory in practice.

Approaches for LLM Memory and Context Management

To build an efficient memory system, we can combine several state-of-the-art methods. Here's an in-depth look at relevant tools and techniques – from **retrieval augmented generation (RAG)** with vector embeddings to **knowledge graphs**, databases, and context engineering practices – and how they map onto our memory layers.

Retrieval-Augmented Generation (RAG) with Vector Stores

At its core, **RAG** is about augmenting the LLM's context with relevant data fetched from an external knowledge source at query time. The typical implementation uses a **vector database**: all reference texts (documents, code files, chat transcripts, etc.) are pre-processed into embedding vectors, and at runtime you embed the current query or task description and find similar vectors to retrieve the most relevant chunks of information. This is a powerful way to give an agent long-term knowledge on-demand [3].

For a coding agent, the knowledge base for RAG might include: the code repository split into reasonably sized chunks (functions, classes, modules), documentation (e.g. API docs, README, design docs), and perhaps transcripts of previous planning conversations. When the frontend agent needs to "add a page with data hooks," it could form an internal query like *"data hooks for similar pages"* or *"how to fetch data X from backend"*, and the vector store would return, say, a snippet from another page's code where a similar data hook is implemented, or a piece of documentation on that data source. These retrieved snippets become that agent's **short-term memory** context in the prompt.

The RAG approach effectively bridges **long-term to short-term memory**: all knowledge sits in the long-term store (the vector DB) until needed; then only the relevant pieces are pulled into short-term (the prompt) [3]. This keeps the working memory uncluttered and relevant. It's also **domain-agnostic** – semantic vector search works on natural language and code alike, capturing similarity in usage or meaning even if exact keywords differ. For instance, two code functions that don't share names but do similar things might still be near each other in embedding space (because of similar context or comments), allowing the agent to find analogues.

A concrete implementation could use tools like FAISS, Pinecone, or other vector DBs, and many frameworks (LangChain, LlamaIndex, etc.) offer ready-made modules for memory. LlamaIndex, for example, provides a `VectorMemoryBlock` which stores and retrieves chat messages or other texts from a vector database [8]. This could be extended to store code or facts as well. Each agent could have access to a shared vector store of project knowledge. The orchestrator or the agent itself would construct the search query based on the current task. Since Claude with MCP supports function calling and tools, one can imagine an MCP tool that performs a vector search query (given a text input, returns the top-N relevant snippets). Indeed, existing systems like **Cognee** (described later) integrate vector search as a tool for Claude.

Pros of RAG/Vector Memory: It's relatively straightforward and highly flexible. You don't need to hard-code relationships; the semantic search will surface relevant info even for queries you didn't anticipate. It's excellent for **unstructured knowledge** – e.g. pulling a helpful code example or explanation from documentation with just a natural language query. Also, by adjusting the embedding model or adding domain-specific fine-tuning, you can improve search accuracy for code (there are code-specific embedding models which understand syntax and semantics of code better).

Cons/Considerations: Pure vector search may sometimes return fragments that are *individually* relevant but lack the bigger picture or logical structure. For example, you might retrieve two functions that both use a variable `dataHook` but they could be from unrelated contexts – the agent then has to figure out how they connect. Without additional structure, the agent might miss a crucial piece that didn't surface in the top results. There's also the issue of keeping the vector store up-to-date as code changes (one should re-embed updated code). Another challenge is that similarity search might favor things that **literally** match the query terms, which could miss out on indirectly related info. This is why some systems combine vectors with other methods (like filtering by recency or metadata, or using knowledge graphs in tandem, as we'll see next).

Despite these caveats, a **basic RAG setup** is often the first and very effective step to give LLM agents long-term memory. For our multi-agent system, one could start by indexing the entire codebase and relevant docs. The orchestrator can ensure that whenever an agent gets a task, a preliminary "information gathering" step runs: query the vector store for keywords related to the task (e.g., "data hooks", specific component names) and feed the top results to the agent. This way, the agent isn't coding blind; it has

examples and facts from project history to guide it. Many agent frameworks do exactly this: for instance, AutoGPT and BabyAGI variants use a vector store to record intermediate results and thoughts, retrieving them when needed so the agent doesn't forget objectives or found information.

Knowledge Graphs for Structured Long-Term Memory

While vectors excel at fuzzy semantic recall, **knowledge graphs (KG)** add a complementary, structured layer of memory. A knowledge graph represents information as nodes and edges – capturing relationships explicitly. In a coding project, you could construct a graph where nodes represent entities like *pages*, *API endpoints*, *database tables*, *functions*, *classes*, *developers* etc., and edges denote relationships like "Frontend page **uses** API endpoint", "API endpoint **fetches from** database table", "Function **calls** function", or "Module **belongs to** subsystem". The graph can encode architectural and domain knowledge that is not easily expressed in plain text embeddings.

