== README.md ==

Re

A Redis-powered memory server built for AI agents and applications. It manages both conversational context and long-t

Features

- **Short-Term Memory**
  - Storage for messages, token count, context, and metadata for a session
  - Automatically and recursively summarizes conversations
  - Client model-aware token limit management (adapts to the context window of the client's LLM)
  - Supports all major OpenAI and Anthropic models

- **Long-Term Memory**
  - Storage for long-term memories across sessions
  - Semantic search to retrieve memories with advanced filtering system
  - Filter by session, namespace, topics, entities, timestamps, and more
  - Supports both exact match and semantic similarity search
  - Automatic topic modeling for stored memories with BERTopic
  - Automatic Entity Recognition using BERT

- **Other Features**
  - Namespace support for session and long-term memory isolation
  - Both a REST interface and MCP server

System Diagram


Project Status and Roadmap
Project Status: In Development, Pre-Release

This project is under active development and is **pre-release** software. Think of it as an early beta!

Roadmap
- [x] Long-term memory deduplication and compaction
- [ ] Configurable strategy for moving session memory to long-term memory
- [ ] Authentication/authorization hooks
- [x] Use a background task system instead of `BackgroundTask`
- [ ] Separate Redis connections for long-term and short-term memory

REST API Endpoints

The following endpoints are available:

- **GET /health**
  A simple health check endpoint returning the current server time.
  Example Response:
  ```json
  {"now": 1616173200}
  ```

- **GET /sessions/**
  Retrieves a paginated list of session IDs.
  Query Parameters:
  - `page` (int): Page number (default: 1)
  - `size` (int): Number of sessions per page (default: 10)
  - `namespace` (string, optional): Filter sessions by namespace.

- **GET /sessions/{session_id}/memory**
  Retrieves conversation memory for a session, including messages and
  summarized older messages.
  Query Parameters:
  - `namespace` (string, optional): The namespace to use for the session
  - `window_size` (int, optional): Number of messages to include in the response (default from config)
  - `model_name` (string, optional): The client's LLM model name to determine appropriate context window size
  - `context_window_max` (int, optional): Direct specification of max context window tokens (overrides model_name)

- **POST /sessions/{session_id}/memory**
  Adds messages (and optional context) to a session's memory.

  Request Body Example:
  ```json
  {
    "messages": [
      {"role": "user", "content": "Hello"},
      {"role": "assistant", "content": "Hi there"}
    ]
  }
  ```

- **DELETE /sessions/{session_id}/memory**
  Deletes all stored memory (messages, context, token count) for a session.

- **POST /long-term-memory/search**
  Performs semantic search on long-term memories with advanced filtering options.
  Request Body Example:
  ```json
  {
    "text": "Search query text",
    "limit": 10,
    "offset": 0,
    "session_id": {"eq": "session-123"},
    "namespace": {"eq": "default"},
    "topics": {"any": ["AI", "Machine Learning"]},
    "entities": {"all": ["OpenAI", "Claude"]},
    "created_at": {"gte": 1672527600, "lte": 1704063599},
    "last_accessed": {"gt": 1704063600},
    "user_id": {"eq": "user-456"}
  }
  ```

  Filter options:
  - Tag filters (session_id, namespace, topics, entities, user_id):
    - `eq`: Equals this value
    - `ne`: Not equals this value
    - `any`: Contains any of these values
    - `all`: Contains all of these values

  - Numeric filters (created_at, last_accessed):
    - `gt`: Greater than
    - `lt`: Less than
    - `gte`: Greater than or equal
    - `lte`: Less than or equal
    - `eq`: Equals
    - `ne`: Not equals
    - `between`: Between two values

MCP Server Interface
Agent Memory Server offers an MCP (Model Context Protocol) server interface powered by FastMCP, providing tool-based

- **set_working_memory**: Set working memory for a session (like PUT /sessions/{id}/memory API). Stores structured me
- **create_long_term_memories**: Create long-term memories directly, bypassing working memory. Useful for bulk memory
- **search_long_term_memory**: Perform semantic search across long-term memories with advanced filtering options.
- **memory_prompt**: Generate prompts enriched with session context and long-term memories. Essential for retrieving

Command Line Interface

The `agent-memory-server` provides a command-line interface (CLI) for managing the server and related tasks. You can

Available Commands

Here's a list of available commands and their functions:

`version`
Displays the current version of `agent-memory-server`.
```bash
agent-memory version
```

`api`
Starts the REST API server.
```bash
agent-memory api [OPTIONS]
```
Options:
*   `--port INTEGER`: Port to run the server on. (Default: value from `settings.port`, usually 8000)
*   `--host TEXT`: Host to run the server on. (Default: "0.0.0.0")
*   `--reload`: Enable auto-reload for development.

Example:
```bash
agent-memory api --port 8080 --reload
```

`mcp`
Starts the Model Context Protocol (MCP) server.
```bash
agent-memory mcp [OPTIONS]
```
Options:
*   `--port INTEGER`: Port to run the MCP server on. (Default: value from `settings.mcp_port`, usually 9000)
*   `--sse`: Run the MCP server in Server-Sent Events (SSE) mode. If not provided, it runs in stdio mode.

Example (SSE mode):
```bash
agent-memory mcp --port 9001 --sse
```
Example (stdio mode):
```bash
agent-memory mcp --port 9001
```

`schedule-task`
Schedules a background task to be processed by a Docket worker.
```bash
agent-memory schedule-task <TASK_PATH> [OPTIONS]
```
Arguments:
*   `TASK_PATH`: The Python import path to the task function. For example: `"agent_memory_server.long_term_memory.com

Options:
*   `--args TEXT` / `-a TEXT`: Arguments to pass to the task in `key=value` format. Can be specified multiple times.

Example:
```bash
agent-memory schedule-task "agent_memory_server.long_term_memory.compact_long_term_memories" -a limit=500 -a namespac
```

`task-worker`
Starts a Docket worker to process background tasks from the queue. This worker uses the Docket name configured in set
```bash
agent-memory task-worker [OPTIONS]
```
Options:
*   `--concurrency INTEGER`: Number of tasks to process concurrently. (Default: 10)
*   `--redelivery-timeout INTEGER`: Seconds to wait before a task is redelivered to another worker if the current wor

Example:
```bash
agent-memory task-worker --concurrency 5 --redelivery-timeout 60
```

`rebuild_index`
Rebuilds the search index for Redis Memory Server.
```bash
agent-memory rebuild_index
```

`migrate_memories`
Runs data migrations. Migrations are reentrant.
```bash
agent-memory migrate_memories
```

To see help for any command, you can use `--help`:
```bash
agent-memory --help
agent-memory api --help
agent-memory mcp --help
# etc.
```

Getting Started

Installation

First, you'll need to download this repository. After you've downloaded it, you can install and run the servers.

This project uses [uv](https://github.com/astral-sh/uv) for dependency management.

1. Install uv:
   ```bash
   pip install uv
   ```

2. Install the package and required dependencies:
   ```bash
   uv sync
   ```

2. Set up environment variables (see Configuration section)

Running

The easiest way to start the REST and MCP servers is to use Docker Compose. See the Docker Compose section of this fi

But you can also run these servers via the CLI commands. Here's how you
run the REST API server:
```bash
uv run agent-memory api
```

And the MCP server:
```
uv run agent-memory mcp --mode <stdio|sse>
```

NOTE: With uv, prefix the command with `uv`, e.g.: `uv run agent-memory --mode sse`. If you installed from source

Docker Compose

To start the API using Docker Compose, follow these steps:

1. Ensure that Docker and Docker Compose are installed on your system.

2. Open a terminal in the project root directory (where the docker-compose.yml file is located).

3. (Optional) Set up your environment variables (such as OPENAI_API_KEY and ANTHROPIC_API_KEY) either in a .env file

4. Build and start the containers by running:
   docker-compose up --build

5. Once the containers are up, the REST API will be available at http://localhost:8000. You can also access the inter

6. To stop the containers, press Ctrl+C in the terminal and then run:
   docker-compose down

Using the MCP Server with Claude Desktop, Cursor, etc.
You can use the MCP server that comes with this project in any application or SDK that supports MCP tools.

Claude


For example, with Claude, use the following configuration:
  ```json
  {
    "mcpServers": {
      "redis-memory-server": {
          "command": "uv",
          "args": [
              "--directory",
              "/ABSOLUTE/PATH/TO/REPO/DIRECTORY/agent-memory-server",
              "run",
              "agent-memory",
              "-mcp",
              "--mode",
              "stdio"
          ]
      }
    }
  }
  ```
NOTE: On a Mac, this configuration requires that you use `brew install uv` to install uv. Probably any method tha
command globally accessible, so Claude can find it, would work.

Cursor

Cursor's MCP config is similar to Claude's, but it also supports SSE servers, so you can run the server in SSE mode a

  ```json
  {
    "mcpServers": {
      "redis-memory-server": {
        "url": "http://localhost:9000/sse"
      }
    }
  }
  ```

Configuration

You can configure the MCP and REST servers and task worker using environment
variables. See the file `config.py` for all the available settings.

The names of the settings map directly to an environment variable, so for
example, you can set the `openai_api_key` setting with the `OPENAI_API_KEY`
environment variable.

Running the Background Task Worker

The Redis Memory Server uses Docket for background task management. There are two ways to run the worker:

1. Using the Docket CLI

After installing the package, you can run the worker using the Docket CLI command:

```bash
docket worker --tasks agent_memory_server.docket_tasks:task_collection --docket memory-server
```

You can customize the concurrency and redelivery timeout:

```bash
```

```
docket worker --tasks agent_memory_server.docket_tasks:task_collection --concurrency 5 --redelivery-timeout 60 --docke
```

NOTE: The name passed with `--docket` is effectively the name of a task queue where
the worker will look for work. This name should match the docket name your API server
is using, configured with the `docket_name` setting via environment variable
or directly in `agent_memory_server.config.Settings`.

Memory Compaction

The memory compaction functionality optimizes storage by merging duplicate and semantically similar memories. This im

Running Compaction

Memory compaction is available as a task function in `agent_memory_server.long_term_memory.compact_long_term_memories
by running the `agent-memory schedule-task` command:

```bash
agent-memory schedule-task "agent_memory_server.long_term_memory.compact_long_term_memories"
```

Key Features

- **Hash-based Deduplication**: Identifies and merges exact duplicate memories using content hashing
- **Semantic Deduplication**: Finds and merges memories with similar meaning using vector search
- **LLM-powered Merging**: Uses language models to intelligently combine memories

Running Migrations
When the data model changes, we add a migration in `migrations.py`. You can run
these to make sure your schema is up to date, like so:

```bash
uv run agent-memory migrate-memories
```

Development

Running Tests
```bash
uv run pytest
```

Contributing
1. Fork the repository
2. Create a feature branch
3. Commit your changes
4. Push to the branch
5. Create a Pull Request

== improving_test_coverage.md ==

Imp

Goal

Improve test coverage to 75%, focusing on files with important application logic.

Current test coverage

```
================================ tests coverage ================================
_________________ coverage: platform darwin, python 3.12.9-final-0 _________________

Name                                       Stmts   Miss  Cover   Missing
------------------------------------------------------------------------
agent_memory_server/__init__.py                1      0   100%
agent_memory_server/api.py                   149     38    74%   129, 150-215, 246, 338, 371
agent_memory_server/cli.py                   111    111     0%   6-237
agent_memory_server/client/__init__.py         0      0   100%
agent_memory_server/client/api.py            200     66    67%   141, 176, 181, 184, 187, 234, 279-298, 339-368, 437, 
agent_memory_server/config.py                 41      2    95%   15-16
agent_memory_server/dependencies.py           24      2    92%   43-44
agent_memory_server/dev_server.py              7      7     0%   4-14
agent_memory_server/docket_tasks.py           16     16     0%   5-49
agent_memory_server/extraction.py            135     49    64%   60-63, 138-142, 172, 270-346
agent_memory_server/filters.py               179     88    51%   20, 22, 24, 26, 32-38, 54, 56, 58, 60, 66, 70, 74-76, 
agent_memory_server/healthcheck.py             7      0   100%
agent_memory_server/llms.py                  121     41    66%   208-213, 224-267, 275, 292, 295-299, 327, 329, 348-35
agent_memory_server/logging.py                16      8    50%   15-38
agent_memory_server/long_term_memory.py      549    295    46%   108-117, 161, 170, 176, 195, 212-216, 286-574, 605-64
agent_memory_server/main.py                   85     85     0%   1-181
agent_memory_server/mcp.py                   113     38    66%   57-83, 106-109, 114, 123, 129, 135-142, 146-153, 227-
agent_memory_server/messages.py               81     55    32%   61-87, 111-165, 178-186
agent_memory_server/migrations.py             74     74     0%   5-135
agent_memory_server/models.py                121      4    97%   338, 344, 347, 353
agent_memory_server/summarization.py          88     76    14%   41-110, 135-233
agent_memory_server/test_config.py            46     46     0%   1-60
agent_memory_server/utils/__init__.py          0      0   100%
agent_memory_server/utils/api_keys.py         10     10     0%   3-26
agent_memory_server/utils/keys.py             28      4    86%   17, 26, 33, 52
agent_memory_server/utils/redis.py            37      3    92%   33, 107-108
agent_memory_server/working_memory.py         59     18    69%   20-22, 42, 57-58, 87-89, 104, 109, 149-153, 170, 178-
------------------------------------------------------------------------
TOTAL                                       2298   1136    51%
```

Plan

Work progress

== refactor.md ==

Re

This plan brings the current memory server codebase in line with the new architecture: memory types are unified, memo

ULID Migration Update

Status: Completed - All ID generation now uses ULIDs

The codebase has been updated to use ULIDs (Universally Unique Lexicographically Sortable Identifiers) instead of nan

- **Client-side**: `MemoryAPIClient.add_memories_to_working_memory()` auto-generates ULIDs for memories without IDs
- **Server-side**: All memory creation, extraction, and merging operations use ULIDs
- **Dependencies**: Replaced `nanoid>=2.0.0` with `python-ulid>=3.0.0` in pyproject.toml
- **Tests**: Updated all test files to use ULID generation
- **Benefits**: ULIDs provide better sortability and are more suitable for distributed systems

Event Date Field Addition

Status: Completed - Added event_date field for episodic memories

Added proper temporal support for episodic memories by implementing an `event_date` field:

- **MemoryRecord Model**: Added `event_date: datetime | None` field to capture when the actual event occurred
- **Redis Storage**: Added `event_date` field to Redis hash storage with timestamp conversion
- **Search Support**: Added `EventDate` filter class and integrated into search APIs
- **Extraction**: Updated LLM extraction prompt to extract event dates for episodic memories
- **API Integration**: All search endpoints now support event_date filtering
- **Benefits**: Enables proper temporal queries for episodic memories (e.g., "what happened last month?")

Memory Type Enum Constraints

Status: Completed - Implemented enum-based memory type validation

Replaced loose string-based memory type validation with strict enum constraints:

- **MemoryTypeEnum**: Created `MemoryTypeEnum(str, Enum)` with values: `EPISODIC`, `SEMANTIC`, `MESSAGE`
- **MemoryRecord Model**: Updated `memory_type` field to use `MemoryTypeEnum` instead of `Literal`
- **EnumFilter Base Class**: Created `EnumFilter` that validates values against enum members
- **MemoryType Filter**: Updated `MemoryType` filter to extend `EnumFilter` with validation
- **Code Updates**: Updated all hardcoded string comparisons to use enum values
- **Benefits**: Prevents invalid memory type values and provides better type safety

REFACTOR COMPLETE!

Status: All stages completed successfully

The Unified Agent Memory System refactor has been completed with all 7 stages plus final integration implemented and

- **Unified Memory Types**: Consistent `memory_type` field across all memory records
- **Clean Architecture**: `Memory*` classes without location-based assumptions
- **Safe Promotion**: ID-based deduplication and conflict resolution
- **Working Memory**: TTL-based session-scoped ephemeral storage
- **Background Processing**: Automatic promotion with timestamp management
- **Unified Search**: Single interface spanning working and long-term memory
- **LLM Tools**: Direct memory storage via MCP tool interfaces
- **Automatic Extraction**: LLM-powered memory extraction from messages
- **Sync Safety**: Robust client state resubmission handling

Test Results: 69 passed, 20 skipped - All functionality verified

---

Running tests

Remember to run tests like this:
```
pytest --run-api-tests tests
```

You can use any normal pytest syntax to run specific tests.

Stage 1: Normalize Memory Types

Goal: Introduce consistent typing for all memory records.

Instructions:
- Define a `memory_type` field for all memory records.
  - Valid values: `"message"`, `"semantic"`, `"episodic"`, `"json"`
- Update APIs to require and validate this field.
- Migrate or adapt storage to use `memory_type` consistently.
- Ensure this field is included in indexing and query logic.

Stage 1.5: Rename `LongTermMemory*` Classes to `Memory*`

Goal: Remove location-based assumptions and align names with unified memory model.

Instructions:
- Rename:
  - `LongTermMemoryRecord` → `MemoryRecord`
  - `LongTermSemanticMemory` → `MemorySemantic`
  - `LongTermEpisodicMemory` → `MemoryEpisodic`
- Update all references in code, route handlers, type hints, and OpenAPI schema.
- Rely on `memory_type` and `persisted_at` to indicate state and type.

Stage 2: Add `id` and `persisted_at`

Goal: Support safe promotion and deduplication across working and long-term memory.

Instructions:
- Add `id: str | None` and `persisted_at: datetime | None` to all memory records.
- Enforce that:
  - `id` is required on memory sent from clients.
  - `persisted_at` is server-assigned and read-only for clients.
- Use `id` as the basis for deduplication and overwrites.

Stage 3: Implement Working Memory

Goal: Provide a TTL-based, session-scoped memory area for ephemeral agent context.

Instructions:
- Define Redis keyspace like `session:{id}:working_memory`.
- Implement:
  - `GET /sessions/{id}/memory` — returns current working memory.
  - `POST /sessions/{id}/memory` — replaces full working memory state.
- Set TTL on the working memory key (e.g. 1 hour default).
- Validate that all entries are valid memory records and carry `id`.

Stage 3.5: Merge Session and Working Memory

Goal: Unify short-term memory abstractions into "WorkingMemory."

Instructions:
1. Standardize on the term working_memory
    -   "Session" is now just an ID value used to scope memory
•Rename all references to session memory or session-scoped memory to working memory
•In class names, route handlers, docs, comments
•E.g. SessionMemoryStore → WorkingMemoryStore

2. Ensure session scoping is preserved in storage
•All working memory should continue to be scoped per session:
•e.g. session:{id}:working_memory
•Validate session ID on all read/write access

3. Unify schema and access
•Replace any duplicate logic, structures, or APIs (e.g. separate SessionMemory and WorkingMemory models)
•Collapse into one structure: WorkingMemory
•Use one canonical POST /sessions/{id}/memory and GET /sessions/{id}/memory

4. Remove or migrate session-memory-only features
•If session memory had special logic (e.g. treating messages differently), migrate that logic into working memory
•Ensure messages, JSON data, and unpersisted semantic/episodic memories all coexist in working_memory

5. Audit all interfaces that reference session memory
•Tool APIs, prompt hydration, memory promotion, etc. should now reference working_memory exclusively
•Update any internal helper functions or routes to reflect the change

---

Stage 4: Add Background Promotion Task

Goal: Automatically move eligible working memory records to long-term storage.

Instructions:
- On working memory update, trigger an async background task.
- Task should:
  - Identify memory records with no `persisted_at`.
  - Use `id` to detect and replace duplicates in long-term memory.
  - Persist the record and stamp it with `persisted_at = now()`.
  - Update the working memory session store to reflect new timestamps.