For our agents, a knowledge graph serves as a **long-term memory** that can be *queried* in a more deterministic way. If the frontend agent needs to add a data hook for, say, *SalesData*, a graph query could traverse: *Page → uses DataHook concept → provided by BackendService X → implemented by API endpoint Y → pulls from Database table Z*. This chain of reasoning can be retrieved by following edges. The agent (or more likely, a tool called on its behalf) might not literally perform graph traversals on its own, but we can expose the graph via natural language query or a specialized search. For example, an MCP tool could accept a question and translate it to graph queries, returning the results. There are already projects tackling this: **Memgraph's GraphRAG** and **Memento MCP** are two examples that integrate graph queries with LLMs to improve context retrieval.

GraphRAG architecture combines a knowledge graph with vector search to retrieve precise, context-rich information. The graph provides explicit relationships (multi-hop links between entities), while vectors handle semantic similarity for unstructured data [9] [10].

GraphRAG (Graph + RAG) is an approach where the knowledge graph is used to narrow down or enrich what the agent retrieves. Memgraph's recent update (v3.0) highlights that **graphs and vectors are a perfect match**: the KG gives explicit relational context, and vector search brings in semantically similar content; together they enable *multi-hop reasoning, fast similarity search, and dynamic context refinement* [10]. In practical terms, one might first do a vector search to find a relevant node or document in the graph, then use graph algorithms to expand to related nodes. For instance, if a vector search finds a code snippet about "SalesData API", the system could then traverse the graph to get related entities like the database or other services that connect to it, ensuring the agent sees the broader context, not just that isolated snippet.

Memento MCP is an open-source project designed for Claude's MCP environment that implements a knowledge graph memory. It uses a Neo4j graph database under the hood and even stores embeddings for nodes to allow hybrid semantic search [11] [12]. With Memento, Claude can be instructed via system prompts to use it as a memory tool – e.g. *"You have access to a knowledge graph memory; always check it for past information and use semantic_search to find relevant info"* [13]. The graph can store things like conversation summaries, facts the user taught the agent, or code relationships. Then, when asked something like *"Remind me what we decided last week about authentication"*, Claude could query the graph via Memento and retrieve the exact memory instead of relying on its finite chat history. This improves persistence and reduces hallucination, because the answer comes from a stored fact. As the Memento

documentation notes, such a system *"provides persistent memory capabilities"* and *"improved context retrieval"* across even complex knowledge graphs [14].

Applied to our coding agents: we could build a **code knowledge graph**. One way is to automatically ingest the codebase structure – many tools can parse code into a graph of function calls or imports. We could also augment it with higher-level knowledge: for example, link user stories or tasks to the code modules that implement them, link test cases to the features they validate, link developers or agents to the code sections they wrote (for accountability or style consistency). A graph can also record events: e.g., a node for "Bug #123" connected to the module it affected and the fix that was applied. This essentially becomes a continuously evolving knowledge network of the project. When an agent needs context, a targeted query can pull a subgraph relevant to the task. The orchestrator might maintain templates for common queries, like: *"Find all components related to [X]"*, *"Show the dependency chain for feature [Y]"*, etc. The returned info can then be converted into a textual form (perhaps as a list of relevant items or a brief summary) and given to the agent.

Advantages of Knowledge Graphs: They excel at questions involving relationships and constraints. They help avoid the *"scraping the surface vs diving deep"* problem [15] – unlike pure vector search which might retrieve a shallow match, a graph can ensure you follow the connections to gather a comprehensive set of facts. This reduces omissions and hallucinations, as the answer can be grounded in an actual chain of linked data. Also, if you need to update something (say a component was renamed or a decision changed), updating a node or edge in the graph ensures all future queries reflect that change. It's a single source of truth for structural knowledge. For multi-agent consistency, a knowledge graph is great because all agents query the **same graph** – so a fact updated by the backend agent is immediately accessible to the frontend agent. Anthropic's MCP vision encourages such shared resources (they mention connecting to content repositories, development tools, etc., via a common protocol) [16], and a knowledge graph server is a prime example of that approach [17].

Challenges with KGs: Setting up a knowledge graph is more involved. You either need to manually define the schema/ontology (decide what node and relation types you use) or use tools to auto-extract it from data (e.g., Cognee's **codify** pipeline can transform a Python codebase into a graph of code entities [18]). Graph queries might be less straightforward for the LLM to generate than plain text queries – though projects like Memgraph's **GraphChat** show that you can let the LLM produce Cypher (graph query language) from English and execute it [19]. Another point: the graph should be kept in sync with the latest info, which could require continuous integration into the development workflow (maybe regenerate or update parts of the graph as code changes). However, this is manageable with automation.