Stage 5: Memory Search Interface (Complete)

Current Status: Completed

Progress:
- Implemented `search_memories` function (renamed from "unified" to just "memories")
- Added `POST /memory/search` endpoint that searches across all memory types
- Applied appropriate indexing and search logic:
  - Vector search for long-term memory (semantic search)
  - Simple text matching for working memory
  - Combined filtering and pagination across both types
- Included `memory_type` in search results along with all other memory fields
- Created comprehensive API tests for memory search endpoint
- Added unit test for `search_memories` function verifying working + long-term memory search
- Fixed linter errors with proper type handling
- Removed "unified" terminology in favor of cleaner "memory search"

Result: The system now provides a single search interface (`POST /memory/search`) that spans both working memory

---

Stage 6: Tool Interfaces for LLMs (Complete)

Current Status: Completed

Progress:
- Defined tool spec with required functions:
  - `store_memory(session_id, memory_type, content, tags, namespace, user_id, id)`
  - `store_json(session_id, data, namespace, user_id, id, tags)`
- Routed tool calls to session working memory via `PUT /sessions/{id}/memory`
- Auto-generated `id` using ULID when not supplied by client
- Marked all tool-created records as pending promotion (`persisted_at = null`)
- Added comprehensive MCP tool documentation with usage patterns
- Implemented proper namespace injection for both URL-based and default namespaces
- Created comprehensive tests for both tool functions including ID auto-generation

- Verified integration with existing working memory and background promotion systems

Result: LLMs can now explicitly store structured memory during conversation through tool calls. The `store_memory

---

Stage 7: Automatic Memory Extraction from Messages (Complete)

Current Status: Completed

Progress:
- Extended background promotion task to include message record extraction
- Implemented `extract_memories_from_messages` function for working memory context
- Added LLM-based extraction using `WORKING_MEMORY_EXTRACTION_PROMPT`
- Tagged extracted records with `extracted_from` field containing source message IDs
- Generated server-side IDs for all extracted memories using ULID
- Added `extracted_from` field to MemoryRecord model and Redis schema
- Updated indexing and search logic to handle extracted_from field
- Integrated extraction into promotion workflow with proper error handling
- Added extracted memories to working memory for future promotion cycles
- Verified all tests pass with new extraction functionality

Result: The system now automatically extracts semantic and episodic memories from message records during the prom

---

Final Integration: Sync and Conflict Safety (Complete)

Current Status: Completed

Progress:
- Verified client state resubmission safety via `PUT /sessions/{id}/memory` endpoint
- Confirmed pending record handling: records with `id` but no `persisted_at` treated as pending
- Validated id-based overwrite logic in `deduplicate_by_id` function
- Ensured working memory always updated with latest `persisted_at` timestamps
- Created comprehensive test for sync and conflict safety scenarios
- Verified client can safely resubmit stale memory state with new records
- Confirmed long-term memory convergence over time through promotion cycles
- Validated that server handles partial client state gracefully
- Ensured proper timestamp management across promotion cycles

Result: The system now provides robust sync and conflict safety. Clients can safely resubmit partial or stale mem

---

Log of work

Stage 1: Normalize Memory Types (Complete)

Current Status: Completed

Progress:
- Analyzed current codebase structure
- Found that `memory_type` field already exists in `LongTermMemory` model with values: `"episodic"`, `"semantic"`, `
- Added `"json"` type support to the Literal type definition
- Verified field validation exists in APIs via MemoryType filter class
- Confirmed indexing and query logic includes this field in Redis search schema
- All memory search, indexing, and storage operations properly handle memory_type

Result: The `memory_type` field is now normalized with all required values: `"message"`, `"semantic"`, `"episodic

Stage 1.5: Rename `LongTermMemory*` Classes to `Memory*` (Complete)

Current Status: Completed

Progress:
- Renamed `LongTermMemory` → `MemoryRecord`
- Renamed `LongTermMemoryResult` → `MemoryRecordResult`
- Renamed `LongTermMemoryResults` → `MemoryRecordResults`

- Renamed `LongTermMemoryResultsResponse` → `MemoryRecordResultsResponse`
- Renamed `CreateLongTermMemoryRequest` → `CreateMemoryRecordRequest`
- Updated all references in code, route handlers, type hints, and OpenAPI schema
- Updated imports across all modules: models, long_term_memory, api, client, mcp, messages, extraction
- Updated all test files and their imports
- Verified all files compile without syntax errors

Result: All `LongTermMemory*` classes have been successfully renamed to `Memory*` classes, removing location-base

Stage 2: Add `id` and `persisted_at` (Complete)

Current Status: Completed

Progress:
- Added `id: str | None` and `persisted_at: datetime | None` to MemoryRecord model
- Updated Redis schema to include id (tag field) and persisted_at (numeric field)
- Updated indexing logic to store these fields with proper timestamp conversion
- Updated search logic to return new fields with datetime conversion
- Added validation to API to enforce id requirement for client-sent memory
- Ensured persisted_at is server-assigned and read-only for clients
- Implemented `deduplicate_by_id` function for id-based deduplication
- Integrated id deduplication as first step in indexing process
- Added comprehensive tests for id validation and deduplication
- Verified all existing tests pass with new functionality

Result: Id and persisted_at fields are now fully implemented with proper validation, deduplication logic, and saf

Stage 3: Implement Working Memory (Complete)

Current Status: Completed

Progress:
- Defined Redis keyspace like `working_memory:{namespace}:{session_id}`
- Implemented `GET /sessions/{id}/working-memory` — returns current working memory
- Implemented `POST /sessions/{id}/working-memory` — replaces full working memory state
- Set TTL on working memory key (1 hour default, configurable)
- Validated that all entries are valid memory records and carry `id`
- Created WorkingMemory model containing list of MemoryRecord objects
- Implemented working memory storage/retrieval functions with JSON serialization
- Added comprehensive tests for working memory functionality and API endpoints
- Verified all tests pass with new functionality

Result: Working memory is now fully implemented as a TTL-based, session-scoped memory area for ephemeral agent co

Stage 3.5: Merge Session and Working Memory (Complete)

Current Status: Completed

Progress:
- Standardized on "working_memory" terminology throughout codebase
- Extended WorkingMemory model to support both messages and structured memory records
- Removed SessionMemory, SessionMemoryRequest, SessionMemoryResponse models
- Unified API endpoints to single /sessions/{id}/memory (GET/PUT/DELETE)
- Removed deprecated /working-memory endpoints
- Preserved session scoping in Redis storage (working_memory:{namespace}:{session_id})
- Removed duplicate logic and APIs between session and working memory
- Updated all interfaces to reference working_memory exclusively
- Migrated all session-memory-only features into working memory
- Updated all test files to use unified WorkingMemory models
- Verified all 80 tests pass with unified architecture

Result: Successfully unified short-term memory abstractions into "WorkingMemory" terminology, eliminating duplica

Additional Improvements (Complete)

Current Status: Completed

Progress:
- **Renamed `client_id` to `id`**: Updated all references throughout the codebase from `client_id` to `id` for clean

- **Implemented immediate summarization**: Modified `PUT /sessions/{id}/memory` to handle summarization inline inste
- **Updated client API**: Modified `MemoryAPIClient.put_session_memory()` to return `WorkingMemoryResponse` instead
- **Fixed test mocks**: Updated all test files to use the new field names and response types.
- **Verified all tests pass**: All 80 tests pass with the updated implementation.

Result: The API now has cleaner field naming (`id` instead of `client_id`) and provides immediate feedback to cli

Stage 4: Add Background Promotion Task (Complete)

Current Status: Completed

Progress:
- Created `promote_working_memory_to_long_term` function that automatically promotes eligible memories
- Implemented identification of memory records with no `persisted_at` in working memory
- Added id-based deduplication and overwrite detection during promotion
- Implemented proper `persisted_at` timestamp assignment using UTC datetime
- Added working memory update logic to reflect new timestamps after promotion
- Integrated promotion task into `put_session_memory` API endpoint as background task
- Added promotion function to Docket task collection for background processing
- Created comprehensive tests for promotion functionality and API integration
- Verified proper triggering of promotion task only when structured memories are present
- Verified all 82 tests pass with new functionality

Result: Background promotion task is now fully implemented. When working memory is updated via the API, unpersist

Stage 5: Memory Search Interface (Complete)

Current Status: Completed

Progress:
- Implemented `search_memories` function (renamed from "unified" to just "memories")
- Added `POST /memory/search` endpoint that searches across all memory types
- Applied appropriate indexing and search logic:
  - Vector search for long-term memory (semantic search)
  - Simple text matching for working memory
  - Combined filtering and pagination across both types
- Included `memory_type` in search results along with all other memory fields
- Created comprehensive API tests for memory search endpoint
- Added unit test for `search_memories` function verifying working + long-term memory search
- Fixed linter errors with proper type handling
- Removed "unified" terminology in favor of cleaner "memory search"

Result: The system now provides a single search interface (`POST /memory/search`) that spans both working memory

Stage 6: Tool Interfaces for LLMs (Complete)

Current Status: Completed

Progress:
- Defined tool spec with required functions:
  - `store_memory(session_id, memory_type, content, tags, namespace, user_id, id)`
  - `store_json(session_id, data, namespace, user_id, id, tags)`
- Routed tool calls to session working memory via `PUT /sessions/{id}/memory`
- Auto-generated `id` using ULID when not supplied by client
- Marked all tool-created records as pending promotion (`persisted_at = null`)
- Added comprehensive MCP tool documentation with usage patterns
- Implemented proper namespace injection for both URL-based and default namespaces
- Created comprehensive tests for both tool functions including ID auto-generation
- Verified integration with existing working memory and background promotion systems

Result: LLMs can now explicitly store structured memory during conversation through tool calls. The `store_memory

Stage 7: Automatic Memory Extraction from Messages (Complete)

Current Status: Completed

Progress:
- Extended background promotion task to include message record extraction
- Implemented `extract_memories_from_messages` function for working memory context

- Added LLM-based extraction using `WORKING_MEMORY_EXTRACTION_PROMPT`
- Tagged extracted records with `extracted_from` field containing source message IDs
- Generated server-side IDs for all extracted memories using nanoid
- Added `extracted_from` field to MemoryRecord model and Redis schema
- Updated indexing and search logic to handle extracted_from field
- Integrated extraction into promotion workflow with proper error handling
- Added extracted memories to working memory for future promotion cycles
- Verified all tests pass with new extraction functionality

Result: The system now automatically extracts semantic and episodic memories from message records during the prom

Final Integration: Sync and Conflict Safety (Complete)

Current Status: Completed

Progress:
- Verified client state resubmission safety via `PUT /sessions/{id}/memory` endpoint
- Confirmed pending record handling: records with `id` but no `persisted_at` treated as pending
- Validated id-based overwrite logic in `deduplicate_by_id` function
- Ensured working memory always updated with latest `persisted_at` timestamps
- Created comprehensive test for sync and conflict safety scenarios
- Verified client can safely resubmit stale memory state with new records
- Confirmed long-term memory convergence over time through promotion cycles
- Validated that server handles partial client state gracefully
- Ensured proper timestamp management across promotion cycles

Result: The system now provides robust sync and conflict safety. Clients can safely resubmit partial or stale mem

== agent_memory_server/__init__.py ==

"""Re

__version__ = "0.1.0"

== agent_memory_server/api.py ==

```
impor

from fastapi import APIRouter, Depends, HTTPException
from mcp.server.fastmcp.prompts import base
from mcp.types import TextContent

from agent_memory_server import long_term_memory, messages, working_memory
from agent_memory_server.config import settings
from agent_memory_server.dependencies import get_background_tasks
from agent_memory_server.llms import get_model_client, get_model_config
from agent_memory_server.logging import get_logger
from agent_memory_server.models import (
    AckResponse,
    CreateMemoryRecordRequest,
    GetSessionsQuery,
    MemoryPromptRequest,
    MemoryPromptResponse,
    MemoryRecordResultsResponse,
    MemoryTypeEnum,
    ModelNameLiteral,
    SearchRequest,
    SessionListResponse,
    SystemMessage,
    WorkingMemory,
    WorkingMemoryResponse,
)
from agent_memory_server.summarization import _incremental_summary
from agent_memory_server.utils.redis import get_redis_conn


logger = get_logger(__name__)

router = APIRouter()


def _get_effective_window_size(
    window_size: int,
    context_window_max: int | None,
    model_name: ModelNameLiteral | None,
) -> int:
    # If context_window_max is explicitly provided, use that
    if context_window_max is not None:
        effective_window_size = min(window_size, context_window_max)
    # If model_name is provided, get its max_tokens from our config
    elif model_name is not None:
        model_config = get_model_config(model_name)
        effective_window_size = min(window_size, model_config.max_tokens)
    # Otherwise use the default window_size
    else:
        effective_window_size = window_size
    return effective_window_size


@router.get("/sessions/", response_model=SessionListResponse)
async def list_sessions(
    options: GetSessionsQuery = Depends(),
):
    """
    Get a list of session IDs, with optional pagination.

    Args:
        options: Query parameters (page, size, namespace)

    Returns:
        List of session IDs
    """
    redis = await get_redis_conn()

    total, session_ids = await messages.list_sessions(
```

```
        redis=redis,
        limit=options.limit,
        offset=options.offset,
        namespace=options.namespace,
    )

    return SessionListResponse(
        sessions=session_ids,
        total=total,
    )


@router.get("/sessions/{session_id}/memory", response_model=WorkingMemoryResponse)
async def get_session_memory(
    session_id: str,
    namespace: str | None = None,
    window_size: int = settings.window_size,
    model_name: ModelNameLiteral | None = None,
    context_window_max: int | None = None,
):
    """
    Get working memory for a session.

    This includes stored conversation messages, context, and structured memory records.

    Args:
        session_id: The session ID
        namespace: The namespace to use for the session
        window_size: The number of messages to include in the response
        model_name: The client's LLM model name (will determine context window size if provided)
        context_window_max: Direct specification of the context window max tokens (overrides model_name)

    Returns:
        Working memory containing messages, context, and structured memory records
    """
    redis = await get_redis_conn()
    effective_window_size = _get_effective_window_size(
        window_size=window_size,
        context_window_max=context_window_max,
        model_name=model_name,
    )

    # Get unified working memory
    working_mem = await working_memory.get_working_memory(
        session_id=session_id,
        namespace=namespace,
        redis_client=redis,
    )

    if not working_mem:
        # Return empty working memory if none exists
        working_mem = WorkingMemory(
            messages=[],
            memories=[],
            session_id=session_id,
            namespace=namespace,
        )

    # Apply window size to messages if needed
    if len(working_mem.messages) > effective_window_size:
        working_mem.messages = working_mem.messages[-effective_window_size:]

    return working_mem


async def _summarize_working_memory(
    memory: WorkingMemory,
    window_size: int,
    model: str = settings.generation_model,
```

```
) -> WorkingMemory:
    """
    Summarize working memory when it exceeds the window size.

    Args:
        memory: The working memory to potentially summarize
        window_size: Maximum number of messages to keep
        model: The model to use for summarization

    Returns:
        Updated working memory with summary and trimmed messages
    """
    if len(memory.messages) <= window_size:
        return memory

    # Get model client for summarization
    client = await get_model_client(model)
    model_config = get_model_config(model)
    max_tokens = model_config.max_tokens

    # Token allocation (same logic as original summarize_session)
    if max_tokens < 10000:
        summary_max_tokens = max(512, max_tokens // 8)  # 12.5%
    elif max_tokens < 50000:
        summary_max_tokens = max(1024, max_tokens // 10)  # 10%
    else:
        summary_max_tokens = max(2048, max_tokens // 20)  # 5%

    buffer_tokens = min(max(230, max_tokens // 100), 1000)
    max_message_tokens = max_tokens - summary_max_tokens - buffer_tokens

    encoding = tiktoken.get_encoding("cl100k_base")
    total_tokens = 0
    messages_to_summarize = []

    # Calculate how many messages from the beginning we should summarize
    # Keep the most recent messages within window_size
    messages_to_check = (
        memory.messages[:-window_size] if len(memory.messages) > window_size else []
    )

    for msg in messages_to_check:
        msg_str = f"{msg.role}: {msg.content}"
        msg_tokens = len(encoding.encode(msg_str))

        # Handle oversized messages
        if msg_tokens > max_message_tokens:
            msg_str = msg_str[: max_message_tokens // 2]
            msg_tokens = len(encoding.encode(msg_str))

        if total_tokens + msg_tokens <= max_message_tokens:
            total_tokens += msg_tokens
            messages_to_summarize.append(msg_str)
        else:
            break

    if not messages_to_summarize:
        # No messages to summarize, just return original memory
        return memory

    # Generate summary
    summary, summary_tokens_used = await _incremental_summary(
        model,
        client,
        memory.context,  # Use existing context as base
        messages_to_summarize,
    )

    # Update working memory with new summary and trimmed messages
```

```
    # Keep only the most recent messages within window_size
    updated_memory = memory.model_copy(deep=True)
    updated_memory.context = summary
    updated_memory.messages = memory.messages[
        -window_size:
    ]  # Keep most recent messages
    updated_memory.tokens = memory.tokens + summary_tokens_used

    return updated_memory


@router.put("/sessions/{session_id}/memory", response_model=WorkingMemoryResponse)
async def put_session_memory(
    session_id: str,
    memory: WorkingMemory,
    background_tasks=Depends(get_background_tasks),
):
    """
    Set working memory for a session. Replaces existing working memory.

    If the message count exceeds the window size, messages will be summarized
    immediately and the updated memory state returned to the client.

    Args:
        session_id: The session ID
        memory: Working memory to save
        background_tasks: DocketBackgroundTasks instance (injected automatically)

    Returns:
        Updated working memory (potentially with summary if messages were condensed)
    """
    redis = await get_redis_conn()

    # Ensure session_id matches
    memory.session_id = session_id

    # Validate that all structured memories have id (if any)
    for mem in memory.memories:
        if not mem.id:
            raise HTTPException(
                status_code=400,
                detail="All memory records in working memory must have an id",
            )

    # Handle summarization if needed (before storing)
    updated_memory = memory
    if memory.messages and len(memory.messages) > settings.window_size:
        updated_memory = await _summarize_working_memory(memory, settings.window_size)

    await working_memory.set_working_memory(
        working_memory=updated_memory,
        redis_client=redis,
    )

    # Background tasks for long-term memory promotion and indexing (if enabled)
    if settings.long_term_memory:
        # Promote structured memories from working memory to long-term storage
        if updated_memory.memories:
            await background_tasks.add_task(
                long_term_memory.promote_working_memory_to_long_term,
                session_id,
                updated_memory.namespace,
            )

        # Index message-based memories (existing logic)
        if updated_memory.messages:
            from agent_memory_server.models import MemoryRecord

            memories = [
```

```
                MemoryRecord(
                    session_id=session_id,
                    text=f"{msg.role}: {msg.content}",
                    namespace=updated_memory.namespace,
                    memory_type=MemoryTypeEnum.MESSAGE,
                )
                for msg in updated_memory.messages
            ]

            await background_tasks.add_task(
                long_term_memory.index_long_term_memories,
                memories,
            )

    return updated_memory


@router.delete("/sessions/{session_id}/memory", response_model=AckResponse)
async def delete_session_memory(
    session_id: str,
    namespace: str | None = None,
):
    """
    Delete working memory for a session.

    This deletes all stored memory (messages, context, structured memories) for a session.

    Args:
        session_id: The session ID
        namespace: Optional namespace for the session

    Returns:
        Acknowledgement response
    """
    redis = await get_redis_conn()

    # Delete unified working memory
    await working_memory.delete_working_memory(
        session_id=session_id,
        namespace=namespace,
        redis_client=redis,
    )

    return AckResponse(status="ok")


@router.post("/long-term-memory", response_model=AckResponse)
async def create_long_term_memory(
    payload: CreateMemoryRecordRequest,
    background_tasks=Depends(get_background_tasks),
):
    """
    Create a long-term memory

    Args:
        payload: Long-term memory payload
        background_tasks: DocketBackgroundTasks instance (injected automatically)

    Returns:
        Acknowledgement response
    """
    if not settings.long_term_memory:
        raise HTTPException(status_code=400, detail="Long-term memory is disabled")

    # Validate and process memories according to Stage 2 requirements
    for memory in payload.memories:
        # Enforce that id is required on memory sent from clients
        if not memory.id:
            raise HTTPException(
```

```
                status_code=400, detail="id is required for all memory records"
            )

        # Ensure persisted_at is server-assigned and read-only for clients
        # Clear any client-provided persisted_at value
        memory.persisted_at = None

    await background_tasks.add_task(
        long_term_memory.index_long_term_memories,
        memories=payload.memories,
    )
    return AckResponse(status="ok")


@router.post("/long-term-memory/search", response_model=MemoryRecordResultsResponse)
async def search_long_term_memory(payload: SearchRequest):
    """
    Run a semantic search on long-term memory with filtering options.

    Args:
        payload: Search payload with filter objects for precise queries

    Returns:
        List of search results
    """
    if not settings.long_term_memory:
        raise HTTPException(status_code=400, detail="Long-term memory is disabled")

    redis = await get_redis_conn()

    # Extract filter objects from the payload
    filters = payload.get_filters()

    kwargs = {
        "redis": redis,
        "distance_threshold": payload.distance_threshold,
        "limit": payload.limit,
        "offset": payload.offset,
        **filters,
    }

    if payload.text:
        kwargs["text"] = payload.text

    # Pass text, redis, and filter objects to the search function
    return await long_term_memory.search_long_term_memories(**kwargs)


@router.post("/memory/search", response_model=MemoryRecordResultsResponse)
async def search_memory(payload: SearchRequest):
    """
    Run a search across all memory types (working memory and long-term memory).

    This endpoint searches both working memory (ephemeral, session-scoped) and
    long-term memory (persistent, indexed) to provide comprehensive results.

    For working memory:
    - Uses simple text matching
    - Searches across all sessions (unless session_id filter is provided)
    - Returns memories that haven't been promoted to long-term storage

    For long-term memory:
    - Uses semantic vector search
    - Includes promoted memories from working memory
    - Supports advanced filtering by topics, entities, etc.

    Args:
        payload: Search payload with filter objects for precise queries
```

```
    Returns:
        Search results from both memory types, sorted by relevance
    """
    redis = await get_redis_conn()

    # Extract filter objects from the payload
    filters = payload.get_filters()

    kwargs = {
        "redis": redis,
        "distance_threshold": payload.distance_threshold,
        "limit": payload.limit,
        "offset": payload.offset,
        **filters,
    }

    if payload.text:
        kwargs["text"] = payload.text

    # Use the search function
    return await long_term_memory.search_memories(**kwargs)


@router.post("/memory-prompt", response_model=MemoryPromptResponse)
async def memory_prompt(params: MemoryPromptRequest) -> MemoryPromptResponse:
    """
    Hydrate a user query with memory context and return a prompt
    ready to send to an LLM.

    `query` is the input text that the caller of this API wants to use to find
    relevant context. If `session_id` is provided and matches an existing
    session, the resulting prompt will include those messages as the immediate
    history of messages leading to a message containing `query`.

    If `long_term_search_payload` is provided, the resulting prompt will include
    relevant long-term memories found via semantic search with the options
    provided in the payload.

    Args:
        params: MemoryPromptRequest

    Returns:
        List of messages to send to an LLM, hydrated with relevant memory context
    """
    if not params.session and not params.long_term_search:
        raise HTTPException(
            status_code=400,
            detail="Either session or long_term_search must be provided",
        )

    redis = await get_redis_conn()
    _messages = []

    if params.session:
        effective_window_size = _get_effective_window_size(
            window_size=params.session.window_size,
            context_window_max=params.session.context_window_max,
            model_name=params.session.model_name,
        )
        working_mem = await working_memory.get_working_memory(
            session_id=params.session.session_id,
            namespace=params.session.namespace,
            redis_client=redis,
        )

        if working_mem:
            if working_mem.context:
                # TODO: Weird to use MCP types here?
                _messages.append(
```

```
                SystemMessage(
                    content=TextContent(
                        type="text",
                        text=f"## A summary of the conversation so far:\n{working_mem.context}",
                    ),
                )
            )
        # Apply window size and ignore past system messages as the latest context may have changed
        recent_messages = (
            working_mem.messages[-effective_window_size:]
            if len(working_mem.messages) > effective_window_size
            else working_mem.messages
        )
        for msg in recent_messages:
            if msg.role == "user":
                msg_class = base.UserMessage
            else:
                msg_class = base.AssistantMessage
            _messages.append(
                msg_class(
                    content=TextContent(type="text", text=msg.content),
                )
            )

if params.long_term_search:
    # TODO: Exclude session messages if we already included them from session memory
    long_term_memories = await search_long_term_memory(
        params.long_term_search,
    )

    if long_term_memories.total > 0:
        long_term_memories_text = "\n".join(
            [f"- {m.text}" for m in long_term_memories.memories]
        )
        _messages.append(
            SystemMessage(
                content=TextContent(
                    type="text",
                    text=f"## Long term memories related to the user's query\n {long_term_memories_text}",
                ),
            )
        )

_messages.append(
    base.UserMessage(
        content=TextContent(type="text", text=params.query),
    )
)

return MemoryPromptResponse(messages=_messages)
```

== agent_memory_server/cli.py ==

```
#!/us
"""
Command-line interface for agent-memory-server.
"""

import datetime
import importlib
import logging
import sys

import click
import uvicorn

from agent_memory_server.config import settings
from agent_memory_server.logging import configure_logging, get_logger
from agent_memory_server.migrations import (
    migrate_add_discrete_memory_extracted_2,
    migrate_add_memory_hashes_1,
    migrate_add_memory_type_3,
)
from agent_memory_server.utils.redis import ensure_search_index_exists, get_redis_conn


configure_logging()
logger = get_logger(__name__)

VERSION = "0.2.0"


@click.group()
def cli():
    """Command-line interface for agent-memory-server."""
    pass


@cli.command()
def version():
    """Show the version of agent-memory-server."""
    click.echo(f"agent-memory-server version {VERSION}")


@cli.command()
def rebuild_index():
    """Rebuild the search index."""
    import asyncio

    async def setup_and_run():
        redis = await get_redis_conn()
        await ensure_search_index_exists(redis, overwrite=True)

    asyncio.run(setup_and_run())


@cli.command()
def migrate_memories():
    """Migrate memories from the old format to the new format."""
    import asyncio

    click.echo("Starting memory migrations...")

    async def run_migrations():
        redis = await get_redis_conn()
        migrations = [
            migrate_add_memory_hashes_1,
            migrate_add_discrete_memory_extracted_2,
            migrate_add_memory_type_3,
        ]
        for migration in migrations:
```

```
            await migration(redis=redis)

    asyncio.run(run_migrations())

    click.echo("Memory migrations completed successfully.")


@cli.command()
@click.option("--port", default=settings.port, help="Port to run the server on")
@click.option("--host", default="0.0.0.0", help="Host to run the server on")
@click.option("--reload", is_flag=True, help="Enable auto-reload")
def api(port: int, host: str, reload: bool):
    """Run the REST API server."""
    from agent_memory_server.main import on_start_logger

    on_start_logger(port)
    uvicorn.run(
        "agent_memory_server.main:app",
        host=host,
        port=port,
        reload=reload,
    )


@cli.command()
@click.option("--port", default=settings.mcp_port, help="Port to run the MCP server on")
@click.option(
    "--mode",
    default="stdio",
    help="Run the MCP server in SSE or stdio mode",
    type=click.Choice(["stdio", "sse"]),
)
def mcp(port: int, mode: str):
    """Run the MCP server."""
    import asyncio

    # Update the port in settings FIRST
    settings.mcp_port = port

    # Import mcp_app AFTER settings have been updated
    from agent_memory_server.mcp import mcp_app

    async def setup_and_run():
        # Redis setup is handled by the MCP app before it starts

        # Run the MCP server
        if mode == "sse":
            logger.info(f"Starting MCP server on port {port}\n")
            await mcp_app.run_sse_async()
        elif mode == "stdio":
            # Try to force all logging to stderr because stdio-mode MCP servers
            # use standard output for the protocol.
            logging.basicConfig(
                level=settings.log_level,
                stream=sys.stderr,
                force=True,  # remove any existing handlers
                format="%(asctime)s %(name)s %(levelname)s %(message)s",
            )
            await mcp_app.run_stdio_async()
        else:
            raise ValueError(f"Invalid mode: {mode}")

    # Update the port in settings
    settings.mcp_port = port

    asyncio.run(setup_and_run())


@cli.command()
```

```
@click.argument("task_path")
@click.option(
    "--args",
    "-a",
    multiple=True,
    help="Arguments to pass to the task in the format key=value",
)
def schedule_task(task_path: str, args: list[str]):
    """
    Schedule a background task by path.

    TASK_PATH is the import path to the task function, e.g.,
    "agent_memory_server.long_term_memory.compact_long_term_memories"
    """
    import asyncio

    from docket import Docket

    # Parse the arguments
    task_args = {}
    for arg in args:
        try:
            key, value = arg.split("=", 1)
            # Try to convert to appropriate type
            if value.lower() == "true":
                task_args[key] = True
            elif value.lower() == "false":
                task_args[key] = False
            elif value.isdigit():
                task_args[key] = int(value)
            elif value.replace(".", "", 1).isdigit() and value.count(".") <= 1:
                task_args[key] = float(value)
            else:
                task_args[key] = value
        except ValueError:
            click.echo(f"Invalid argument format: {arg}. Use key=value format.")
            sys.exit(1)

    async def setup_and_run_task():
        redis = await get_redis_conn()
        await ensure_search_index_exists(redis)

        # Import the task function
        module_path, function_name = task_path.rsplit(".", 1)
        try:
            module = importlib.import_module(module_path)
            task_func = getattr(module, function_name)
        except (ImportError, AttributeError) as e:
            click.echo(f"Error importing task: {e}")
            sys.exit(1)

        # Initialize Docket client
        async with Docket(
            name=settings.docket_name,
            url=settings.redis_url,
        ) as docket:
            click.echo(f"Scheduling task {task_path} with arguments: {task_args}")
            await docket.add(task_func)(**task_args)
            click.echo("Task scheduled successfully")

    asyncio.run(setup_and_run_task())


@cli.command()
@click.option(
    "--concurrency", default=10, help="Number of tasks to process concurrently"
)
@click.option(
    "--redelivery-timeout",
```

```
    default=30,
    help="Seconds to wait before redelivering a task to another worker",
)
def task_worker(concurrency: int, redelivery_timeout: int):
    """
    Start a Docket worker using the Docket name from settings.

    This command starts a worker that processes background tasks registered
    with Docket. The worker uses the Docket name from settings.
    """
    import asyncio

    from docket import Worker

    if not settings.use_docket:
        click.echo("Docket is disabled in settings. Cannot run worker.")
        sys.exit(1)

    asyncio.run(
        Worker.run(
            docket_name=settings.docket_name,
            url=settings.redis_url,
            concurrency=concurrency,
            redelivery_timeout=datetime.timedelta(seconds=redelivery_timeout),
            tasks=["agent_memory_server.docket_tasks:task_collection"],
        )
    )


if __name__ == "__main__":
    cli()
```

== agent_memory_server/config.py ==

```
impor
from typing import Literal

import yaml
from dotenv import load_dotenv
from pydantic_settings import BaseSettings


load_dotenv()


def load_yaml_settings():
    config_path = os.getenv("APP_CONFIG_FILE", "config.yaml")
    if os.path.exists(config_path):
        with open(config_path) as f:
            return yaml.safe_load(f) or {}
    return {}


class Settings(BaseSettings):
    redis_url: str = "redis://localhost:6379"
    long_term_memory: bool = True
    window_size: int = 20
    openai_api_key: str | None = None
    anthropic_api_key: str | None = None
    generation_model: str = "gpt-4o-mini"
    embedding_model: str = "text-embedding-3-small"
    port: int = 8000
    mcp_port: int = 9000

    # The server indexes messages in long-term memory by default. If this
    # setting is enabled, we also extract discrete memories from message text
    # and save them as separate long-term memory records.
    enable_discrete_memory_extraction: bool = True

    # Topic modeling
    topic_model_source: Literal["BERTopic", "LLM"] = "LLM"
    topic_model: str = (
        "MaartenGr/BERTopic_Wikipedia"  # Use an LLM model name here if using LLM
    )
    enable_topic_extraction: bool = True
    top_k_topics: int = 3

    # Used for extracting entities from text
    ner_model: str = "dbmdz/bert-large-cased-finetuned-conll03-english"
    enable_ner: bool = True

    # RedisVL Settings
    redisvl_distance_metric: str = "COSINE"
    redisvl_vector_dimensions: str = "1536"
    redisvl_index_name: str = "memory"
    redisvl_index_prefix: str = "memory"

    # Docket settings
    docket_name: str = "memory-server"
    use_docket: bool = True

    # Other Application settings
    log_level: Literal["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"] = "INFO"

    class Config:
        env_file = ".env"
        env_file_encoding = "utf-8"


# Load YAML config first, then let env vars override
yaml_settings = load_yaml_settings()
settings = Settings(**yaml_settings)
```

== agent_memory_server/dependencies.py ==

```
from typing import Any

from fastapi import BackgroundTasks

from agent_memory_server.config import settings
from agent_memory_server.logging import get_logger


logger = get_logger(__name__)


class DocketBackgroundTasks(BackgroundTasks):
    """A BackgroundTasks implementation that uses Docket."""

    async def add_task(
        self, func: Callable[..., Any], *args: Any, **kwargs: Any
    ) -> None:
        """Run tasks either directly or through Docket"""
        from docket import Docket

        from agent_memory_server.utils.redis import get_redis_conn

        logger.info("Adding task to background tasks...")

        if settings.use_docket:
            logger.info("Scheduling task through Docket")
            # Get the Redis connection that's already configured (will use testcontainer in tests)
            redis_conn = await get_redis_conn()
            # Use the connection's URL instead of settings.redis_url directly
            redis_url = redis_conn.connection_pool.connection_kwargs.get(
                "url", settings.redis_url
            )
            logger.info("redis_url: %s", redis_url)
            logger.info("docket_name: %s", settings.docket_name)
            async with Docket(
                name=settings.docket_name,
                url=redis_url,
            ) as docket:
                # Schedule task through Docket
                await docket.add(func)(*args, **kwargs)
        else:
            logger.info("Running task directly")
            await func(*args, **kwargs)


def get_background_tasks() -> DocketBackgroundTasks:
    """
    Dependency function that returns a DocketBackgroundTasks instance.

    This is used by API endpoints to inject a consistent background tasks object.
    """
    logger.info("Getting background tasks class")
    return DocketBackgroundTasks()
```

== agent_memory_server/dev_server.py ==

```
#!/usr

"""Run the Redis Agent Memory Server."""

import os

import uvicorn

from agent_memory_server.main import on_start_logger


if __name__ == "__main__":
    port = int(os.environ.get("PORT", "8000"))
    on_start_logger(port)
    uvicorn.run("agent_memory_server.main:app", host="0.0.0.0", port=port, reload=True)
```

```
== agent_memory_server/docket_tasks.py ==
                                                                                """
Background task management using Docket.
"""

import logging

from docket import Docket

from agent_memory_server.config import settings
from agent_memory_server.extraction import extract_discrete_memories
from agent_memory_server.long_term_memory import (
    compact_long_term_memories,
    extract_memory_structure,
    index_long_term_memories,
    promote_working_memory_to_long_term,
)
from agent_memory_server.summarization import summarize_session


logger = logging.getLogger(__name__)


# Register functions in the task collection for the CLI worker
task_collection = [
    extract_memory_structure,
    summarize_session,
    index_long_term_memories,
    compact_long_term_memories,
    extract_discrete_memories,
    promote_working_memory_to_long_term,
]


async def register_tasks() -> None:
    """Register all task functions with Docket."""
    if not settings.use_docket:
        logger.info("Docket is disabled, skipping task registration")
        return

    # Initialize Docket client
    async with Docket(
        name=settings.docket_name,
        url=settings.redis_url,
    ) as docket:
        # Register all tasks
        for task in task_collection:
            docket.register(task)

        logger.info(f"Registered {len(task_collection)} background tasks with Docket")
```