In summary, **Knowledge Graphs serve as a powerful long-term memory**, especially when combined with vector search. They're like the structured backbone of knowledge that ensures agents not only recall facts but understand how those facts interrelate. This is very much like human **semantic memory** – a structured understanding of how concepts relate – complementing the more associative recall of pure embeddings.

Traditional Databases and Structured Data Stores

In some cases, the information we want to store/retrieve is highly structured or numeric, where a **relational database** or a key-value store might be the best fit. Think of things like configuration tables, user permissions, performance metrics, or a simple lookup (e.g., mapping feature names to code module paths).

Rather than forcing these into a vector or graph, we can use a database and let the LLM query it via natural language interfaces.

Anthropic's Claude already demonstrated connecting to a Postgres database via MCP [20] – the LLM could directly execute read-only SQL queries to fetch data. For a coding assistant, one might maintain certain **project metadata** in a database. For example, a table of all API endpoints with columns for which service owns them, what data they return, last updated date, etc. If the agent needs "the latest endpoints related to sales data", it could formulate an SQL (or use a specialized function) to query that table (e.g. `SELECT * FROM Endpoints WHERE data_topic='Sales' ORDER BY last_updated DESC`). The result could be returned in a structured format and then included in the context for the agent's reasoning. Similarly, a table mapping frontend pages to backend APIs could be queried to answer "which API does page X rely on?" without having to search unstructured text at all.

State and Memory: Databases might also be used to store the *state* of the project or conversation that doesn't fit neatly into text. For instance, the orchestrator could maintain a database of tasks done and their status, or environment variables, or configuration flags that agents should respect. Instead of hardcoding those into prompts, an agent can fetch them via a query at runtime. This keeps prompts cleaner and ensures consistency (all agents reading from the same state source).

One could also imagine using a simpler store like Redis for quick key-value memory (like caching recent results or a quick lookup by ID). For example, if an agent creates a piece of code, the orchestrator might assign it an ID and store the code snippet in Redis; later, if another agent asks for that code by ID (perhaps via a reference), it can be pulled out. This is more of an implementation detail, but it shows that not all memory needs to go through the LLM's text prompt – some can be out-of-band data passing if the environment supports it.

Integration: In a Claude MCP environment, each database or service can be an MCP server. The orchestrator (or user) can add them to Claude's tool list, and then the LLM can be instructed how to use them. We might have slash commands or special mentions for these. For example, if a MySQL MCP server is connected, the agent might do: `@mysql:query://SELECT ...` as a way to query it (MCP supports referencing resources via the `@` notation in prompts [21]). The system can fetch that and provide the result back to the LLM. This makes the experience seamless – the agent just asks in natural language, and behind the scenes it turns into a DB query.

When to use a database for memory? When the information is **highly structured, frequently updated, or requires precise querying**. A good example is *temporal data*: if you have logs of events or performance data, a time-series DB or SQL table can let the agent ask "What tests failed in the last 24 hours?" and get an accurate answer. Doing that via pure language on a massive log file would be inefficient and error-prone. Another example is to enforce constraints – if the agent should not use certain libraries, maybe have a table of allowed licenses; the agent can check that table to validate choices (rather than trusting its internal knowledge).

Hybrid Approach: Often, a knowledge graph or vector store can incorporate structured data too (for instance, node properties or metadata filters). But using a dedicated database might be simpler for certain data. There's no harm in mixing these: e.g., the orchestrator could first do a graph query to find which component is relevant, then a SQL query to fetch its latest performance metrics, then feed both results to the agent.

In summary, **databases** act as an extension of long-term memory for facts that are better handled with precise queries. They ensure the agent's answers can be as up-to-date and accurate as the underlying data. By plugging into MCP, these become just another resource the agent can draw upon, much like how a developer might run a quick SQL query to check something rather than searching through documentation. It's about using the right tool for the right type of memory.

Vector Embeddings for Conversation and Ephemeral Memory

We discussed using embeddings for static knowledge (code, docs), but they are equally useful for **dynamic memory** – i.e., things that come up during the conversation or work process that should be remembered later. For instance, if the QA agent discovers a bug and notes it, that detail can be embedded and stored so that if later an agent wonders "Were there any known issues with module X?", a semantic search can recall the QA agent's note. This is how "episodic" memory can be handled. Projects like *Generative Agents* (the simulated Sims-like agents from Stanford) recorded every observation the agent made into a vector database (with an importance score) and retrieved memories by relevance + recency when the agent needed to act [22] [23]. They found it effective to combine **recency, importance, and semantic similarity** to decide which memories to retrieve for the agent's context [24] [23]. We can apply a similar idea: maintain an **interaction memory store** of key events (with metadata like timestamps or importance tags) and use an algorithm to fetch the top-K relevant ones when needed. For example, when switching tasks or when an agent is idle for a while, the orchestrator might fetch a summary of "recent important developments" to remind the agent.