== agent_memory_server/extraction.py ==

```
import os
from typing import Any

from bertopic import BERTopic
from redis.asyncio.client import Redis
from redisvl.query.filter import Tag
from redisvl.query.query import FilterQuery
from tenacity.asyncio import AsyncRetrying
from tenacity.stop import stop_after_attempt
from transformers import AutoModelForTokenClassification, AutoTokenizer, pipeline
from ulid import ULID

from agent_memory_server.config import settings
from agent_memory_server.llms import (
    AnthropicClientWrapper,
    OpenAIClientWrapper,
    get_model_client,
)
from agent_memory_server.logging import get_logger
from agent_memory_server.models import MemoryRecord
from agent_memory_server.utils.redis import get_redis_conn, get_search_index


logger = get_logger(__name__)

# Set tokenizer parallelism environment variable
os.environ["TOKENIZERS_PARALLELISM"] = "false"

# Global model instances
_topic_model: BERTopic | None = None
_ner_model: Any | None = None
_ner_tokenizer: Any | None = None


def get_topic_model() -> BERTopic:
    """
    Get or initialize the BERTopic model.

    Returns:
        The BERTopic model instance
    """
    global _topic_model
    if _topic_model is None:
        # TODO: Expose this as a config option
        _topic_model = BERTopic.load(
            settings.topic_model, embedding_model="all-MiniLM-L6-v2"
        )
    return _topic_model  # type: ignore


def get_ner_model() -> Any:
    """
    Get or initialize the NER model and tokenizer.

    Returns:
        The NER pipeline instance
    """
    global _ner_model, _ner_tokenizer
    if _ner_model is None:
        _ner_tokenizer = AutoTokenizer.from_pretrained(settings.ner_model)
        _ner_model = AutoModelForTokenClassification.from_pretrained(settings.ner_model)
    return pipeline("ner", model=_ner_model, tokenizer=_ner_tokenizer)


def extract_entities(text: str) -> list[str]:
    """
    Extract named entities from text using the NER model.
```

```
    TODO: Cache this output.

    Args:
        text: The text to extract entities from

    Returns:
        List of unique entity names
    """
    try:
        ner = get_ner_model()
        results = ner(text)

        # Group tokens by entity
        current_entity = []
        entities = []

        for result in results:
            if result["word"].startswith("##"):
                # This is a continuation of the previous entity
                current_entity.append(result["word"][2:])
            else:
                # This is a new entity
                if current_entity:
                    entities.append("".join(current_entity))
                current_entity = [result["word"]]

        # Add the last entity if exists
        if current_entity:
            entities.append("".join(current_entity))

        return list(set(entities))  # Remove duplicates

    except Exception as e:
        logger.error(f"Error extracting entities: {e}")
        return []


async def extract_topics_llm(
    text: str,
    num_topics: int | None = None,
    client: OpenAIClientWrapper | AnthropicClientWrapper | None = None,
) -> list[str]:
    """
    Extract topics from text using the LLM model.
    """
    _client = client or await get_model_client(settings.generation_model)
    _num_topics = num_topics if num_topics is not None else settings.top_k_topics

    prompt = f"""
    Extract the topic {_num_topics} topics from the following text:
    {text}

    Return a list of topics in JSON format, for example:
    {{
        "topics": ["topic1", "topic2", "topic3"]
    }}
    """
    topics = []

    async for attempt in AsyncRetrying(stop=stop_after_attempt(3)):
        with attempt:
            response = await _client.create_chat_completion(
                model=settings.generation_model,
                prompt=prompt,
                response_format={"type": "json_object"},
            )
            try:
                topics = json.loads(response.choices[0].message.content)["topics"]
```

```
            except (json.JSONDecodeError, KeyError):
                logger.error(
                    f"Error decoding JSON: {response.choices[0].message.content}"
                )
                topics = []
            if topics:
                topics = topics[:_num_topics]

    return topics


def extract_topics_bertopic(text: str, num_topics: int | None = None) -> list[str]:
    """
    Extract topics from text using the BERTopic model.

    TODO: Cache this output.

    Args:
        text: The text to extract topics from

    Returns:
        List of topic labels
    """
    # Get model instance
    model = get_topic_model()

    _num_topics = num_topics if num_topics is not None else settings.top_k_topics

    # Get topic indices and probabilities
    topic_indices, _ = model.transform([text])

    topics = []
    for i, topic_idx in enumerate(topic_indices):
        if _num_topics and i >= _num_topics:
            break
        # Convert possible numpy integer to Python int
        topic_idx_int = int(topic_idx)
        if topic_idx_int != -1:  # Skip outlier topic (-1)
            topic_info: list[tuple[str, float]] = model.get_topic(topic_idx_int)  # type: ignore
            if topic_info:
                topics.extend([info[0] for info in topic_info])

    return topics


async def handle_extraction(text: str) -> tuple[list[str], list[str]]:
    """
    Handle topic and entity extraction for a message.

    Args:
        text: The text to process

    Returns:
        Tuple of extracted topics and entities
    """
    # Extract topics if enabled
    topics = []
    if settings.enable_topic_extraction:
        if settings.topic_model_source == "BERTopic":
            topics = extract_topics_bertopic(text)
        else:
            topics = await extract_topics_llm(text)

    # Extract entities if enabled
    entities = []
    if settings.enable_ner:
        entities = extract_entities(text)

    # Merge with existing topics and entities
```

```
    if topics:
        topics = list(set(topics))
    if entities:
        entities = list(set(entities))

    return topics, entities


DISCRETE_EXTRACTION_PROMPT = """
    You are a long-memory manager. Your job is to analyze text and extract
    information that might be useful in future conversations with users.

    Extract two types of memories:
    1. EPISODIC: Personal experiences specific to a user or agent.
       Example: "User prefers window seats" or "User had a bad experience in Paris"

    2. SEMANTIC: User preferences and general knowledge outside of your training data.
       Example: "Trek discontinued the Trek 520 steel touring bike in 2023"

    For each memory, return a JSON object with the following fields:
    - type: str --The memory type, either "episodic" or "semantic"
    - text: str -- The actual information to store
    - topics: list[str] -- The topics of the memory (top {top_k_topics})
    - entities: list[str] -- The entities of the memory
    -

    Return a list of memories, for example:
    {{
        "memories": [
            {{
                "type": "semantic",
                "text": "User prefers window seats",
                "topics": ["travel", "airline"],
                "entities": ["User", "window seat"],
            }},
            {{
                "type": "episodic",
                "text": "Trek discontinued the Trek 520 steel touring bike in 2023",
                "topics": ["travel", "bicycle"],
                "entities": ["Trek", "Trek 520 steel touring bike"],
            }},
        ]
    }}

    IMPORTANT RULES:
    1. Only extract information that would be genuinely useful for future interactions.
    2. Do not extract procedural knowledge - that is handled by the system's built-in tools and prompts.
    3. You are a large language model - do not extract facts that you already know.

    Message:
    {message}

    Extracted memories:
    """


async def extract_discrete_memories(
    redis: Redis | None = None,
    deduplicate: bool = True,
):
    """
    Extract episodic and semantic memories from text using an LLM.
    """
    redis = await get_redis_conn()
    client = await get_model_client(settings.generation_model)
    query = FilterQuery(
        filter_expression=(Tag("discrete_memory_extracted") == "f")
        & (Tag("memory_type") == "message")
    )
```

```
    offset = 0

    while True:
        query.paging(num=25, offset=offset)
        search_index = get_search_index(redis=redis)
        messages = await search_index.query(query)
        discrete_memories = []

        for message in messages:
            if not message or not message.get("text"):
                logger.info(f"Deleting memory with no text: {message}")
                await redis.delete(message["id"])
                continue
            id_ = message.get("id_")
            if not id_:
                logger.error(f"Skipping memory with no ID: {message}")
                continue

            async for attempt in AsyncRetrying(stop=stop_after_attempt(3)):
                with attempt:
                    response = await client.create_chat_completion(
                        model=settings.generation_model,
                        prompt=DISCRETE_EXTRACTION_PROMPT.format(
                            message=message["text"], top_k_topics=settings.top_k_topics
                        ),
                        response_format={"type": "json_object"},
                    )
                    try:
                        new_message = json.loads(response.choices[0].message.content)
                    except json.JSONDecodeError:
                        logger.error(
                            f"Error decoding JSON: {response.choices[0].message.content}"
                        )
                        raise
                    try:
                        assert isinstance(new_message, dict)
                        assert isinstance(new_message["memories"], list)
                    except AssertionError:
                        logger.error(
                            f"Invalid response format: {response.choices[0].message.content}"
                        )
                        raise
                    discrete_memories.extend(new_message["memories"])

            await redis.hset(
                name=message["id"],
                key="discrete_memory_extracted",
                value="t",
            )  # type: ignore

        if len(messages) < 25:
            break
        offset += 25

    # TODO: Added to avoid a circular import
    from agent_memory_server.long_term_memory import index_long_term_memories

    if discrete_memories:
        long_term_memories = [
            MemoryRecord(
                id_=str(ULID()),
                text=new_memory["text"],
                memory_type=new_memory.get("type", "episodic"),
                topics=new_memory.get("topics", []),
                entities=new_memory.get("entities", []),
                discrete_memory_extracted="t",
            )
            for new_memory in discrete_memories
        ]
```

```
await index_long_term_memories(
    long_term_memories,
    deduplicate=deduplicate,
)
```

== agent_memory_server/filters.py ==

```
from enum import Enum
from typing import Self

from pydantic import BaseModel
from pydantic.functional_validators import model_validator
from redisvl.query.filter import FilterExpression, Num, Tag


class TagFilter(BaseModel):
    field: str
    eq: str | None = None
    ne: str | None = None
    any: list[str] | None = None
    all: list[str] | None = None

    @model_validator(mode="after")
    def validate_filters(self) -> Self:
        if self.eq is not None and self.ne is not None:
            raise ValueError("eq and ne cannot both be set")
        if self.any is not None and self.all is not None:
            raise ValueError("any and all cannot both be set")
        if self.all is not None and len(self.all) == 0:
            raise ValueError("all cannot be an empty list")
        if self.any is not None and len(self.any) == 0:
            raise ValueError("any cannot be an empty list")
        return self

    def to_filter(self) -> FilterExpression:
        if self.eq is not None:
            return Tag(self.field) == self.eq
        if self.ne is not None:
            return Tag(self.field) != self.ne
        if self.any is not None:
            return Tag(self.field) == self.any
        if self.all is not None:
            return Tag(self.field) == self.all
        raise ValueError("No filter provided")


class EnumFilter(BaseModel):
    """Filter for enum fields - accepts enum values and validates them"""

    field: str
    enum_class: type[Enum]
    eq: str | None = None
    ne: str | None = None
    any: list[str] | None = None
    all: list[str] | None = None

    @model_validator(mode="after")
    def validate_filters(self) -> Self:
        if self.eq is not None and self.ne is not None:
            raise ValueError("eq and ne cannot both be set")
        if self.any is not None and self.all is not None:
            raise ValueError("any and all cannot both be set")
        if self.all is not None and len(self.all) == 0:
            raise ValueError("all cannot be an empty list")
        if self.any is not None and len(self.any) == 0:
            raise ValueError("any cannot be an empty list")

        # Validate enum values
        valid_values = [e.value for e in self.enum_class]

        if self.eq is not None and self.eq not in valid_values:
            raise ValueError(
                f"eq value '{self.eq}' not in valid enum values: {valid_values}"
            )
```

```
        if self.ne is not None and self.ne not in valid_values:
            raise ValueError(
                f"ne value '{self.ne}' not in valid enum values: {valid_values}"
            )
        if self.any is not None:
            for val in self.any:
                if val not in valid_values:
                    raise ValueError(
                        f"any value '{val}' not in valid enum values: {valid_values}"
                    )
        if self.all is not None:
            for val in self.all:
                if val not in valid_values:
                    raise ValueError(
                        f"all value '{val}' not in valid enum values: {valid_values}"
                    )

        return self

    def to_filter(self) -> FilterExpression:
        if self.eq is not None:
            return Tag(self.field) == self.eq
        if self.ne is not None:
            return Tag(self.field) != self.ne
        if self.any is not None:
            return Tag(self.field) == self.any
        if self.all is not None:
            return Tag(self.field) == self.all
        raise ValueError("No filter provided")


class NumFilter(BaseModel):
    field: str
    gt: int | None = None
    lt: int | None = None
    gte: int | None = None
    lte: int | None = None
    eq: int | None = None
    ne: int | None = None
    between: list[float] | None = None
    inclusive: str = "both"

    @model_validator(mode="after")
    def validate_filters(self) -> Self:
        if self.between is not None and len(self.between) != 2:
            raise ValueError("between must be a list of two numbers")
        if self.between is not None and self.eq is not None:
            raise ValueError("between and eq cannot both be set")
        if self.between is not None and self.ne is not None:
            raise ValueError("between and ne cannot both be set")
        if self.between is not None and self.gt is not None:
            raise ValueError("between and gt cannot both be set")
        if self.between is not None and self.lt is not None:
            raise ValueError("between and lt cannot both be set")
        if self.between is not None and self.gte is not None:
            raise ValueError("between and gte cannot both be set")
        return self

    def to_filter(self) -> FilterExpression:
        if self.between is not None:
            return Num(self.field).between(
                int(self.between[0]), int(self.between[1]), self.inclusive
            )
        if self.eq is not None:
            return Num(self.field) == self.eq
        if self.ne is not None:
            return Num(self.field) != self.ne
        if self.gt is not None:
            return Num(self.field) > self.gt
```

```
        if self.lt is not None:
            return Num(self.field) < self.lt
        if self.gte is not None:
            return Num(self.field) >= self.gte
        if self.lte is not None:
            return Num(self.field) <= self.lte
        raise ValueError("No filter provided")


class DateTimeFilter(BaseModel):
    """Filter for datetime fields - accepts datetime objects and converts to timestamps for Redis queries"""

    field: str
    gt: datetime | None = None
    lt: datetime | None = None
    gte: datetime | None = None
    lte: datetime | None = None
    eq: datetime | None = None
    ne: datetime | None = None
    between: list[datetime] | None = None
    inclusive: str = "both"

    @model_validator(mode="after")
    def validate_filters(self) -> Self:
        if self.between is not None and len(self.between) != 2:
            raise ValueError("between must be a list of two datetimes")
        if self.between is not None and self.eq is not None:
            raise ValueError("between and eq cannot both be set")
        if self.between is not None and self.ne is not None:
            raise ValueError("between and ne cannot both be set")
        if self.between is not None and self.gt is not None:
            raise ValueError("between and gt cannot both be set")
        if self.between is not None and self.lt is not None:
            raise ValueError("between and lt cannot both be set")
        if self.between is not None and self.gte is not None:
            raise ValueError("between and gte cannot both be set")
        return self

    def to_filter(self) -> FilterExpression:
        """Convert datetime objects to timestamps for Redis numerical queries"""
        if self.between is not None:
            return Num(self.field).between(
                int(self.between[0].timestamp()),
                int(self.between[1].timestamp()),
                self.inclusive,
            )
        if self.eq is not None:
            return Num(self.field) == int(self.eq.timestamp())
        if self.ne is not None:
            return Num(self.field) != int(self.ne.timestamp())
        if self.gt is not None:
            return Num(self.field) > int(self.gt.timestamp())
        if self.lt is not None:
            return Num(self.field) < int(self.lt.timestamp())
        if self.gte is not None:
            return Num(self.field) >= int(self.gte.timestamp())
        if self.lte is not None:
            return Num(self.field) <= int(self.lte.timestamp())
        raise ValueError("No filter provided")


class SessionId(TagFilter):
    field: str = "session_id"


class UserId(TagFilter):
    field: str = "user_id"
```

```
class Namespace(TagFilter):
    field: str = "namespace"


class CreatedAt(DateTimeFilter):
    field: str = "created_at"


class LastAccessed(DateTimeFilter):
    field: str = "last_accessed"


class Topics(TagFilter):
    field: str = "topics"


class Entities(TagFilter):
    field: str = "entities"


class MemoryType(EnumFilter):
    field: str = "memory_type"
    enum_class: type[Enum] | None = None  # Will be set in __init__

    def __init__(self, **data):
        # Import here to avoid circular imports
        from agent_memory_server.models import MemoryTypeEnum

        data["enum_class"] = MemoryTypeEnum
        super().__init__(**data)


class EventDate(DateTimeFilter):
    field: str = "event_date"
```

== agent_memory_server/healthcheck.py ==

```
import time

from fastapi import APIRouter

from agent_memory_server.models import HealthCheckResponse


router = APIRouter()


@router.get("/health", response_model=HealthCheckResponse)
async def get_health():
    """
    Health check endpoint

    Returns:
        HealthCheckResponse with current timestamp
    """
    # Return current time in milliseconds
    return HealthCheckResponse(now=int(time.time() * 1000))
```

== agent_memory_server/llms.py ==

```
import logging
import os
from enum import Enum
from typing import Any

import anthropic
import numpy as np
from openai import AsyncOpenAI
from pydantic import BaseModel


logger = logging.getLogger(__name__)


class ModelProvider(str, Enum):
    """Type of model provider"""

    OPENAI = "openai"
    ANTHROPIC = "anthropic"


class ModelConfig(BaseModel):
    """Configuration for a model"""

    provider: ModelProvider
    name: str
    max_tokens: int
    embedding_dimensions: int = 1536  # Default for OpenAI ada-002


# Model configurations
MODEL_CONFIGS = {
    # OpenAI Models
    "gpt-3.5-turbo": ModelConfig(
        provider=ModelProvider.OPENAI,
        name="gpt-3.5-turbo",
        max_tokens=4096,
        embedding_dimensions=1536,
    ),
    "gpt-3.5-turbo-16k": ModelConfig(
        provider=ModelProvider.OPENAI,
        name="gpt-3.5-turbo-16k",
        max_tokens=16384,
        embedding_dimensions=1536,
    ),
    "gpt-4": ModelConfig(
        provider=ModelProvider.OPENAI,
        name="gpt-4",
        max_tokens=8192,
        embedding_dimensions=1536,
    ),
    "gpt-4-32k": ModelConfig(
        provider=ModelProvider.OPENAI,
        name="gpt-4-32k",
        max_tokens=32768,
        embedding_dimensions=1536,
    ),
    "gpt-4o": ModelConfig(
        provider=ModelProvider.OPENAI,
        name="gpt-4o",
        max_tokens=128000,
        embedding_dimensions=1536,
    ),
    "gpt-4o-mini": ModelConfig(
        provider=ModelProvider.OPENAI,
        name="gpt-4o-mini",
        max_tokens=128000,
```

```
    embedding_dimensions=1536,
),
# Newer reasoning models
"o1": ModelConfig(
    provider=ModelProvider.OPENAI,
    name="o1",
    max_tokens=200000,
    embedding_dimensions=1536,
),
"o1-mini": ModelConfig(
    provider=ModelProvider.OPENAI,
    name="o1-mini",
    max_tokens=128000,
    embedding_dimensions=1536,
),
"o3-mini": ModelConfig(
    provider=ModelProvider.OPENAI,
    name="o3-mini",
    max_tokens=200000,
    embedding_dimensions=1536,
),
# Embedding models
"text-embedding-ada-002": ModelConfig(
    provider=ModelProvider.OPENAI,
    name="text-embedding-ada-002",
    max_tokens=8191,
    embedding_dimensions=1536,
),
"text-embedding-3-small": ModelConfig(
    provider=ModelProvider.OPENAI,
    name="text-embedding-3-small",
    max_tokens=8191,
    embedding_dimensions=1536,
),
"text-embedding-3-large": ModelConfig(
    provider=ModelProvider.OPENAI,
    name="text-embedding-3-large",
    max_tokens=8191,
    embedding_dimensions=3072,
),
# Anthropic Models
"claude-3-opus-20240229": ModelConfig(
    provider=ModelProvider.ANTHROPIC,
    name="claude-3-opus-20240229",
    max_tokens=200000,
    embedding_dimensions=1536,
),
"claude-3-sonnet-20240229": ModelConfig(
    provider=ModelProvider.ANTHROPIC,
    name="claude-3-sonnet-20240229",
    max_tokens=200000,
    embedding_dimensions=1536,
),
"claude-3-haiku-20240307": ModelConfig(
    provider=ModelProvider.ANTHROPIC,
    name="claude-3-haiku-20240307",
    max_tokens=200000,
    embedding_dimensions=1536,
),
"claude-3-5-sonnet-20240620": ModelConfig(
    provider=ModelProvider.ANTHROPIC,
    name="claude-3-5-sonnet-20240620",
    max_tokens=200000,
    embedding_dimensions=1536,
),
# Latest Anthropic Models
"claude-3-7-sonnet-20250219": ModelConfig(
    provider=ModelProvider.ANTHROPIC,
    name="claude-3-7-sonnet-20250219",
```

```
        max_tokens=200000,
        embedding_dimensions=1536,
    ),
    "claude-3-5-sonnet-20241022": ModelConfig(
        provider=ModelProvider.ANTHROPIC,
        name="claude-3-5-sonnet-20241022",
        max_tokens=200000,
        embedding_dimensions=1536,
    ),
    "claude-3-5-haiku-20241022": ModelConfig(
        provider=ModelProvider.ANTHROPIC,
        name="claude-3-5-haiku-20241022",
        max_tokens=200000,
        embedding_dimensions=1536,
    ),
    # Convenience aliases
    "claude-3-7-sonnet-latest": ModelConfig(
        provider=ModelProvider.ANTHROPIC,
        name="claude-3-7-sonnet-20250219",
        max_tokens=200000,
        embedding_dimensions=1536,
    ),
    "claude-3-5-sonnet-latest": ModelConfig(
        provider=ModelProvider.ANTHROPIC,
        name="claude-3-5-sonnet-20241022",
        max_tokens=200000,
        embedding_dimensions=1536,
    ),
    "claude-3-5-haiku-latest": ModelConfig(
        provider=ModelProvider.ANTHROPIC,
        name="claude-3-5-haiku-20241022",
        max_tokens=200000,
        embedding_dimensions=1536,
    ),
    "claude-3-opus-latest": ModelConfig(
        provider=ModelProvider.ANTHROPIC,
        name="claude-3-opus-20240229",
        max_tokens=200000,
        embedding_dimensions=1536,
    ),
}


def get_model_config(model_name: str) -> ModelConfig:
    """Get configuration for a model"""
    if model_name in MODEL_CONFIGS:
        return MODEL_CONFIGS[model_name]

    # Default to GPT-4o-mini if model not found
    logger.warning(f"Model {model_name} not found in configuration, using gpt-4o-mini")
    return MODEL_CONFIGS["gpt-4o-mini"]


class ChatResponse:
    """Unified wrapper for chat responses from different providers"""

    def __init__(self, choices: list[Any], usage: dict[str, int]):
        self.choices = choices or []
        self.usage = usage or {"total_tokens": 0}

    @property
    def total_tokens(self) -> int:
        return self.usage.get("total_tokens", 0)


class AnthropicClientWrapper:
    """Wrapper for Anthropic client"""

    def __init__(self, api_key: str | None = None):
```

```
    """Initialize the Anthropic client"""
    anthropic_api_key = api_key or os.environ.get("ANTHROPIC_API_KEY")

    if not anthropic_api_key:
        raise ValueError("Anthropic API key is required")

    self.client = anthropic.AsyncAnthropic(api_key=anthropic_api_key)

async def create_chat_completion(
    self,
    model: str,
    prompt: str,
    response_format: dict[str, str] | None = None,
    functions: list[dict[str, Any]] | None = None,
    function_call: dict[str, str] | None = None,
) -> ChatResponse:
    """Create a chat completion using the Anthropic API"""
    try:
        # For Anthropic, we need to handle structured output differently
        if response_format and response_format.get("type") == "json_object":
            prompt = f"{prompt}\n\nYou must respond with a valid JSON object."

        if functions and function_call:
            # Add function schema to prompt
            schema = functions[0]["parameters"]
            prompt = f"{prompt}\n\nYou must respond with a JSON object matching this schema:\n{json.dumps(schema,

        response = await self.client.messages.create(
            model=model,
            messages=[{"role": "user", "content": prompt}],
            max_tokens=1024,
        )

        # Convert to a unified format - safely extract content
        content = ""
        if (
            hasattr(response, "content")
            and response.content
            and len(response.content) > 0
            and hasattr(response.content[0], "text")
        ):
            content = response.content[0].text

        choices = [{"message": {"content": content}}]

        # Handle both object and dictionary usage formats for testing
        input_tokens = output_tokens = 0
        if hasattr(response, "usage"):
            if isinstance(response.usage, dict):
                input_tokens = response.usage.get("input_tokens", 0)
                output_tokens = response.usage.get("output_tokens", 0)
            else:
                input_tokens = getattr(response.usage, "input_tokens", 0)
                output_tokens = getattr(response.usage, "output_tokens", 0)

        usage = {"total_tokens": input_tokens + output_tokens}

        return ChatResponse(choices=choices, usage=usage)
    except Exception as e:
        logger.error(f"Error creating chat completion with Anthropic: {e}")
        raise

async def create_embedding(self, query_vec: list[str]) -> np.ndarray:
    """
    Create embeddings for the given texts
    Note: Anthropic doesn't offer an embedding API, so we'll use OpenAI's
    embeddings or raise an error if needed
    """
    raise NotImplementedError(
```

```
            "Anthropic does not provide an embedding API. "
            "Please use OpenAI for embeddings."
        )


class OpenAIClientWrapper:
    """Wrapper for OpenAI client"""

    def __init__(self, api_key: str | None = None, base_url: str | None = None):
        """Initialize the OpenAI client based on environment variables"""

        # Regular OpenAI setup
        openai_api_base = base_url or os.environ.get("OPENAI_API_BASE")
        openai_api_key = api_key or os.environ.get("OPENAI_API_KEY")

        if not openai_api_key:
            raise ValueError("OpenAI API key is required")

        if openai_api_base:
            self.completion_client = AsyncOpenAI(
                api_key=openai_api_key,
                base_url=openai_api_base,
            )
            self.embedding_client = AsyncOpenAI(
                api_key=openai_api_key,
                base_url=openai_api_base,
            )
        else:
            self.completion_client = AsyncOpenAI(api_key=openai_api_key)
            self.embedding_client = AsyncOpenAI(api_key=openai_api_key)

    async def create_chat_completion(
        self,
        model: str,
        prompt: str,
        response_format: dict[str, str] | None = None,
        functions: list[dict[str, Any]] | None = None,
        function_call: dict[str, str] | None = None,
    ) -> ChatResponse:
        """Create a chat completion using the OpenAI API"""
        try:
            # Build the request parameters
            request_params = {
                "model": model,
                "messages": [{"role": "user", "content": prompt}],
            }

            # Add optional parameters if provided
            if response_format:
                request_params["response_format"] = response_format
            if functions:
                request_params["functions"] = functions
            if function_call:
                request_params["function_call"] = function_call

            response = await self.completion_client.chat.completions.create(
                **request_params
            )

            # Convert to unified format
            # Handle both object and dictionary usage formats for testing
            total_tokens = 0
            if hasattr(response, "usage"):
                if isinstance(response.usage, dict):
                    total_tokens = response.usage.get("total_tokens", 0)
                else:
                    total_tokens = getattr(response.usage, "total_tokens", 0)

            return ChatResponse(
```

```
                choices=response.choices,
                usage={"total_tokens": total_tokens},
            )
        except Exception as e:
            logger.error(f"Error creating chat completion with OpenAI: {e}")
            raise

    async def create_embedding(self, query_vec: list[str]) -> np.ndarray:
        """Create embeddings for the given texts"""
        try:
            embeddings = []
            embedding_model = "text-embedding-ada-002"

            # Process in batches of 20 to avoid rate limits
            batch_size = 20
            for i in range(0, len(query_vec), batch_size):
                batch = query_vec[i : i + batch_size]
                response = await self.embedding_client.embeddings.create(
                    model=embedding_model,
                    input=batch,
                )
                batch_embeddings = [item.embedding for item in response.data]
                embeddings.extend(batch_embeddings)

            return np.array(embeddings, dtype=np.float32)
        except Exception as e:
            logger.error(f"Error creating embedding: {e}")
            raise


# Global LLM client cache
_model_clients = {}


# TODO: This should take a provider as input, not model name, and cache on provider
async def get_model_client(
    model_name: str,
) -> OpenAIClientWrapper | AnthropicClientWrapper:
    """Get the appropriate client for a model using the factory.

    This is a module-level function that caches clients for reuse.

    Args:
        model_name: Name of the model to get a client for

    Returns:
        An appropriate client wrapper for the model
    """
    global _model_clients
    model = None

    if model_name not in _model_clients:
        model_config = get_model_config(model_name)

        if model_config.provider == ModelProvider.OPENAI:
            model = OpenAIClientWrapper(api_key=os.environ.get("OPENAI_API_KEY"))
        if model_config.provider == ModelProvider.ANTHROPIC:
            model = AnthropicClientWrapper(api_key=os.environ.get("ANTHROPIC_API_KEY"))

        if model:
            _model_clients[model_name] = model
            return model

        raise ValueError(f"Unsupported model provider: {model_config.provider}")

    return _model_clients[model_name]
```