Summarization and Compression: To prevent the memory store from growing without bound or to keep it useful, we use summarization. After a lengthy discussion or after finishing a subtask, the orchestrator can prompt an agent (or use a background process) to summarize the key points of that episode and save that summary as a distilled memory (this is akin to how humans form long-term memories during rest – by consolidating the day's experiences). IBM's research on memory augmentation highlights **consolidation** as a key principle: compress information so it can be stored efficiently, merging new information with existing clusters where appropriate [25]. For instance, if over multiple interactions the agents talked about "using OAuth2 for authentication", at some point you don't need all the raw chat logs of that conversation – you just need a summary like "Decided to implement OAuth2 (for reasons A, B) on 2025-07-01" stored as a fact. This summary could be a static note in the knowledge base or a node in the knowledge graph (with relationships to the "Authentication" topic). The next time authentication comes up, the agent can retrieve this summary rather than wading through the entire transcript. Summarization can be done hierarchically: one can have daily/weekly summaries, or summary by topic, etc., building a layered long-term memory.

Working Memory Strategies: Even within a single prompt, sometimes we might chunk a task. One technique is to use an internal **scratchpad** (which can be thought of as the LLM's volatile working memory). For example, the orchestrator might ask the agent first to outline a solution (in a hidden prompt or as a plan) before actually writing code – this outline is like using a bit of working memory to structure thoughts, which then helps the final answer. Some frameworks explicitly manage this by separating the "thinking step" from the "answer step" through prompt engineering.

Another interesting method to extend working memory is using **transformer-XL or attention caching** – but that's more on the model architecture side (beyond our scope, since we can't change Claude's internal architecture). Instead, we simulate it via external means: feed the model a rolling window of context and some distilled notes of what fell out of the window.

Memory Indexing and Search: A practical tip from the generative agents research is to index memories not just by embedding, but also by key metadata (time, importance). In an implementation, one could tag each memory entry with keywords or categories. For example, a memory entry "Backend switched from REST to GraphQL" might be tagged with `{"topic": "API", "subsystem": "Backend", "date": "2025-06-30"}`. Then if the frontend agent later is working on "data hooks for a GraphQL API", a search can filter by subsystem=Backend and maybe boost recent changes. This is akin to adding a **contextual filter** on top of raw semantic similarity, increasing precision.

In LangChain and LlamaIndex, they allow combining multiple memory modules – e.g., a vector memory for semantic recall plus a **knowledge** or **static memory** for things that should always be present (like system instructions or key facts) [26] [27]. We can designate certain facts as **always include** (like a static memory block containing "Project coding style guidelines" or "Definition of done criteria"), which is analogous to a cheat-sheet pinned to the agent's short-term memory at all times.

Context Engineering Techniques

Regardless of the specific memory stores, how we **feed the context to the model** is an art in itself. **Context engineering** is about deciding which pieces of information (from all that's available) to include in the prompt for a given step [6]. Some best practices and techniques:

- **Tool/Knowledge Source Selection:** First determine *where* to get additional context. In complex systems, you might have multiple knowledge bases (code vs documentation vs user requirements) and tools (database vs vector search). An agent (or orchestrator) should pick the right source for the job [28]. For instance, if the question is about a specific error message, searching a log database might be better than a generic vector search. One way to do this is to include a step where the LLM is given a list of available tools and asked which one is most relevant. Anthropic's MCP facilitates this by standardizing tool access [16] – the agent can have a view of all connected servers (GitHub, Google Drive, Graph, etc.) and choose. We should ensure the agent's system prompt or instructions clearly enumerate what memory resources are available (e.g., "You have access to: VectorSearch (for code/documents), KnowledgeGraph, SQLDatabase, etc."). This helps it route queries appropriately.

- **Ordering and Prioritization:** If including multiple pieces of retrieved context, order them by relevance or logical flow. Perhaps list the high-importance facts first. In some cases, the order can be crucial (for example, conversation history should be chronological for coherence). Or if using both retrieved data and user input in a prompt, one might put the user query last (typical) but preface it with any vital facts. Experiments show that LLMs can be sensitive to the ordering of context pieces when there's potential token conflict or too much info [29]. So, as context engineers, we might say: "Place the summary of last discussion before the code snippet, because it frames the problem; place the coding style guide at the very top as a system instruction, so it's always considered." These decisions can affect output quality.

- **Context Size Management:** We often cannot include *all* relevant info due to token limits. Techniques include **summarization** (as discussed) and **fallbacks**. For example, retrieve, say, five documents, then summarize each in one sentence, and only include those summaries or top 2 documents fully. Or use a two-pass approach: first pass with summaries in context to get an answer outline, then maybe fetch a specific detailed piece for a second pass if needed. IBM's CAMELoT

memory module essentially did something similar internally by consolidating tokens – which improved model accuracy by letting it "see" more context in compressed form [30] [31].