== agent_memory_server/logging.py ==

```
import sys

import structlog

from agent_memory_server.config import settings


_configured = False


def configure_logging():
    """Configure structured logging for the application"""
    global _configured
    if _configured:
        return

    # Configure standard library logging based on settings.log_level
    level = getattr(logging, settings.log_level.upper(), logging.INFO)
    handler = logging.StreamHandler(sys.stdout)
    handler.setLevel(level)
    logging.basicConfig(level=level, handlers=[handler], format="%(message)s")

    # Configure structlog with processors honoring the log level and structured output
    structlog.configure(
        processors=[
            structlog.stdlib.filter_by_level,
            structlog.stdlib.add_logger_name,
            structlog.stdlib.add_log_level,
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.format_exc_info,
            structlog.processors.JSONRenderer(),
        ],
        wrapper_class=structlog.stdlib.BoundLogger,
        logger_factory=structlog.stdlib.LoggerFactory(),
        cache_logger_on_first_use=True,
    )
    _configured = True


def get_logger(name: str | None = None) -> structlog.stdlib.BoundLogger:
    """
    Get a configured logger instance.

    Args:
        name: Optional name for the logger (usually __name__)

    Returns:
        A configured logger instance
    """
    return structlog.get_logger(name)
```

== agent_memory_server/long_term_memory.py ==

```
import json
import logging
import time
from datetime import UTC, datetime
from functools import reduce
from typing import Any

from redis.asyncio import Redis
from redis.commands.search.query import Query
from redisvl.query import VectorQuery, VectorRangeQuery
from redisvl.utils.vectorize import OpenAITextVectorizer
from ulid import ULID

from agent_memory_server.config import settings
from agent_memory_server.dependencies import get_background_tasks
from agent_memory_server.extraction import extract_discrete_memories, handle_extraction
from agent_memory_server.filters import (
    CreatedAt,
    Entities,
    EventDate,
    LastAccessed,
    MemoryType,
    Namespace,
    SessionId,
    Topics,
    UserId,
)
from agent_memory_server.llms import (
    AnthropicClientWrapper,
    OpenAIClientWrapper,
    get_model_client,
)
from agent_memory_server.models import (
    MemoryRecord,
    MemoryRecordResult,
    MemoryRecordResults,
    MemoryTypeEnum,
)
from agent_memory_server.utils.keys import Keys
from agent_memory_server.utils.redis import (
    ensure_search_index_exists,
    get_redis_conn,
    get_search_index,
    safe_get,
)


DEFAULT_MEMORY_LIMIT = 1000
MEMORY_INDEX = "memory_idx"

# Prompt for extracting memories from messages in working memory context
WORKING_MEMORY_EXTRACTION_PROMPT = """
You are a memory extraction assistant. Your job is to analyze conversation
messages and extract information that might be useful in future conversations.

Extract two types of memories from the following message:
1. EPISODIC: Experiences or events that have a time dimension.
   (They MUST have a time dimension to be "episodic.")
   Example: "User mentioned they visited Paris last month" or "User had trouble with the login process"

2. SEMANTIC: User preferences, facts, or general knowledge that would be useful long-term.
   Example: "User prefers dark mode UI" or "User works as a data scientist"

For each memory, return a JSON object with the following fields:
- type: str -- The memory type, either "episodic" or "semantic"
- text: str -- The actual information to store
- topics: list[str] -- Relevant topics for this memory
```

```
- entities: list[str] -- Named entities mentioned
- event_date: str | null -- For episodic memories, the date/time when the event occurred (ISO 8601 format), null for

IMPORTANT RULES:
1. Only extract information that would be genuinely useful for future interactions.
2. Do not extract procedural knowledge or instructions.
3. If given `user_id`, focus on user-specific information, preferences, and facts.
4. Return an empty list if no useful memories can be extracted.

Message: {message}

Return format:
{{
    "memories": [
        {{
            "type": "episodic",
            "text": "...",
            "topics": ["..."],
            "entities": ["..."],
            "event_date": "2024-01-15T14:30:00Z"
        }},
        {{
            "type": "semantic",
            "text": "...",
            "topics": ["..."],
            "entities": ["..."],
            "event_date": null
        }}
    ]
}}

Extracted memories:
"""


logger = logging.getLogger(__name__)


async def extract_memory_structure(_id: str, text: str, namespace: str | None):
    redis = await get_redis_conn()

    # Process messages for topic/entity extraction
    topics, entities = await handle_extraction(text)

    # Convert lists to comma-separated strings for TAG fields
    topics_joined = ",".join(topics) if topics else ""
    entities_joined = ",".join(entities) if entities else ""

    await redis.hset(
        Keys.memory_key(_id, namespace),
        mapping={
            "topics": topics_joined,
            "entities": entities_joined,
        },
    )  # type: ignore


def generate_memory_hash(memory: dict) -> str:
    """
    Generate a stable hash for a memory based on text, user_id, and session_id.

    Args:
        memory: Dictionary containing memory data

    Returns:
        A stable hash string
    """
    # Create a deterministic string representation of the key fields
    text = memory.get("text", "")
```

```
    user_id = memory.get("user_id", "") or ""
    session_id = memory.get("session_id", "") or ""

    # Combine the fields in a predictable order
    hash_content = f"{text}|{user_id}|{session_id}"

    # Create a stable hash
    return hashlib.sha256(hash_content.encode()).hexdigest()


async def merge_memories_with_llm(memories: list[dict], llm_client: Any = None) -> dict:
    """
    Use an LLM to merge similar or duplicate memories.

    Args:
        memories: List of memory dictionaries to merge
        llm_client: Optional LLM client to use for merging

    Returns:
        A merged memory dictionary
    """
    # If there's only one memory, just return it
    if len(memories) == 1:
        return memories[0]

    # Create a unified set of topics and entities
    all_topics = set()
    all_entities = set()

    for memory in memories:
        if memory.get("topics"):
            if isinstance(memory["topics"], str):
                all_topics.update(memory["topics"].split(","))
            else:
                all_topics.update(memory["topics"])

        if memory.get("entities"):
            if isinstance(memory["entities"], str):
                all_entities.update(memory["entities"].split(","))
            else:
                all_entities.update(memory["entities"])

    # Get the memory texts for LLM prompt
    memory_texts = [m["text"] for m in memories]

    # Construct the LLM prompt
    instruction = "Merge these similar memories into a single, coherent memory:"

    prompt = f"{instruction}\n\n"
    for i, text in enumerate(memory_texts, 1):
        prompt += f"Memory {i}: {text}\n\n"

    prompt += "\nMerged memory:"

    model_name = "gpt-4o-mini"

    if not llm_client:
        model_client: (
            OpenAIClientWrapper | AnthropicClientWrapper
        ) = await get_model_client(model_name)
    else:
        model_client = llm_client

    response = await model_client.create_chat_completion(
        model=model_name,
        prompt=prompt,  # type: ignore
    )

    # Extract the merged content
```

```
    merged_text = ""
    if response.choices and len(response.choices) > 0:
        # Handle different response formats
        if hasattr(response.choices[0], "message"):
            merged_text = response.choices[0].message.content
        elif hasattr(response.choices[0], "text"):
            merged_text = response.choices[0].text
        else:
            # Fallback if the structure is different
            merged_text = str(response.choices[0])

    # Use the earliest creation timestamp
    created_at = min(int(m.get("created_at", int(time.time()))) for m in memories)

    # Use the most recent last_accessed timestamp
    last_accessed = max(int(m.get("last_accessed", int(time.time()))) for m in memories)

    # Prefer non-empty namespace, user_id, session_id from memories
    namespace = next((m["namespace"] for m in memories if m.get("namespace")), None)
    user_id = next((m["user_id"] for m in memories if m.get("user_id")), None)
    session_id = next((m["session_id"] for m in memories if m.get("session_id")), None)

    # Get the memory type from the first memory
    memory_type = next(
        (m["memory_type"] for m in memories if m.get("memory_type")), "semantic"
    )

    # Get the discrete_memory_extracted from the first memory
    discrete_memory_extracted = next(
        (
            m["discrete_memory_extracted"]
            for m in memories
            if m.get("discrete_memory_extracted")
        ),
        "t",
    )

    # Create the merged memory
    merged_memory = {
        "text": merged_text.strip(),
        "id_": str(ULID()),
        "user_id": user_id,
        "session_id": session_id,
        "namespace": namespace,
        "created_at": created_at,
        "last_accessed": last_accessed,
        "updated_at": int(datetime.now(UTC).timestamp()),
        "topics": list(all_topics) if all_topics else None,
        "entities": list(all_entities) if all_entities else None,
        "memory_type": memory_type,
        "discrete_memory_extracted": discrete_memory_extracted,
    }

    # Generate a new hash for the merged memory
    merged_memory["memory_hash"] = generate_memory_hash(merged_memory)

    return merged_memory


async def compact_long_term_memories(
    limit: int = 1000,
    namespace: str | None = None,
    user_id: str | None = None,
    session_id: str | None = None,
    llm_client: OpenAIClientWrapper | AnthropicClientWrapper | None = None,
    redis_client: Redis | None = None,
    vector_distance_threshold: float = 0.12,
    compact_hash_duplicates: bool = True,
    compact_semantic_duplicates: bool = True,
```

```
) -> int:
    """
    Compact long-term memories by merging duplicates and semantically similar memories.

    This function can identify and merge two types of duplicate memories:
    1. Hash-based duplicates: Memories with identical content (using memory_hash)
    2. Semantic duplicates: Memories with similar meaning but different text

    Returns the count of remaining memories after compaction.
    """
    if not redis_client:
        redis_client = await get_redis_conn()

    if not llm_client:
        llm_client = await get_model_client(model_name="gpt-4o-mini")

    logger.info(
        f"Starting memory compaction: namespace={namespace}, "
        f"user_id={user_id}, session_id={session_id}, "
        f"hash_duplicates={compact_hash_duplicates}, "
        f"semantic_duplicates={compact_semantic_duplicates}"
    )

    # Build filters for memory queries
    filters = []
    if namespace:
        filters.append(f"@namespace:{{{namespace}}}")
    if user_id:
        filters.append(f"@user_id:{{{user_id}}}")
    if session_id:
        filters.append(f"@session_id:{{{session_id}}}")

    filter_str = " ".join(filters) if filters else "*"

    # Track metrics
    memories_merged = 0
    start_time = time.time()

    # Step 1: Compact hash-based duplicates using Redis aggregation
    if compact_hash_duplicates:
        logger.info("Starting hash-based duplicate compaction")
        try:
            index_name = Keys.search_index_name()

            # Create aggregation query to group by memory_hash and find duplicates
            agg_query = (
                f"FT.AGGREGATE {index_name} {filter_str} "
                "GROUPBY 1 @memory_hash "
                "REDUCE COUNT 0 AS count "
                'FILTER "@count>1" '  # Only groups with more than 1 memory
                "SORTBY 2 @count DESC "
                f"LIMIT 0 {limit}"
            )

            # Execute aggregation to find duplicate groups
            duplicate_groups = await redis_client.execute_command(agg_query)

            if duplicate_groups and duplicate_groups[0] > 0:
                num_groups = duplicate_groups[0]
                logger.info(f"Found {num_groups} groups of hash-based duplicates")

                # Process each group of duplicates
                for i in range(1, len(duplicate_groups), 2):
                    try:
                        # Get the hash and count from aggregation results
                        group_data = duplicate_groups[i]
                        memory_hash = None
                        count = 0
```

```
                    for j in range(0, len(group_data), 2):
                        if group_data[j] == b"memory_hash":
                            memory_hash = group_data[j + 1].decode()
                        elif group_data[j] == b"count":
                            count = int(group_data[j + 1])

                    if not memory_hash or count <= 1:
                        continue

                    # Find all memories with this hash
                    # Use FT.SEARCH to find the actual memories with this hash
                    # TODO: Use RedisVL index
                    search_query = (
                        f"FT.SEARCH {index_name} "
                        f"(@memory_hash:{{{memory_hash}}}) {' '.join(filters)} "
                        "RETURN 6 id_ text last_accessed created_at user_id session_id "
                        "SORTBY last_accessed ASC"  # Oldest first
                    )

                    search_results = await redis_client.execute_command(
                        search_query
                    )

                    if search_results and search_results[0] > 1:
                        num_duplicates = search_results[0]

                        # Keep the newest memory (last in sorted results)
                        # and delete the rest
                        memories_to_delete = []

                        for j in range(1, len(search_results), 2):
                            # Skip the last item (newest) which we'll keep
                            if j < (int(num_duplicates) - 1) * 2 + 1:
                                key = search_results[j].decode()
                                memories_to_delete.append(key)

                        # Delete older duplicates
                        if memories_to_delete:
                            pipeline = redis_client.pipeline()
                            for key in memories_to_delete:
                                pipeline.delete(key)

                            await pipeline.execute()
                            memories_merged += len(memories_to_delete)
                            logger.info(
                                f"Deleted {len(memories_to_delete)} hash-based duplicates "
                                f"with hash {memory_hash}"
                            )
                except Exception as e:
                    logger.error(f"Error processing duplicate group: {e}")

            logger.info(
                f"Completed hash-based deduplication. Merged {memories_merged} memories."
            )
        except Exception as e:
            logger.error(f"Error during hash-based duplicate compaction: {e}")

    # Step 2: Compact semantic duplicates using vector search
    semantic_memories_merged = 0
    if compact_semantic_duplicates:
        logger.info("Starting semantic duplicate compaction")
        # Get the correct index name
        index_name = Keys.search_index_name()
        logger.info(f"Using index '{index_name}' for semantic duplicate compaction.")

        # Check if the index exists before proceeding
        try:
            await redis_client.execute_command(f"FT.INFO {index_name}")
        except Exception as info_e:
```

```
    if "unknown index name" in str(info_e).lower():
        logger.info(f"Search index {index_name} doesn't exist, creating it")
        # Ensure 'get_search_index' is called with the correct name to create it if needed
        await ensure_search_index_exists(redis_client, index_name=index_name)
    else:
        logger.warning(
            f"Error checking index '{index_name}': {info_e} - attempting to proceed."
        )

# Get all memories matching the filters, using the correct index name
index = get_search_index(redis_client, index_name=index_name)
query_str = filter_str if filter_str != "*" else "*"

# Create a query to get all memories
q = Query(query_str).paging(0, limit)
q.return_fields("id_", "text", "vector", "user_id", "session_id", "namespace")

# Execute the query to get memories
search_result = None
try:
    search_result = await index.search(q)
except Exception as e:
    logger.error(f"Error searching for memories: {e}")

if search_result and search_result.total > 0:
    logger.info(
        f"Found {search_result.total} memories to check for semantic duplicates"
    )

    # Process memories in batches to avoid overloading Redis
    batch_size = 50
    processed_keys = set()  # Track which memories have been processed

    for i in range(0, len(search_result.docs), batch_size):
        batch = search_result.docs[i : i + batch_size]

        for memory in batch:
            memory_key = safe_get(memory, "id")  # We get the Redis key as "id"
            memory_id = safe_get(memory, "id_")  # This is our own generated ID

            # Skip if already processed
            if memory_key in processed_keys:
                continue

            # Get memory data with error handling
            memory_data = {}
            try:
                memory_data_raw = await redis_client.hgetall(memory_key)  # type: ignore
                if memory_data_raw:
                    # Convert memory data from bytes to strings
                    memory_data = {
                        k.decode() if isinstance(k, bytes) else k: v
                        if isinstance(v, bytes)
                        and (k == b"vector" or k == "vector")
                        else v.decode()
                        if isinstance(v, bytes)
                        else v
                        for k, v in memory_data_raw.items()
                    }
            except Exception as e:
                logger.error(f"Error retrieving memory {memory_key}: {e}")
                continue

            # Skip if memory not found
            if not memory_data:
                continue

            # Convert to LongTermMemory object for deduplication
            memory_type_value = str(memory_data.get("memory_type", "semantic"))
```

```
if memory_type_value not in [
    "episodic",
    "semantic",
    "message",
]:
    memory_type_value = "semantic"

discrete_memory_extracted_value = str(
    memory_data.get("discrete_memory_extracted", "t")
)
if discrete_memory_extracted_value not in ["t", "f"]:
    discrete_memory_extracted_value = "t"

memory_obj = MemoryRecord(
    id_=memory_id,
    text=str(memory_data.get("text", "")),
    user_id=str(memory_data.get("user_id"))
    if memory_data.get("user_id")
    else None,
    session_id=str(memory_data.get("session_id"))
    if memory_data.get("session_id")
    else None,
    namespace=str(memory_data.get("namespace"))
    if memory_data.get("namespace")
    else None,
    created_at=datetime.fromtimestamp(
        int(memory_data.get("created_at", 0))
    ),
    last_accessed=datetime.fromtimestamp(
        int(memory_data.get("last_accessed", 0))
    ),
    topics=str(memory_data.get("topics", "")).split(",")
    if memory_data.get("topics")
    else [],
    entities=str(memory_data.get("entities", "")).split(",")
    if memory_data.get("entities")
    else [],
    memory_type=memory_type_value,  # type: ignore
    discrete_memory_extracted=discrete_memory_extracted_value,  # type: ignore
)

# Add this memory to processed list
processed_keys.add(memory_key)

# Check for semantic duplicates
(
    merged_memory,
    was_merged,
) = await deduplicate_by_semantic_search(
    memory=memory_obj,
    redis_client=redis_client,
    llm_client=llm_client,
    namespace=namespace,
    user_id=user_id,
    session_id=session_id,
    vector_distance_threshold=vector_distance_threshold,
)

if was_merged:
    semantic_memories_merged += 1
    # We need to delete the original memory and save the merged one
    await redis_client.delete(memory_key)

    # Re-index the merged memory
    if merged_memory:
        await index_long_term_memories(
            [merged_memory],
            redis_client=redis_client,
            deduplicate=False,  # Already deduplicated
```

```
                            )
        logger.info(
            f"Completed semantic deduplication. Merged {semantic_memories_merged} memories."
        )

    # Get the count of remaining memories
    total_memories = await count_long_term_memories(
        namespace=namespace,
        user_id=user_id,
        session_id=session_id,
        redis_client=redis_client,
    )

    end_time = time.time()
    total_merged = memories_merged + semantic_memories_merged

    logger.info(
        f"Memory compaction completed in {end_time - start_time:.2f}s. "
        f"Merged {total_merged} memories. "
        f"{total_memories} memories remain."
    )

    return total_memories


async def index_long_term_memories(
    memories: list[MemoryRecord],
    redis_client: Redis | None = None,
    deduplicate: bool = False,
    vector_distance_threshold: float = 0.12,
    llm_client: Any = None,
) -> None:
    """
    Index long-term memories in Redis for search, with optional deduplication

    Args:
        memories: List of long-term memories to index
        redis_client: Optional Redis client to use. If None, a new connection will be created.
        deduplicate: Whether to deduplicate memories before indexing
        vector_distance_threshold: Threshold for semantic similarity
        llm_client: Optional LLM client for semantic merging
    """
    redis = redis_client or await get_redis_conn()
    model_client = (
        llm_client or await get_model_client(model_name=settings.generation_model)
        if deduplicate
        else None
    )
    background_tasks = get_background_tasks()

    # Process memories for deduplication if requested
    processed_memories = []
    if deduplicate:
        for memory in memories:
            current_memory = memory
            was_deduplicated = False

            # Check for id-based duplicates FIRST (Stage 2 requirement)
            if not was_deduplicated:
                deduped_memory, was_overwrite = await deduplicate_by_id(
                    memory=current_memory,
                    redis_client=redis,
                )
                if was_overwrite:
                    # This overwrote an existing memory with the same id
                    current_memory = deduped_memory or current_memory
                    logger.info(f"Overwrote memory with id {memory.id}")
                else:
                    current_memory = deduped_memory or current_memory
```

```
        # Check for hash-based duplicates
        if not was_deduplicated:
            deduped_memory, was_dup = await deduplicate_by_hash(
                memory=current_memory,
                redis_client=redis,
            )
            if was_dup:
                # This is a duplicate, skip it
                was_deduplicated = True
            else:
                current_memory = deduped_memory or current_memory

        # Check for semantic duplicates
        if not was_deduplicated:
            deduped_memory, was_merged = await deduplicate_by_semantic_search(
                memory=current_memory,
                redis_client=redis,
                llm_client=model_client,
                vector_distance_threshold=vector_distance_threshold,
            )
            if was_merged:
                current_memory = deduped_memory or current_memory

        # Add the memory to be indexed if not a pure duplicate
        if not was_deduplicated:
            processed_memories.append(current_memory)
else:
    processed_memories = memories

# If all memories were duplicates, we're done
if not processed_memories:
    logger.info("All memories were duplicates, nothing to index")
    return

# Now proceed with indexing the processed memories
vectorizer = OpenAITextVectorizer()
embeddings = await vectorizer.aembed_many(
    [memory.text for memory in processed_memories],
    batch_size=20,
    as_buffer=True,
)

async with redis.pipeline(transaction=False) as pipe:
    for idx, vector in enumerate(embeddings):
        memory = processed_memories[idx]
        id_ = memory.id_ if memory.id_ else str(ULID())
        key = Keys.memory_key(id_, memory.namespace)

        # Generate memory hash for the memory
        memory_hash = generate_memory_hash(
            {
                "text": memory.text,
                "user_id": memory.user_id or "",
                "session_id": memory.session_id or "",
            }
        )
        print("Memory hash: ", memory_hash)

        await pipe.hset(  # type: ignore
            key,
            mapping={
                "text": memory.text,
                "id_": id_,
                "session_id": memory.session_id or "",
                "user_id": memory.user_id or "",
                "last_accessed": int(memory.last_accessed.timestamp()),
                "created_at": int(memory.created_at.timestamp()),
                "updated_at": int(memory.updated_at.timestamp()),
```

```
                    "namespace": memory.namespace or "",
                    "memory_hash": memory_hash,  # Store the hash for aggregation
                    "memory_type": memory.memory_type,
                    "vector": vector,
                    "discrete_memory_extracted": memory.discrete_memory_extracted,
                    "id": memory.id or "",
                    "persisted_at": int(memory.persisted_at.timestamp())
                    if memory.persisted_at
                    else 0,
                    "extracted_from": ",".join(memory.extracted_from)
                    if memory.extracted_from
                    else "",
                    "event_date": int(memory.event_date.timestamp())
                    if memory.event_date
                    else 0,
                },
            )

            await background_tasks.add_task(
                extract_memory_structure, id_, memory.text, memory.namespace
            )

        await pipe.execute()

    logger.info(f"Indexed {len(processed_memories)} memories")
    if settings.enable_discrete_memory_extraction:
        # Extract discrete memories from the indexed messages and persist
        # them as separate long-term memory records. This process also
        # runs deduplication if requested.
        await background_tasks.add_task(
            extract_discrete_memories,
            deduplicate=deduplicate,
        )


async def search_long_term_memories(
    text: str,
    redis: Redis,
    session_id: SessionId | None = None,
    user_id: UserId | None = None,
    namespace: Namespace | None = None,
    created_at: CreatedAt | None = None,
    last_accessed: LastAccessed | None = None,
    topics: Topics | None = None,
    entities: Entities | None = None,
    distance_threshold: float | None = None,
    memory_type: MemoryType | None = None,
    event_date: EventDate | None = None,
    limit: int = 10,
    offset: int = 0,
) -> MemoryRecordResults:
    """
    Search for long-term memories using vector similarity and filters.
    """
    vectorizer = OpenAITextVectorizer()
    vector = await vectorizer.aembed(text)
    filters = []

    if session_id:
        filters.append(session_id.to_filter())
    if user_id:
        filters.append(user_id.to_filter())
    if namespace:
        filters.append(namespace.to_filter())
    if created_at:
        filters.append(created_at.to_filter())
    if last_accessed:
        filters.append(last_accessed.to_filter())
    if topics:
```

```
    filters.append(topics.to_filter())
if entities:
    filters.append(entities.to_filter())
if memory_type:
    filters.append(memory_type.to_filter())
if event_date:
    filters.append(event_date.to_filter())
filter_expression = reduce(lambda x, y: x & y, filters) if filters else None

if distance_threshold is not None:
    q = VectorRangeQuery(
        vector=vector,
        vector_field_name="vector",
        distance_threshold=distance_threshold,
        num_results=limit,
        return_score=True,
        return_fields=[
            "text",
            "id_",
            "dist",
            "created_at",
            "last_accessed",
            "user_id",
            "session_id",
            "namespace",
            "topics",
            "entities",
            "memory_type",
            "memory_hash",
            "id",
            "persisted_at",
            "extracted_from",
            "event_date",
        ],
    )
else:
    q = VectorQuery(
        vector=vector,
        vector_field_name="vector",
        num_results=limit,
        return_score=True,
        return_fields=[
            "text",
            "id_",
            "dist",
            "created_at",
            "last_accessed",
            "user_id",
            "session_id",
            "namespace",
            "topics",
            "entities",
            "memory_type",
            "memory_hash",
            "id",
            "persisted_at",
            "extracted_from",
            "event_date",
        ],
    )
if filter_expression:
    q.set_filter(filter_expression)

q.paging(offset=offset, num=limit)

index = get_search_index(redis)
search_result = await index.query(q)

results = []
```

```
    memory_hashes = []

    for doc in search_result:
        if safe_get(doc, "memory_hash") not in memory_hashes:
            memory_hashes.append(safe_get(doc, "memory_hash"))
        else:
            continue

        # NOTE: Because this may not be obvious. We index hashes, and we extract
        # topics and entities separately from main long-term indexing. However,
        # when we store the topics and entities, we store them as comma-separated
        # strings in the hash. Our search index picks these up and indexes them
        # in TAG fields, and we get them back as comma-separated strings.
        doc_topics = safe_get(doc, "topics", [])
        if isinstance(doc_topics, str):
            doc_topics = doc_topics.split(",")  # type: ignore

        doc_entities = safe_get(doc, "entities", [])
        if isinstance(doc_entities, str):
            doc_entities = doc_entities.split(",")  # type: ignore

        # Handle extracted_from field
        doc_extracted_from = safe_get(doc, "extracted_from", [])
        if isinstance(doc_extracted_from, str) and doc_extracted_from:
            doc_extracted_from = doc_extracted_from.split(",")  # type: ignore
        elif not doc_extracted_from:
            doc_extracted_from = []

        # Handle event_date field
        doc_event_date = safe_get(doc, "event_date", 0)
        parsed_event_date = None
        if doc_event_date and int(doc_event_date) != 0:
            parsed_event_date = datetime.fromtimestamp(int(doc_event_date))

        results.append(
            MemoryRecordResult(
                id_=safe_get(doc, "id_"),
                text=safe_get(doc, "text", ""),
                dist=float(safe_get(doc, "vector_distance", 0)),
                created_at=datetime.fromtimestamp(int(safe_get(doc, "created_at", 0))),
                updated_at=datetime.fromtimestamp(int(safe_get(doc, "updated_at", 0))),
                last_accessed=datetime.fromtimestamp(
                    int(safe_get(doc, "last_accessed", 0))
                ),
                user_id=safe_get(doc, "user_id"),
                session_id=safe_get(doc, "session_id"),
                namespace=safe_get(doc, "namespace"),
                topics=doc_topics,
                entities=doc_entities,
                memory_hash=safe_get(doc, "memory_hash"),
                memory_type=safe_get(doc, "memory_type", "message"),
                id=safe_get(doc, "id"),
                persisted_at=datetime.fromtimestamp(
                    int(safe_get(doc, "persisted_at", 0))
                )
                if safe_get(doc, "persisted_at", 0) != 0
                else None,
                extracted_from=doc_extracted_from,
                event_date=parsed_event_date,
            )
        )

    # Handle different types of search_result - fix the linter error
    total_results = len(results)
    try:
        # Check if search_result has a total attribute and use it
        total_attr = getattr(search_result, "total", None)
        if total_attr is not None:
            total_results = int(total_attr)
```

```
    except (AttributeError, TypeError):
        # Fallback to list length if search_result is a list or doesn't have total
        total_results = (
            len(search_result) if isinstance(search_result, list) else len(results)
        )

    logger.info(f"Found {len(results)} results for query")
    return MemoryRecordResults(
        total=total_results,
        memories=results,
        next_offset=offset + limit if offset + limit < total_results else None,
    )


async def search_memories(
    text: str,
    redis: Redis,
    session_id: SessionId | None = None,
    user_id: UserId | None = None,
    namespace: Namespace | None = None,
    created_at: CreatedAt | None = None,
    last_accessed: LastAccessed | None = None,
    topics: Topics | None = None,
    entities: Entities | None = None,
    distance_threshold: float | None = None,
    memory_type: MemoryType | None = None,
    event_date: EventDate | None = None,
    limit: int = 10,
    offset: int = 0,
    include_working_memory: bool = True,
    include_long_term_memory: bool = True,
) -> MemoryRecordResults:
    """
    Search for memories across both working memory and long-term storage.

    This provides a search interface that spans all memory types and locations.

    Args:
        text: Search query text
        redis: Redis client
        session_id: Filter by session ID
        user_id: Filter by user ID
        namespace: Filter by namespace
        created_at: Filter by creation date
        last_accessed: Filter by last access date
        topics: Filter by topics
        entities: Filter by entities
        distance_threshold: Distance threshold for semantic search
        memory_type: Filter by memory type
        limit: Maximum number of results to return
        offset: Offset for pagination
        include_working_memory: Whether to include working memory in search
        include_long_term_memory: Whether to include long-term memory in search

    Returns:
        Combined search results from both working and long-term memory
    """
    from agent_memory_server import working_memory

    all_results = []
    total_count = 0

    # Search long-term memory if enabled
    if include_long_term_memory and settings.long_term_memory:
        try:
            long_term_results = await search_long_term_memories(
                text=text,
                redis=redis,
                session_id=session_id,
```

```
            user_id=user_id,
            namespace=namespace,
            created_at=created_at,
            last_accessed=last_accessed,
            topics=topics,
            entities=entities,
            distance_threshold=distance_threshold,
            memory_type=memory_type,
            event_date=event_date,
            limit=limit,
            offset=offset,
        )
        all_results.extend(long_term_results.memories)
        total_count += long_term_results.total

        logger.info(
            f"Found {len(long_term_results.memories)} long-term memory results"
        )
    except Exception as e:
        logger.error(f"Error searching long-term memory: {e}")

# Search working memory if enabled
if include_working_memory:
    try:
        # Get all working memory sessions if no specific session filter
        if session_id and hasattr(session_id, "eq") and session_id.eq:
            # Search specific session
            session_ids_to_search = [session_id.eq]
        else:
            # Get all sessions for broader search
            from agent_memory_server import messages

            namespace_value = None
            if namespace and hasattr(namespace, "eq"):
                namespace_value = namespace.eq

            _, session_ids_to_search = await messages.list_sessions(
                redis=redis,
                limit=1000,  # Get a reasonable number of sessions
                offset=0,
                namespace=namespace_value,
            )

        # Search working memory in relevant sessions
        working_memory_results = []
        for session_id_str in session_ids_to_search:
            try:
                working_mem = await working_memory.get_working_memory(
                    session_id=session_id_str,
                    namespace=namespace_value if namespace else None,
                    redis_client=redis,
                )

                if working_mem and working_mem.memories:
                    # Filter memories based on criteria
                    filtered_memories = working_mem.memories

                    # Apply memory_type filter
                    if memory_type:
                        if hasattr(memory_type, "eq") and memory_type.eq:
                            filtered_memories = [
                                mem
                                for mem in filtered_memories
                                if mem.memory_type == memory_type.eq
                            ]
                        elif hasattr(memory_type, "any") and memory_type.any:
                            filtered_memories = [
                                mem
                                for mem in filtered_memories
```

```
                                if mem.memory_type in memory_type.any
                            ]

                        # Apply user_id filter
                        if user_id and hasattr(user_id, "eq") and user_id.eq:
                            filtered_memories = [
                                mem
                                for mem in filtered_memories
                                if mem.user_id == user_id.eq
                            ]

                        # Convert to MemoryRecordResult format and add to results
                        for memory in filtered_memories:
                            # Simple text matching for working memory (no vector search)
                            if text.lower() in memory.text.lower():
                                working_memory_results.append(
                                    MemoryRecordResult(
                                        id_=memory.id_ or "",
                                        text=memory.text,
                                        dist=0.0,  # No vector distance for working memory
                                        created_at=memory.created_at or 0,
                                        updated_at=memory.updated_at or 0,
                                        last_accessed=memory.last_accessed or 0,
                                        user_id=memory.user_id,
                                        session_id=session_id_str,
                                        namespace=memory.namespace,
                                        topics=memory.topics or [],
                                        entities=memory.entities or [],
                                        memory_hash="",  # Working memory doesn't have hash
                                        memory_type=memory.memory_type,
                                        id=memory.id,
                                        persisted_at=memory.persisted_at,
                                        event_date=memory.event_date,
                                    )
                                )

                except Exception as e:
                    logger.warning(
                        f"Error searching working memory for session {session_id_str}: {e}"
                    )
                    continue

            all_results.extend(working_memory_results)
            total_count += len(working_memory_results)

            logger.info(f"Found {len(working_memory_results)} working memory results")

        except Exception as e:
            logger.error(f"Error searching working memory: {e}")

    # Sort combined results by relevance (distance for long-term, text match quality for working)
    # For simplicity, put working memory results first (distance 0.0), then long-term by distance
    all_results.sort(key=lambda x: (x.dist, x.created_at))

    # Apply pagination to combined results
    paginated_results = all_results[offset : offset + limit] if all_results else []

    logger.info(
        f"Memory search found {len(all_results)} total results, returning {len(paginated_results)}"
    )

    return MemoryRecordResults(
        total=total_count,
        memories=paginated_results,
        next_offset=offset + limit if offset + limit < len(all_results) else None,
    )


async def count_long_term_memories(
```

```
    namespace: str | None = None,
    user_id: str | None = None,
    session_id: str | None = None,
    redis_client: Redis | None = None,
) -> int:
    """
    Count the total number of long-term memories matching the given filters.

    Args:
        namespace: Optional namespace filter
        user_id: Optional user ID filter
        session_id: Optional session ID filter
        redis_client: Optional Redis client

    Returns:
        Total count of memories matching filters
    """
    # TODO: Use RedisVL here.
    if not redis_client:
        redis_client = await get_redis_conn()

    # Build filters for the query
    filters = []
    if namespace:
        filters.append(f"@namespace:{{{namespace}}}")
    if user_id:
        filters.append(f"@user_id:{{{user_id}}}")
    if session_id:
        filters.append(f"@session_id:{{{session_id}}}")

    filter_str = " ".join(filters) if filters else "*"

    # Execute a search to get the total count
    index_name = Keys.search_index_name()
    query = f"FT.SEARCH {index_name} {filter_str} LIMIT 0 0"

    try:
        # First try to check if the index exists
        try:
            await redis_client.execute_command(f"FT.INFO {index_name}")
        except Exception as info_e:
            if "unknown index name" in str(info_e).lower():
                # Index doesn't exist, create it
                logger.info(f"Search index {index_name} doesn't exist, creating it")
                await ensure_search_index_exists(redis_client)
            else:
                logger.warning(f"Error checking index: {info_e}")

        result = await redis_client.execute_command(query)
        # First element in the result is the total count
        if result and len(result) > 0:
            return result[0]
        return 0
    except Exception as e:
        logger.error(f"Error counting memories: {e}")
        return 0


async def deduplicate_by_hash(
    memory: MemoryRecord,
    redis_client: Redis | None = None,
    namespace: str | None = None,
    user_id: str | None = None,
    session_id: str | None = None,
) -> tuple[MemoryRecord | None, bool]:
    """
    Check if a memory has hash-based duplicates and handle accordingly.

    Args:
```

```
    memory: The memory to check for duplicates
    redis_client: Optional Redis client
    namespace: Optional namespace filter
    user_id: Optional user ID filter
    session_id: Optional session ID filter

Returns:
    Tuple of (memory to save (if any), was_duplicate)
"""
if not redis_client:
    redis_client = await get_redis_conn()

# Generate hash for the memory
memory_hash = generate_memory_hash(
    {
        "text": memory.text,
        "user_id": memory.user_id or "",
        "session_id": memory.session_id or "",
    }
)

# Build filters for the search
filters = []
if namespace or memory.namespace:
    ns = namespace or memory.namespace
    filters.append(f"@namespace:{{{ns}}}")
if user_id or memory.user_id:
    uid = user_id or memory.user_id
    filters.append(f"@user_id:{{{uid}}}")
if session_id or memory.session_id:
    sid = session_id or memory.session_id
    filters.append(f"@session_id:{{{sid}}}")

filter_str = " ".join(filters) if filters else ""

# Search for existing memories with the same hash
index_name = Keys.search_index_name()

# Use FT.SEARCH to find memories with this hash
# TODO: Use RedisVL
search_query = (
    f"FT.SEARCH {index_name} "
    f"(@memory_hash:{{{memory_hash}}}) {filter_str} "
    "RETURN 1 id_ "
    "SORTBY last_accessed DESC"  # Newest first
)

search_results = await redis_client.execute_command(search_query)

if search_results and search_results[0] > 0:
    # Found existing memory with the same hash
    logger.info(f"Found existing memory with hash {memory_hash}")

    # Update the last_accessed timestamp of the existing memory
    if search_results[0] >= 1:
        existing_key = search_results[1].decode()
        await redis_client.hset(
            existing_key,
            "last_accessed",
            str(int(datetime.now(UTC).timestamp())),
        )  # type: ignore

        # Don't save this memory, it's a duplicate
        return None, True

# No duplicates found, return the original memory
return memory, False
```

```
async def deduplicate_by_id(
    memory: MemoryRecord,
    redis_client: Redis | None = None,
    namespace: str | None = None,
    user_id: str | None = None,
    session_id: str | None = None,
) -> tuple[MemoryRecord | None, bool]:
    """
    Check if a memory with the same id exists and handle accordingly.
    This implements Stage 2 requirement: use id as the basis for deduplication and overwrites.

    Args:
        memory: The memory to check for id duplicates
        redis_client: Optional Redis client
        namespace: Optional namespace filter
        user_id: Optional user ID filter
        session_id: Optional session ID filter

    Returns:
        Tuple of (memory to save (potentially updated), was_overwrite)
    """
    if not redis_client:
        redis_client = await get_redis_conn()

    # If no id, can't deduplicate by id
    if not memory.id:
        return memory, False

    # Build filters for the search
    filters = []
    if namespace or memory.namespace:
        ns = namespace or memory.namespace
        filters.append(f"@namespace:{{{ns}}}")
    if user_id or memory.user_id:
        uid = user_id or memory.user_id
        filters.append(f"@user_id:{{{uid}}}")
    if session_id or memory.session_id:
        sid = session_id or memory.session_id
        filters.append(f"@session_id:{{{sid}}}")

    filter_str = " ".join(filters) if filters else ""

    # Search for existing memories with the same id
    index_name = Keys.search_index_name()

    # Use FT.SEARCH to find memories with this id
    # TODO: Use RedisVL
    search_query = (
        f"FT.SEARCH {index_name} "
        f"(@id:{{{memory.id}}}) {filter_str} "
        "RETURN 2 id_ persisted_at "
        "SORTBY last_accessed DESC"  # Newest first
    )

    search_results = await redis_client.execute_command(search_query)

    if search_results and search_results[0] > 0:
        # Found existing memory with the same id
        logger.info(f"Found existing memory with id {memory.id}, will overwrite")

        # Get the existing memory key and persisted_at
        existing_key = search_results[1]
        if isinstance(existing_key, bytes):
            existing_key = existing_key.decode()

        existing_persisted_at = "0"
        if len(search_results) > 2:
            existing_persisted_at = search_results[2]
            if isinstance(existing_persisted_at, bytes):
```

```
                existing_persisted_at = existing_persisted_at.decode()

        # Delete the existing memory
        await redis_client.delete(existing_key)

        # If the existing memory was already persisted, preserve that timestamp
        if existing_persisted_at != "0":
            memory.persisted_at = datetime.fromtimestamp(int(existing_persisted_at))

        # Return the memory to be saved (overwriting the existing one)
        return memory, True

    # No existing memory with this id found
    return memory, False


async def deduplicate_by_semantic_search(
    memory: MemoryRecord,
    redis_client: Redis | None = None,
    llm_client: Any = None,
    namespace: str | None = None,
    user_id: str | None = None,
    session_id: str | None = None,
    vector_distance_threshold: float = 0.12,
) -> tuple[MemoryRecord | None, bool]:
    """
    Check if a memory has semantic duplicates and merge if found.

    Args:
        memory: The memory to check for semantic duplicates
        redis_client: Optional Redis client
        llm_client: Optional LLM client for merging
        namespace: Optional namespace filter
        user_id: Optional user ID filter
        session_id: Optional session ID filter
        vector_distance_threshold: Distance threshold for semantic similarity

    Returns:
        Tuple of (memory to save (potentially merged), was_merged)
    """
    if not redis_client:
        redis_client = await get_redis_conn()

    if not llm_client:
        llm_client = await get_model_client(model_name="gpt-4o-mini")

    # Get the vector for the memory
    vectorizer = OpenAITextVectorizer()
    vector = await vectorizer.aembed(memory.text, as_buffer=True)

    # Build filters
    filter_expression = None
    if namespace or memory.namespace:
        ns = namespace or memory.namespace
        filter_expression = Namespace(eq=ns).to_filter()
    if user_id or memory.user_id:
        uid = user_id or memory.user_id
        user_filter = UserId(eq=uid).to_filter()
        filter_expression = (
            user_filter
            if filter_expression is None
            else filter_expression & user_filter
        )
    if session_id or memory.session_id:
        sid = session_id or memory.session_id
        session_filter = SessionId(eq=sid).to_filter()
        filter_expression = (
            session_filter
            if filter_expression is None
```

```
        else filter_expression & session_filter
    )

# Use vector search to find semantically similar memories
index = get_search_index(redis_client)

vector_query = VectorRangeQuery(
    vector=vector,
    vector_field_name="vector",
    distance_threshold=vector_distance_threshold,
    num_results=5,
    return_fields=[
        "id_",
        "text",
        "user_id",
        "session_id",
        "namespace",
        "id",
        "created_at",
        "last_accessed",
        "topics",
        "entities",
        "memory_type",
    ],
)

if filter_expression:
    vector_query.set_filter(filter_expression)

vector_search_result = await index.query(vector_query)

if vector_search_result and len(vector_search_result) > 0:
    # Found semantically similar memories
    similar_memory_keys = []
    for similar_memory in vector_search_result:
        similar_memory_keys.append(similar_memory["id"])
        similar_memory["created_at"] = similar_memory.get(
            "created_at", int(datetime.now(UTC).timestamp())
        )
        similar_memory["last_accessed"] = similar_memory.get(
            "last_accessed", int(datetime.now(UTC).timestamp())
        )
        # Merge the memories
        merged_memory = await merge_memories_with_llm(
            [memory.model_dump()] + [similar_memory],
            llm_client=llm_client,
        )

        # Convert back to LongTermMemory
        merged_memory_obj = MemoryRecord(
            id_=memory.id_ or str(ULID()),
            text=merged_memory["text"],
            user_id=merged_memory["user_id"],
            session_id=merged_memory["session_id"],
            namespace=merged_memory["namespace"],
            created_at=merged_memory["created_at"],
            last_accessed=merged_memory["last_accessed"],
            topics=merged_memory.get("topics", []),
            entities=merged_memory.get("entities", []),
            memory_type=merged_memory.get("memory_type", "semantic"),
            discrete_memory_extracted=merged_memory.get(
                "discrete_memory_extracted", "t"
            ),
        )

        # Delete the similar memories if requested
        for key in similar_memory_keys:
            await redis_client.delete(key)
```

```
        logger.info(
            f"Merged new memory with {len(similar_memory_keys)} semantic duplicates"
        )
        return merged_memory_obj, True

    # No similar memories found or error occurred
    return memory, False


async def promote_working_memory_to_long_term(
    session_id: str,
    namespace: str | None = None,
    redis_client: Redis | None = None,
) -> int:
    """
    Promote eligible working memory records to long-term storage.

    This function:
    1. Identifies memory records with no persisted_at from working memory
    2. For message records, runs extraction to generate semantic/episodic memories
    3. Uses id to detect and replace duplicates in long-term memory
    4. Persists the record and stamps it with persisted_at = now()
    5. Updates the working memory session store to reflect new timestamps

    Args:
        session_id: The session ID to promote memories from
        namespace: Optional namespace for the session
        redis_client: Optional Redis client to use

    Returns:
        Number of memories promoted to long-term storage
    """

    from agent_memory_server import working_memory
    from agent_memory_server.utils.redis import get_redis_conn

    redis = redis_client or await get_redis_conn()

    # Get current working memory
    current_working_memory = await working_memory.get_working_memory(
        session_id=session_id,
        namespace=namespace,
        redis_client=redis,
    )

    if not current_working_memory:
        logger.debug(f"No working memory found for session {session_id}")
        return 0

    # Find memories with no persisted_at (eligible for promotion)
    unpersisted_memories = [
        memory
        for memory in current_working_memory.memories
        if memory.persisted_at is None
    ]

    if not unpersisted_memories:
        logger.debug(f"No unpersisted memories found in session {session_id}")
        return 0

    logger.info(
        f"Promoting {len(unpersisted_memories)} memories from session {session_id}"
    )

    promoted_count = 0
    updated_memories = []
    extracted_memories = []

    # Stage 7: Extract memories from message records if enabled
```

```
if settings.enable_discrete_memory_extraction:
    message_memories = [
        memory
        for memory in unpersisted_memories
        if memory.memory_type == MemoryTypeEnum.MESSAGE
        and memory.discrete_memory_extracted == "f"
    ]

    if message_memories:
        logger.info(
            f"Extracting memories from {len(message_memories)} message records"
        )
        extracted_memories = await extract_memories_from_messages(message_memories)

        # Mark message memories as extracted
        for message_memory in message_memories:
            message_memory.discrete_memory_extracted = "t"

for memory in current_working_memory.memories:
    if memory.persisted_at is None:
        # This memory needs to be promoted

        # Check for id-based duplicates and handle accordingly
        deduped_memory, was_overwrite = await deduplicate_by_id(
            memory=memory,
            redis_client=redis,
        )

        # Set persisted_at timestamp
        current_memory = deduped_memory or memory
        current_memory.persisted_at = datetime.now(UTC)

        # Index the memory in long-term storage
        await index_long_term_memories(
            [current_memory],
            redis_client=redis,
            deduplicate=False,  # Already deduplicated by id
        )

        promoted_count += 1
        updated_memories.append(current_memory)

        if was_overwrite:
            logger.info(f"Overwrote existing memory with id {memory.id}")
        else:
            logger.info(f"Promoted new memory with id {memory.id}")
    else:
        # This memory is already persisted, keep as-is
        updated_memories.append(memory)

# Add extracted memories to working memory for future promotion
if extracted_memories:
    logger.info(
        f"Adding {len(extracted_memories)} extracted memories to working memory"
    )
    updated_memories.extend(extracted_memories)

# Update working memory with the new persisted_at timestamps and extracted memories
if promoted_count > 0 or extracted_memories:
    updated_working_memory = current_working_memory.model_copy()
    updated_working_memory.memories = updated_memories
    updated_working_memory.updated_at = datetime.now(UTC)

    await working_memory.set_working_memory(
        working_memory=updated_working_memory,
        redis_client=redis,
    )

    logger.info(
```

```
            f"Successfully promoted {promoted_count} memories to long-term storage"
            + (
                f" and extracted {len(extracted_memories)} new memories"
                if extracted_memories
                else ""
            )
        )

    return promoted_count


async def extract_memories_from_messages(
    message_records: list[MemoryRecord],
    llm_client: OpenAIClientWrapper | AnthropicClientWrapper | None = None,
) -> list[MemoryRecord]:
    """
    Extract semantic and episodic memories from message records.

    Args:
        message_records: List of message-type memory records to extract from
        llm_client: Optional LLM client for extraction

    Returns:
        List of extracted memory records with extracted_from field populated
    """
    if not message_records:
        return []

    client = llm_client or await get_model_client(settings.generation_model)
    extracted_memories = []

    for message_record in message_records:
        if message_record.memory_type != MemoryTypeEnum.MESSAGE:
            continue

        try:
            # Use LLM to extract memories from the message
            response = await client.create_chat_completion(
                model=settings.generation_model,
                prompt=WORKING_MEMORY_EXTRACTION_PROMPT.format(
                    message=message_record.text
                ),
                response_format={"type": "json_object"},
            )

            extraction_result = json.loads(response.choices[0].message.content)

            if "memories" in extraction_result and extraction_result["memories"]:
                for memory_data in extraction_result["memories"]:
                    # Parse event_date if provided
                    event_date = None
                    if memory_data.get("event_date"):
                        try:
                            event_date = datetime.fromisoformat(
                                memory_data["event_date"].replace("Z", "+00:00")
                            )
                        except (ValueError, TypeError) as e:
                            logger.warning(
                                f"Could not parse event_date '{memory_data.get('event_date')}': {e}"
                            )

                    # Create a new memory record from the extraction
                    extracted_memory = MemoryRecord(
                        id=str(ULID()),  # Server-generated ID
                        text=memory_data["text"],
                        memory_type=memory_data.get("type", "semantic"),
                        topics=memory_data.get("topics", []),
                        entities=memory_data.get("entities", []),
                        extracted_from=[message_record.id] if message_record.id else [],
```

```
                    event_date=event_date,
                    # Inherit context from the source message
                    session_id=message_record.session_id,
                    user_id=message_record.user_id,
                    namespace=message_record.namespace,
                    persisted_at=None,  # Will be set during promotion
                    discrete_memory_extracted="t",
                )
                extracted_memories.append(extracted_memory)

            logger.info(
                f"Extracted {len(extraction_result['memories'])} memories from message {message_record.id}"
            )

    except Exception as e:
        logger.error(
            f"Error extracting memories from message {message_record.id}: {e}"
        )
        continue

return extracted_memories
```