- **Dynamic Context Length Adaptation:** If the conversation is short, you can afford to include more background knowledge. If it's already long, you might include less or more compressed context. The orchestrator can monitor token usage and decide when to trim older context. Anthropic's Claude has a relatively large window (especially Claude 2) but it's still finite. A strategy might be: always keep the last few user and agent messages verbatim (working memory), include a short summary of older messages (short-term mem compression), and bring in 1–2 relevant long-term facts. This layered inclusion ensures all three memory types are represented in the prompt each time, in condensed forms as needed.

- **Validation and Correction:** Memory can be wrong or outdated, so it's wise to have checks. If a retrieved fact says "Function A returns data in format X" but the agent's current situation contradicts that (maybe the code shows format Y), the agent should reconcile it. Either via prompt instructions like "If context seems inconsistent, double-check or ask for clarification," or via automated consistency checks. In a coding scenario, unit tests or static analysis could serve as a form of memory validation (e.g., the QA agent's role is partly to ensure what the frontend/backends did aligns with requirements). Any discrepancies could lead to updating the knowledge base.

- **User Feedback Loop:** Since this is an engineering environment, there might be a human overseer or user who can provide feedback ("No, we decided not to use library Z after all"). The system should capture such feedback into long-term memory (so the agent doesn't suggest Z again later). Tools for quick updates (like a command to "forget" or mark something deprecated in the knowledge base) can be handy. Claude's MCP allows real-time data injection, so one could imagine an `/update_knowledge` command that a user/dev triggers to modify the memory graph or notes.

By carefully combining these context-engineering tactics with the memory stores, we make sure the right things end up in the **working memory prompt** every time. The LLM effectively gets a curated packet of information – some from immediate interaction, some retrieved from long-term sources, all organized logically. This significantly boosts performance: it reduces irrelevant babble, prevents forgetting crucial details, and speeds up problem-solving since the agent doesn't have to derive everything from scratch.

Implementing in Claude Code's MCP Environment

Anthropic's Model Context Protocol (MCP) is tailor-made for plugging in these memory systems. In Claude's coding environment (Claude Code), you can connect various MCP servers that serve as extensions of Claude's knowledge and abilities [16]. To refine memory and context, we would set up MCP servers for our chosen memory tools and ensure each agent role knows how to use them.

1. Vector Search MCP Server: For example, you might use a tool like **Cognee's MCP server**, which can ingest data from multiple sources (code files, docs, etc.) and build a unified memory index. Cognee specifically combines vector embeddings with graph structures to give Claude a rich recall capability [32]. In practice, you'd clone the Cognee MCP server and add it to your Claude configuration [33]. Once running, Claude Desktop (or Claude Code) will list new commands/tools from Cognee – e.g., a tool to perform semantic search or to generate a knowledge graph from the repository [34]. Cognee's blog demonstrates

using a `cognify` tool to create a knowledge graph of a codebase, which then allows more precise queries (reducing hallucinations by giving structured context) [35]. For a coding agent, this means they can literally ask something like "Find usages of the getUserData function" and behind the scenes the MCP server might translate that into a graph or vector query and return a result.

2. Knowledge Graph Memory via MCP: We could also integrate **Memento MCP** or a similar knowledge graph service. By configuring Claude's `claude_desktop_config.json` to include the Memento server (as shown in the Memento documentation) [12] [36], the agent then has access to persistent graph memory. We would likely include system instructions to the agents like: *"You have a persistent memory graph. When you need to recall a past conversation or fact, use the memory search tool."* [13] This way, the agent is aware of this long-term memory and will invoke it proactively. In use, if the user or orchestrator says "Remember that earlier design choice about caching?", the agent could do a semantic search in the Memento graph for "caching design choice" and retrieve the stored memory of that discussion. The result might come back as a snippet: "On 2025-06-20, decided to add Redis caching to improve response time, but to invalidate on each deploy." The agent can then incorporate that in its answer or code (e.g., ensuring to include caching logic or at least not to propose an already-rejected idea).

3. Orchestrator as Memory Manager: The orchestrator agent in the Claude environment can take on the role of a **memory manager and context supplier**. Since it coordinates the other agents, it can intercept their tasks and do preparatory steps. For instance, when a task comes in for the frontend agent ("Add page X with data hook to Y"), the orchestrator could: - Query the vector store MCP for "page similar to X" or "data hook Y usage" and gather results. - Query the knowledge graph for "dependencies of Y" (maybe find what backend provides Y). - Summarize these findings into a concise brief: *"Context: Feature Y is provided by BackendServiceB (API `/getYData`). Similar implementation found in PageW.vue for reference. Last updated 2 weeks ago. Remember to follow pattern from PageW."* - Provide this brief to the frontend agent along with the original task request. This brief essentially fills the agent's **short-term memory** with relevant long-term info, so that when the agent generates code, it can directly incorporate those details (e.g., calling the right API endpoint, following the style from the reference page).