== agent_memory_server/main.py ==

```
import sys
from contextlib import asynccontextmanager

import uvicorn
from fastapi import FastAPI

from agent_memory_server.api import router as memory_router
from agent_memory_server.config import settings
from agent_memory_server.docket_tasks import register_tasks
from agent_memory_server.healthcheck import router as health_router
from agent_memory_server.llms import MODEL_CONFIGS, ModelProvider
from agent_memory_server.logging import get_logger
from agent_memory_server.utils.redis import (
    _redis_pool as connection_pool,
    ensure_search_index_exists,
    get_redis_conn,
)


logger = get_logger(__name__)


@asynccontextmanager
async def lifespan(app: FastAPI):
    """Initialize the application on startup"""
    logger.info("Starting Redis Agent Memory Server ")

    # Check for required API keys
    available_providers = []

    if settings.openai_api_key:
        available_providers.append(ModelProvider.OPENAI)
    else:
        logger.warning("OpenAI API key not set, OpenAI models will not be available")

    if settings.anthropic_api_key:
        available_providers.append(ModelProvider.ANTHROPIC)
    else:
        logger.warning(
            "Anthropic API key not set, Anthropic models will not be available"
        )

    # Check if the configured models are available
    generation_model_config = MODEL_CONFIGS.get(settings.generation_model)
    embedding_model_config = MODEL_CONFIGS.get(settings.embedding_model)

    if (
        generation_model_config
        and generation_model_config.provider not in available_providers
    ):
        logger.warning(
            f"Selected generation model {settings.generation_model} requires {generation_model_config.provider} API k
        )

    if (
        embedding_model_config
        and embedding_model_config.provider not in available_providers
    ):
        logger.warning(
            f"Selected embedding model {settings.embedding_model} requires {embedding_model_config.provider} API key"
        )

    # If long-term memory is enabled but OpenAI isn't available, warn user
    if settings.long_term_memory and ModelProvider.OPENAI not in available_providers:
        logger.warning(
            "Long-term memory requires OpenAI for embeddings, but OpenAI API key is not set"
        )
```

```
# Set up RediSearch index if long-term memory is enabled
if settings.long_term_memory:
    redis = await get_redis_conn()

    # Get embedding dimensions from model config
    embedding_model_config = MODEL_CONFIGS.get(settings.embedding_model)
    vector_dimensions = (
        str(embedding_model_config.embedding_dimensions)
        if embedding_model_config
        else "1536"
    )
    distance_metric = "COSINE"

    try:
        await ensure_search_index_exists(
            redis,
            index_name=settings.redisvl_index_name,
            vector_dimensions=vector_dimensions,
            distance_metric=distance_metric,
        )
    except Exception as e:
        logger.error(f"Failed to ensure RediSearch index: {e}")
        raise

# Initialize Docket for background tasks if enabled
if settings.use_docket:
    try:
        await register_tasks()
        logger.info("Initialized Docket for background tasks")
        logger.info("To run the worker, use one of these methods:")
        logger.info(
            "1. CLI: docket worker --tasks agent_memory_server.docket_tasks:task_collection"
        )
        logger.info("2. Python: python -m agent_memory_server.worker")
    except Exception as e:
        logger.error(f"Failed to initialize Docket: {e}")
        raise

# Show available models
openai_models = [
    model
    for model, config in MODEL_CONFIGS.items()
    if config.provider == ModelProvider.OPENAI
    and ModelProvider.OPENAI in available_providers
]
anthropic_models = [
    model
    for model, config in MODEL_CONFIGS.items()
    if config.provider == ModelProvider.ANTHROPIC
    and ModelProvider.ANTHROPIC in available_providers
]

if openai_models:
    logger.info(f"Available OpenAI models: {', '.join(openai_models)}")
if anthropic_models:
    logger.info(f"Available Anthropic models: {', '.join(anthropic_models)}")

logger.info(
    "Redis Agent Memory Server initialized",
    window_size=settings.window_size,
    generation_model=settings.generation_model,
    embedding_model=settings.embedding_model,
    long_term_memory=settings.long_term_memory,
)

yield

logger.info("Shutting down Redis Agent Memory Server")
```

```
    if connection_pool is not None:
        await connection_pool.aclose()


# Create FastAPI app
app = FastAPI(title="Redis Agent Memory Server", lifespan=lifespan)


app.include_router(health_router)
app.include_router(memory_router)


def on_start_logger(port: int):
    """Log startup information"""
    print("\n-----------------------------------")
    print(f" Redis Agent Memory Server running on port: {port}")
    print("-----------------------------------\n")


# Run the application
if __name__ == "__main__":
    # Parse command line arguments for port
    port = settings.app_port

    # Check if --port argument is provided
    if "--port" in sys.argv:
        try:
            port_index = sys.argv.index("--port") + 1
            if port_index < len(sys.argv):
                port = int(sys.argv[port_index])
                print(f"Using port from command line: {port}")
        except (ValueError, IndexError):
            # If conversion fails or index out of bounds, use default
            print(f"Invalid port argument, using default: {port}")
    else:
        print(f"No port argument provided, using default: {port}")

    # Explicitly unset the PORT environment variable if it exists
    if "PORT" in os.environ:
        port_val = os.environ.pop("PORT")
        print(f"Removed environment variable PORT={port_val}")

    on_start_logger(port)
    uvicorn.run(
        app,  # Using the app instance directly
        host="0.0.0.0",
        port=port,
        reload=False,
    )
```