The orchestrator can do such memory retrieval using the MCP tools available. Claude's MCP allows the orchestrator (which could be an LLM function or just our controlling logic) to call `claude mcp` commands or to mention `@server:resource` in a prompt that it sends to an agent [21]. For example, the orchestrator could send to the frontend agent: *"Here is the API documentation you need: @docs:file://api/getYData"* if a docs MCP server is connected, and Claude will automatically attach that file content to the prompt [37]. This is a powerful way to inject knowledge – effectively *attaching documents on the fly* from the long-term store.

4. Role-Specific Memory: Each agent might also have some dedicated memory or tools. For instance, the QA agent could have a connection to a testing database or a log file MCP server (to retrieve error logs). The backend agent might have a tool to run code or query a schema (maybe there's a PostgreSQL MCP for the database schema as given in the Anthropic example [20], so the backend agent can ask the DB about table structures directly). By tailoring some memory resources to each role, you ensure they have efficient access to the information they most often need. The orchestrator's job is to make sure cross-cutting info is shared (like if the frontend and backend need to sync on data models, orchestrator might fetch the model definition and give it to both).

5. Updating Memory: With multiple agents making changes (especially code changes), we need to update the long-term memory. In a Claude Code setting, after an agent writes code, we could have a git MCP or file system MCP that the agent can use to *read/update files*. Claude Code actually has built-in tools like `edit` or `ls` which can navigate a project's files. So when code is added or changed, the orchestrator could instruct an agent (or use Claude's auto functions) to update the knowledge base. For example, after adding "Page X", run a script to embed that new file into the vector store so it's searchable; or update the knowledge graph by adding a node for "Page X" and linking it to its data source. Some of this can be automated if we integrate the memory update into the code commit process.

6. Example Workflow: To illustrate, let's walk through a mini-scenario: - **Task:** *"Frontend agent, create a new report page that uses sales data (from SalesService)."* - **Orchestrator preparation:** Orchestrator searches the memory: - Vector search for "sales data report page" might retrieve an older "RevenueReport" page code. - Knowledge graph query might find that SalesService is a backend service providing `/salesData` API and that there's a database table `SalesRecords`. - Orchestrator gathers: snippet of `RevenueReport.vue`, snippet of API spec for `/salesData`, note that "SalesService→SalesRecords DB". - **Prompt assembly:** Orchestrator sends to Frontend agent something like: 1. System message: "You are a Frontend Dev Agent. Your job is to implement UI features. You have tools: code editor, vector_search, knowledge_graph." (ensuring it knows it can use them). 2. Context attachment: include the retrieved code snippet and API info (perhaps as an attached file or within the prompt, summarized if lengthy). 3. User message (task): "Implement a page `SalesReport.vue` that fetches sales data from `SalesService` and displays it. Use the same data hook pattern as in RevenueReport. Ensure you call the endpoint correctly and handle the response." - **Agent action:** The Frontend agent now has everything needed in working memory: it knows where to get the data and has a pattern to follow. It writes the code accordingly. If it needs more info (say how to format dates in the report), it might call the vector search tool itself (e.g., search for "date format util") and retrieve an answer (this could happen iteratively within Claude Code environment). - **Post-task:** Orchestrator verifies the code (maybe the QA agent runs tests). Once confirmed, orchestrator updates the memory: index `SalesReport.vue` into the vector DB, and add a node in the graph linking `SalesReport.vue` to `SalesService` and tag it as a "ReportPage".

This cycle demonstrates reduced workload: the agent doesn't waste time scanning the entire codebase for examples – the memory system proactively provided it. And it's reliable because it likely used the correct API (since the actual spec was given) rather than guessing.

Additionally, the presence of these memory tools can prevent hallucinations. If the agent is asked a question about the project, well beyond what's in the current prompt, it can fall back on the memory search. For instance, *"Backend agent, what does SalesService do?"* – if that wasn't in the prompt, a naive LLM might bluff an answer. But with a memory system, the agent can query the knowledge graph or docs for SalesService and return a factual answer [38]. The system is thereby more **grounded** in reality.

Options and Recommendation Mix

Bringing it all together, here are a few configurations for memory and context, ranging from simpler to more advanced – each with its pros/cons. You can choose one or even blend aspects of each (as the user suggests, a mix of all might yield the best results):

- **Option 1: Vector RAG-Only (Baseline)** – Use a vector database to store all long-term knowledge (code, docs, decisions). Whenever an agent needs context, perform a semantic search and feed the top results. This is relatively easy to implement and has immediate benefits. Many open-source tools (LangChain, etc.) support this pattern out-of-the-box. It addresses the basic memory need: agents will recall relevant past info rather than working from scratch. However, it may not capture complex relationships and could require tuning to get the best results (e.g., chunk sizes, embedding choice). If you want minimal complexity, start here. It's basically short-term memory on demand from a long-term store [3].

- **Option 2: Vector + Summarization (Enhanced RAG)** – Improve on Option 1 by adding context engineering: maintain summaries of important past interactions and store those as well (perhaps even pin some summaries in the prompt if always needed). Also implement a scheme to periodically compress old context into long-term memory (so the system "learns" from each session). This helps manage context window limits. For example, after each day of coding, have the orchestrator summarize what was done and store that summary with a high importance score. Later, retrieval can pull from these summaries first (faster and briefer). This option stays relatively simple (still mostly vector operations) but improves efficiency and the **quality** of context by avoiding raw clutter. It aligns with human memory's consolidation process [25].

- **Option 3: Hybrid Knowledge Graph + Vector (GraphRAG)** – Implement a knowledge graph for the project and integrate it with vector search for a hybrid memory system [10]. This could be done via an existing solution like Memgraph or Cognee, or custom-building a Neo4j with embeddings. The graph adds a powerful way to answer structured queries and retrieve connected info, reducing the chance of missing relevant pieces. The vector part ensures flexibility for unstructured queries. This approach is more complex – you'd need to set up the graph (defining what goes into it) and possibly maintain two stores (graph + vector or a graph with vector index). The payoff is a much richer understanding for the agents. It's recommended for larger projects or those with intricate domain logic where relationships matter (e.g., many microservices, data lineage, etc.), because it will help the AI agents to navigate those relationships reliably [9]. Given that you use Claude's MCP, adopting something like Memento (knowledge graph memory) is quite feasible – the heavy lifting (vector indexing inside the graph, MCP integration) is already done in that tool [36]. So this option could leverage existing libraries.

- **Option 4: Full Mix – Graph + Vector + Database + Orchestrated Context** – This is the "all of the above" solution for maximum robustness. In this setup, you maintain:

- A **knowledge graph** for core project knowledge and schema.
- A **vector store** for all textual content and code embeddings.
- One or more **databases** or file-based stores for any specialized data (like logs, configurations).
- A smart **orchestrator** that uses context engineering to select which source to query and how to assemble the context for each agent.

- Agents that have access (via MCP tools) to all these resources and are instructed when to use which.

Such a system might, for example, use the knowledge graph to figure out *what components are relevant* to a query, then use vector search to grab the detailed docs for those components, use a database query to fetch latest stats for them, and finally give all that to the agent in a nicely formatted prompt. This yields very rich, accurate context and should minimize instances of the agent saying "I don't know" or giving wrong info – because if the info exists anywhere, the orchestrator can likely fetch it. The trade-off is complexity: maintaining consistency between the graph, vectors, and database requires disciplined updates (possibly automated with your development workflow). Also, more moving parts mean more can go wrong, so robust monitoring is needed (e.g., if a vector search fails, maybe fall back to a keyword search as Memento does [39]). But if done right, this approach provides the **most effective memory** – akin to a human who has a well-organized library of knowledge and a sharp memory of experiences to draw from.

- **Option 5: Role-Specific Memory Variations** – A variant to consider is giving each agent a slightly different view or slice of the memory. For instance, the frontend agent's long-term memory access might prioritize UI/UX documents, while the backend agent prioritizes architecture docs. They all share the global knowledge base, but the orchestrator (or their prompts) could filter what they fetch. This can simplify each agent's job – they won't be overwhelmed with irrelevant info outside their domain. It's like each specialist having their own "expert knowledge cache" plus access to general knowledge when needed. This is an optional tweak and can be combined with any of the above options.

Finally, it's worth noting that these strategies are not mutually exclusive. In fact, most advanced systems use a combination. For example, even with a knowledge graph in place, one might still use summarization for very long chains of reasoning (the graph might tell you which nodes are related, but you might still summarize the content of those nodes if they contain long text like meeting notes). The key is **balance** – use the simpler solution when it works, and the more complex ones when necessary. Often, an incremental approach is wise: start with RAG, observe where it fails (maybe it missed a linked concept or gave a dated answer), then consider adding a graph to encode that linkage or a database to hold updated facts.

Conclusion

Incorporating human brain-inspired memory layers into a multi-agent coding system can dramatically enhance its performance and reliability. By dividing knowledge into working memory (immediate context), short-term memory (recent, relevant info for the current session), and long-term memory (persistent knowledge base), we ensure the AI agents have both the detail-oriented focus and the big-picture awareness needed for complex tasks. Modern techniques like retrieval augmented generation (for semantic recall of facts) and knowledge graphs (for structured reasoning over knowledge) allow us to implement these memory layers in practice [10] [32].

Crucially, in the Claude Code MCP environment you use, these concepts are not just theoretical – they can be realized with existing tools. Anthropic's MCP framework enables seamless integration of external memory: the agents can fetch documentation, search a knowledge graph, query databases, or read project files on demand, all within their prompts [16] [40]. By leveraging MCP-compatible servers (like Cognee for knowledge graphs + vectors, or Memento for persistent graph memory), you can give your coding agents a form of "long-term memory" that persists across sessions and a mechanism to load the right context into their "short-term memory" when needed [13] [32]. This means, for example, your frontend dev agent will

automatically recall relevant backend details when implementing a feature, just as a human developer would recall conversations with colleagues or past projects.

From the options discussed, you have the flexibility to choose a strategy that fits your team's needs: - If you prefer simplicity, start with a vector-based retrieval system and some summarization – you'll cover a lot of ground with minimal setup. - If you need more precision and your project has complex interdependencies, consider adding a knowledge graph to capture those relations explicitly. - If maximum efficiency is the goal, combine approaches to cover all bases, ensuring no important context is ever missing when an agent is working.

Remember that **context engineering is iterative**. Monitor your agents: if you notice them getting confused or asking for info that was given, maybe the context window was overloaded – you might then tighten the short-term memory or improve summaries. If they hallucinate answers that contradict known facts, that's a sign to integrate the knowledge base more tightly (perhaps via system prompts reminding them to check the memory tools first [13]). The field of LLM memory is rapidly evolving (with new research emerging on augmenting LLMs with explicit memory modules and better long-context handling), so keep an eye out for new techniques. But the fundamental principles remain consolidation, relevant retrieval, and iterative refinement [25] [22].

By taking inspiration from the human brain and leveraging existing AI tooling, we can substantially **reduce our workload** (less time spent re-feeding information or fixing inconsistencies) and **improve what already works** (build on proven methods like RAG and knowledge graphs rather than reinventing the wheel). The end result is a multi-agent system that feels much more like a cohesive, intelligent team – one where the right hand knows what the left hand did, and where past knowledge is never truly lost. Each agent can operate with a limited immediate focus (working memory) yet draw from a vast shared brain (long-term memory) when needed, which is a recipe for a reliable and efficient coding assistant system.

Sources: The approaches above draw on recent literature and tools in the AI community, such as retrieval-based memory for LLMs [3], cognitive-inspired memory frameworks [5], generative agents' memory retrieval strategies [22] [23], IBM Research's work on long-term memory modules [25], and practical implementations like GraphRAG and Cognee that merge graph databases with vector search for context engineering [10] [32]. These demonstrate the effectiveness of mixing multiple knowledge representations to achieve human-like context retention and are directly applicable to building your system. Each tool (MCP servers, vector DB, graph DB) addresses a facet of the memory challenge, and together they can fulfill the vision of a robust working, short-term, and long-term memory system for your AI agents. [1] [10]

---

[1] [25] [30] [31] How memory augmentation can improve large language models - IBM Research
https://research.ibm.com/blog/memory-augmented-LLMs

[2] [4] Short-Term vs. Long-Term LLM Memory: When to Use Prompts vs. Long-Term Recall? – RandomTrees – Blog
https://randomtrees.com/blog/short-term-vs-long-term-llm-memory-prompts-vs-recall/

[3] [6] [7] [8] [26] [27] [28] [29] Context Engineering - What it is, and techniques to consider — LlamaIndex - Build Knowledge Assistants over your Enterprise Data
https://www.llamaindex.ai/blog/context-engineering-what-it-is-and-techniques-to-consider

5 [2312.17259] Empowering Working Memory for Large Language Model Agents
https://arxiv.org/abs/2312.17259

9 10 15 19 Memgraph 3.0 Is Out: Solve the LLM Context Problem
https://memgraph.com/blog/memgraph-3-graph-database-llm-context-problem

11 12 13 14 36 38 39 GitHub - gannonh/memento-mcp: Memento MCP: A Knowledge Graph Memory System for LLMs
https://github.com/gannonh/memento-mcp

16 18 32 33 34 35 Cognee - Model Context Protocol + Cognee: LLM Memory Made Simple
https://www.cognee.ai/blog/deep-dives/model-context-protocol-cognee-llm-memory-made-simple

17 Knowledge Graph Memory MCP Server by Anthropic - PulseMCP
https://www.pulsemcp.com/servers/modelcontextprotocol-knowledge-graph-memory

20 21 37 40 Model Context Protocol (MCP) - Anthropic
https://docs.anthropic.com/en/docs/claude-code/mcp

22 23 24 Paper Review: Generative Agents: Interactive Simulacra of Human Behavior | by Andrew Lukyanenko | Medium
https://artgor.medium.com/paper-review-generative-agents-interactive-simulacra-of-human-behavior-cc5f8294b4ac