== agent_memory_server/mcp.py ==

```
import os
from typing import Any

from mcp.server.fastmcp import FastMCP as _FastMCPBase
from ulid import ULID

from agent_memory_server.api import (
    create_long_term_memory as core_create_long_term_memory,
    memory_prompt as core_memory_prompt,
    put_session_memory as core_put_session_memory,
    search_long_term_memory as core_search_long_term_memory,
)
from agent_memory_server.config import settings
from agent_memory_server.dependencies import get_background_tasks
from agent_memory_server.filters import (
    CreatedAt,
    Entities,
    LastAccessed,
    MemoryType,
    Namespace,
    SessionId,
    Topics,
    UserId,
)
from agent_memory_server.models import (
    AckResponse,
    CreateMemoryRecordRequest,
    MemoryMessage,
    MemoryPromptRequest,
    MemoryPromptResponse,
    MemoryRecord,
    MemoryRecordResults,
    ModelNameLiteral,
    SearchRequest,
    WorkingMemory,
    WorkingMemoryRequest,
    WorkingMemoryResponse,
)


logger = logging.getLogger(__name__)

# Default namespace for STDIO mode
DEFAULT_NAMESPACE = os.getenv("MCP_NAMESPACE")


class FastMCP(_FastMCPBase):
    """Extend FastMCP to support optional URL namespace and default STDIO namespace."""

    def __init__(self, *args, default_namespace=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.default_namespace = default_namespace
        self._current_request = None  # Initialize the attribute

    def sse_app(self):
        from mcp.server.sse import SseServerTransport
        from starlette.applications import Starlette
        from starlette.requests import Request
        from starlette.routing import Mount, Route

        sse = SseServerTransport(self.settings.message_path)

        async def handle_sse(request: Request) -> None:
            # Store the request in the FastMCP instance so call_tool can access it
            self._current_request = request

            try:
```

```
            async with sse.connect_sse(
                request.scope,
                request.receive,
                request._send,  # type: ignore
            ) as (read_stream, write_stream):
                await self._mcp_server.run(
                    read_stream,
                    write_stream,
                    self._mcp_server.create_initialization_options(),
                )
        finally:
            # Clean up request reference
            self._current_request = None

    return Starlette(
        debug=self.settings.debug,
        routes=[
            Route(self.settings.sse_path, endpoint=handle_sse),
            Route(f"/{{namespace}}{self.settings.sse_path}", endpoint=handle_sse),
            Mount(self.settings.message_path, app=sse.handle_post_message),
            Mount(
                f"/{{namespace}}{self.settings.message_path}",
                app=sse.handle_post_message,
            ),
        ],
    )

async def call_tool(self, name, arguments):
    # Get the namespace from the request context
    namespace = None
    try:
        # RequestContext doesn't expose the path_params directly
        # We use a ThreadLocal or context variable pattern instead
        from starlette.requests import Request

        request = getattr(self, "_current_request", None)
        if isinstance(request, Request):
            namespace = request.path_params.get("namespace")
    except Exception:
        # Silently continue if we can't get namespace from request
        pass

    # Inject namespace only for tools that accept it
    if name in ("search_long_term_memory", "hydrate_memory_prompt"):
        if namespace and "namespace" not in arguments:
            arguments["namespace"] = Namespace(eq=namespace)
        elif (
            not namespace
            and self.default_namespace
            and "namespace" not in arguments
        ):
            arguments["namespace"] = Namespace(eq=self.default_namespace)
    elif name in ("set_working_memory",):
        if namespace and "namespace" not in arguments:
            arguments["namespace"] = namespace
        elif (
            not namespace
            and self.default_namespace
            and "namespace" not in arguments
        ):
            arguments["namespace"] = self.default_namespace

    return await super().call_tool(name, arguments)

async def run_sse_async(self):
    """Ensure Redis search index exists before starting SSE server."""
    from agent_memory_server.utils.redis import (
        ensure_search_index_exists,
        get_redis_conn,
```

```
        )

        redis = await get_redis_conn()
        await ensure_search_index_exists(redis)
        return await super().run_sse_async()

    async def run_stdio_async(self):
        """Ensure Redis search index exists before starting STDIO MCP server."""
        from agent_memory_server.utils.redis import (
            ensure_search_index_exists,
            get_redis_conn,
        )

        redis = await get_redis_conn()
        await ensure_search_index_exists(redis)
        return await super().run_stdio_async()


INSTRUCTIONS = """
    When responding to user queries, ALWAYS check memory first before answering
    questions about user preferences, history, or personal information.
"""


mcp_app = FastMCP(
    "Redis Agent Memory Server",
    port=settings.mcp_port,
    instructions=INSTRUCTIONS,
    default_namespace=DEFAULT_NAMESPACE,
)


@mcp_app.tool()
async def create_long_term_memories(
    memories: list[MemoryRecord],
) -> AckResponse:
    """
    Create long-term memories that can be searched later.

    This tool saves memories contained in the payload for future retrieval.

    IMPORTANT NOTES ON SESSION IDs:
    - When including a session_id, use the EXACT session identifier from the current conversation
    - NEVER invent or guess a session ID - if you don't know it, omit the field
    - If you want memories accessible across all sessions, omit the session_id field

    COMMON USAGE PATTERNS:

    1. Basic memory creation:
    ```python
    create_long_term_memories(
        memories=[
          {
            "text": "The user prefers dark mode in all applications",
            "user_id": "user_789",
            "namespace": "user_preferences",
            "topics": ["preferences", "ui"],
          }
        ]
    )
    ```

    2. Create multiple memories at once:
    ```python
    create_long_term_memories(
        memories=[
            {"text": "Memory 1"},
            {"text": "Memory 2"},
        ]
```
```

```
    )

    3. Create memories with different namespaces:
    ```python
    create_long_term_memories(
        memories=[
            {"text": "Memory 1", "namespace": "user_preferences"},
            {"text": "Memory 2", "namespace": "user_settings"},
        ]
    )
    ```

    Args:
        memories: A list of MemoryRecord objects to create

    Returns:
        An acknowledgement response indicating success
    """
    # Apply default namespace for STDIO if not provided in memory entries
    if DEFAULT_NAMESPACE:
        for mem in memories:
            if mem.namespace is None:
                mem.namespace = DEFAULT_NAMESPACE

    payload = CreateMemoryRecordRequest(memories=memories)
    return await core_create_long_term_memory(
        payload, background_tasks=get_background_tasks()
)

@mcp_app.tool()
async def search_long_term_memory(
    text: str | None,
    session_id: SessionId | None = None,
    namespace: Namespace | None = None,
    topics: Topics | None = None,
    entities: Entities | None = None,
    created_at: CreatedAt | None = None,
    last_accessed: LastAccessed | None = None,
    user_id: UserId | None = None,
    memory_type: MemoryType | None = None,
    distance_threshold: float | None = None,
    limit: int = 10,
    offset: int = 0,
) -> MemoryRecordResults:
    """
    Search for memories related to a text query.

    Finds memories based on a combination of semantic similarity and input filters.

    This tool performs a semantic search on stored memories using the query text and filters
    in the payload. Results are ranked by relevance.

    DATETIME INPUT FORMAT:
    - All datetime filters accept ISO 8601 formatted strings (e.g., "2023-01-01T00:00:00Z")
    - Timezone-aware datetimes are recommended (use "Z" for UTC or "+HH:MM" for other timezones)
    - Supported operations: gt, gte, lt, lte, eq, ne, between
    - Example: {"gt": "2023-01-01T00:00:00Z", "lt": "2024-01-01T00:00:00Z"}

    IMPORTANT NOTES ON SESSION IDs:
    - When including a session_id filter, use the EXACT session identifier
    - NEVER invent or guess a session ID - if you don't know it, omit this filter
    - If you want to search across all sessions, don't include a session_id filter
    - Session IDs from examples will NOT work with real data

    COMMON USAGE PATTERNS:

    1. Basic search with just query text:
    ```python
```
```

```
search_long_term_memory(text="user's favorite color")
```

2. Search with simple session filter:
```python
search_long_term_memory(text="user's favorite color", session_id={
    "eq": "session_12345"
})
```

3. Search with complex filters:
```python
search_long_term_memory(
    text="user preferences",
    topics={
        "any": ["preferences", "settings"]
    },
    created_at={
        "gt": "2023-01-01T00:00:00Z"
    },
    limit=5
)
```

4. Search with datetime range filters:
```python
search_long_term_memory(
    text="recent conversations",
    created_at={
        "gte": "2024-01-01T00:00:00Z",
        "lt": "2024-02-01T00:00:00Z"
    },
    last_accessed={
        "gt": "2024-01-15T12:00:00Z"
    }
)
```

5. Search with between datetime filter:
```python
search_long_term_memory(
    text="holiday discussions",
    created_at={
        "between": ["2023-12-20T00:00:00Z", "2023-12-31T23:59:59Z"]
    }
)
```

Args:
    text: The semantic search query text (required)
    session_id: Filter by session ID
    namespace: Filter by namespace
    topics: Filter by topics
    entities: Filter by entities
    created_at: Filter by creation date
    last_accessed: Filter by last access date
    user_id: Filter by user ID
    memory_type: Filter by memory type
    distance_threshold: Distance threshold for semantic search
    limit: Maximum number of results
    offset: Offset for pagination

Returns:
    MemoryRecordResults containing matched memories sorted by relevance
"""
try:
    payload = SearchRequest(
        text=text,
        session_id=session_id,

```
            namespace=namespace,
            topics=topics,
            entities=entities,
            created_at=created_at,
            last_accessed=last_accessed,
            user_id=user_id,
            memory_type=memory_type,
            distance_threshold=distance_threshold,
            limit=limit,
            offset=offset,
        )
        results = await core_search_long_term_memory(payload)
        results = MemoryRecordResults(
            total=results.total,
            memories=results.memories,
            next_offset=results.next_offset,
        )
    except Exception as e:
        logger.error(f"Error in search_long_term_memory tool: {e}")
        results = MemoryRecordResults(
            total=0,
            memories=[],
            next_offset=None,
        )
    return results


# Notes that exist outside of the docstring to avoid polluting the LLM prompt:
# 1. The "prompt" abstraction in FastAPI doesn't support search filters, so we use a tool.
# 2. Some applications, such as Cursor, get confused with nested objects in tool parameters,
#    so we use a flat set of parameters instead.
@mcp_app.tool()
async def memory_prompt(
    query: str,
    session_id: SessionId | None = None,
    namespace: Namespace | None = None,
    window_size: int = settings.window_size,
    model_name: ModelNameLiteral | None = None,
    context_window_max: int | None = None,
    topics: Topics | None = None,
    entities: Entities | None = None,
    created_at: CreatedAt | None = None,
    last_accessed: LastAccessed | None = None,
    user_id: UserId | None = None,
    memory_type: MemoryType | None = None,
    distance_threshold: float | None = None,
    limit: int = 10,
    offset: int = 0,
) -> MemoryPromptResponse:
    """
    Hydrate a user query with relevant session history and long-term memories.

    CRITICAL: Use this tool for EVERY question that might benefit from memory context,
    especially when you don't have sufficient information to answer confidently.

    This tool enriches the user's query by retrieving:
    1. Context from the current conversation session
    2. Relevant long-term memories related to the query

    ALWAYS use this tool when:
    - The user references past conversations
    - The question is about user preferences or personal information
    - You need additional context to provide a complete answer
    - The question seems to assume information you don't have in current context

    The function uses the text field from the payload as the user's query,
    and any filters to retrieve relevant memories.

    DATETIME INPUT FORMAT:
```

- All datetime filters accept ISO 8601 formatted strings (e.g., "2023-01-01T00:00:00Z")
- Timezone-aware datetimes are recommended (use "Z" for UTC or "+HH:MM" for other timezones)
- Supported operations: gt, gte, lt, lte, eq, ne, between
- Example: {"gt": "2023-01-01T00:00:00Z", "lt": "2024-01-01T00:00:00Z"}

IMPORTANT NOTES ON SESSION IDs:
- When filtering by session_id, you must provide the EXACT session identifier
- NEVER invent or guess a session ID - if you don't know it, omit this filter
- Session IDs from examples will NOT work with real data

COMMON USAGE PATTERNS:
```python
1. Hydrate a user prompt with long-term memory search:
hydrate_memory_prompt(text="What was my favorite color?")
```

2. Hydrate a user prompt with long-term memory search and session filter:
```
hydrate_memory_prompt(
    text="What is my favorite color?",
    session_id={
        "eq": "session_12345"
    },
    namespace={
        "eq": "user_preferences"
    }
)

3. Hydrate a user prompt with long-term memory search and complex filters:
hydrate_memory_prompt(
    text="What was my favorite color?",
    topics={
        "any": ["preferences", "settings"]
    },
    created_at={
        "gt": "2023-01-01T00:00:00Z"
    },
    limit=5
)

4. Search with datetime range filters:
hydrate_memory_prompt(
    text="What did we discuss recently?",
    created_at={
        "gte": "2024-01-01T00:00:00Z",
        "lt": "2024-02-01T00:00:00Z"
    },
    last_accessed={
        "gt": "2024-01-15T12:00:00Z"
    }
)
```

Args:
    - text: The user's query
    - session_id: Add conversation history from a session
    - namespace: Filter session and long-term memory namespace
    - topics: Search for long-term memories matching topics
    - entities: Search for long-term memories matching entities
    - created_at: Search for long-term memories matching creation date
    - last_accessed: Search for long-term memories matching last access date
    - user_id: Search for long-term memories matching user ID
    - distance_threshold: Distance threshold for semantic search
    - limit: Maximum number of long-term memory results
    - offset: Offset for pagination of long-term memory results

Returns:
    A list of messages, including memory context and the user's query
"""
_session_id = session_id.eq if session_id and session_id.eq else None

```
    session = None

    if _session_id is not None:
        session = WorkingMemoryRequest(
            session_id=_session_id,
            namespace=namespace.eq if namespace and namespace.eq else None,
            window_size=window_size,
            model_name=model_name,
            context_window_max=context_window_max,
        )

    search_payload = SearchRequest(
        text=query,
        session_id=session_id,
        namespace=namespace,
        topics=topics,
        entities=entities,
        created_at=created_at,
        last_accessed=last_accessed,
        user_id=user_id,
        distance_threshold=distance_threshold,
        memory_type=memory_type,
        limit=limit,
        offset=offset,
    )
    _params = {}
    if session is not None:
        _params["session"] = session
    if search_payload is not None:
        _params["long_term_search"] = search_payload

    return await core_memory_prompt(params=MemoryPromptRequest(query=query, **_params))


@mcp_app.tool()
async def set_working_memory(
    session_id: str,
    memories: list[MemoryRecord] | None = None,
    messages: list[MemoryMessage] | None = None,
    context: str | None = None,
    data: dict[str, Any] | None = None,
    namespace: str | None = None,
    user_id: str | None = None,
    ttl_seconds: int = 3600,
) -> WorkingMemoryResponse:
    """
    Set working memory for a session. This works like the PUT /sessions/{id}/memory API endpoint.

    Replaces existing working memory with new content. Can store structured memory records
    and messages, but agents should primarily use this for memory records and JSON data,
    not conversation messages.

    USAGE PATTERNS:

    1. Store structured memory records:
    ```python
    set_working_memory(
        session_id="current_session",
        memories=[
            {
                "text": "User prefers dark mode",
                "id": "pref_dark_mode",
                "memory_type": "semantic",
                "topics": ["preferences", "ui"]
            }
        ]
    )
    ```
```

2. Store arbitrary JSON data separately:
```python
set_working_memory(
    session_id="current_session",
    data={
        "user_settings": {"theme": "dark", "lang": "en"},
        "preferences": {"notifications": True, "sound": False}
    }
)
```

3. Store both memories and JSON data:
```python
set_working_memory(
    session_id="current_session",
    memories=[
        {
            "text": "User prefers dark mode",
            "id": "pref_dark_mode",
            "memory_type": "semantic",
            "topics": ["preferences", "ui"]
        }
    ],
    data={
        "current_settings": {"theme": "dark", "lang": "en"}
    }
)
```

4. Replace entire working memory state:
```python
set_working_memory(
    session_id="current_session",
    memories=[...],  # structured memories
    messages=[...],  # conversation history
    context="Summary of previous conversation",
    user_id="user123"
)
```

Args:
    session_id: The session ID to set memory for (required)
    memories: List of structured memory records (semantic, episodic, message types)
    messages: List of conversation messages (role/content pairs)
    context: Optional summary/context text
    data: Optional dictionary for storing arbitrary JSON data
    namespace: Optional namespace for scoping
    user_id: Optional user ID
    ttl_seconds: TTL for the working memory (default 1 hour)

Returns:
    Updated working memory response (may include summarization if window exceeded)
"""
```python
# Apply default namespace if configured
memory_namespace = namespace
if not memory_namespace and DEFAULT_NAMESPACE:
    memory_namespace = DEFAULT_NAMESPACE

# Auto-generate IDs for memories that don't have them
processed_memories = []
if memories:
    for memory in memories:
        # Handle both MemoryRecord objects and dict inputs
        if isinstance(memory, MemoryRecord):
            # Already a MemoryRecord object, ensure it has an ID
            memory_id = memory.id or str(ULID())
            processed_memory = memory.model_copy(
                update={
                    "id": memory_id,
```

```
                    "persisted_at": None,  # Mark as pending promotion
                }
            )
        else:
            # Dictionary input, convert to MemoryRecord
            memory_dict = dict(memory)
            if not memory_dict.get("id"):
                memory_dict["id"] = str(ULID())
            memory_dict["persisted_at"] = None
            processed_memory = MemoryRecord(**memory_dict)

        processed_memories.append(processed_memory)

# Create the working memory object
working_memory_obj = WorkingMemory(
    session_id=session_id,
    namespace=memory_namespace,
    memories=processed_memories,
    messages=messages or [],
    context=context,
    data=data or {},
    user_id=user_id,
    ttl_seconds=ttl_seconds,
)

# Update working memory via the API - this handles summarization and background promotion
result = await core_put_session_memory(
    session_id=session_id,
    memory=working_memory_obj,
    background_tasks=get_background_tasks(),
)

# Convert to WorkingMemoryResponse to satisfy return type
return WorkingMemoryResponse(**result.model_dump())
```

== agent_memory_server/messages.py ==

```
"""Ses

import json
import logging
import time

from redis import WatchError
from redis.asyncio import Redis

from agent_memory_server.config import settings
from agent_memory_server.dependencies import DocketBackgroundTasks
from agent_memory_server.long_term_memory import index_long_term_memories
from agent_memory_server.models import (
    MemoryMessage,
    MemoryRecord,
    WorkingMemory,
)
from agent_memory_server.summarization import summarize_session
from agent_memory_server.utils.keys import Keys


logger = logging.getLogger(__name__)


async def list_sessions(
    redis: Redis,
    limit: int = 10,
    offset: int = 0,
    namespace: str | None = None,
) -> tuple[int, list[str]]:
    """List sessions"""
    # Calculate start and end indices (0-indexed start, inclusive end)
    start = offset
    end = offset + limit - 1

    sessions_key = Keys.sessions_key(namespace=namespace)

    # Check if the sessions key exists
    await redis.exists(sessions_key)

    # Try to get all sessions directly
    await redis.zrange(sessions_key, 0, -1)

    async with redis.pipeline() as pipe:
        pipe.zcard(sessions_key)
        pipe.zrange(sessions_key, start, end)
        total, session_ids = await pipe.execute()

    return total, [
        s.decode("utf-8") if isinstance(s, bytes) else s for s in session_ids
    ]


async def get_session_memory(
    redis: Redis,
    session_id: str,
    window_size: int = settings.window_size,
    namespace: str | None = None,
) -> WorkingMemory | None:
    """Get a session's memory"""
    sessions_key = Keys.sessions_key(namespace=namespace)
    messages_key = Keys.messages_key(session_id, namespace=namespace)
    metadata_key = Keys.metadata_key(session_id, namespace=namespace)

    session_exists = await redis.zscore(sessions_key, session_id)
    if not session_exists:
        return None
```

```
    async with redis.pipeline() as pipe:
        pipe.lrange(messages_key, -window_size, -1)  # Get the most recent messages
        pipe.hgetall(metadata_key)
        messages_data, metadata = await pipe.execute()

    messages = []
    for msg_data in messages_data:
        if isinstance(msg_data, bytes):
            msg_data = msg_data.decode("utf-8")
        msg = json.loads(msg_data)
        messages.append(MemoryMessage(**msg))

    metadata_dict = {}
    for k, v in metadata.items():
        key = k.decode("utf-8") if isinstance(k, bytes) else k
        value = v.decode("utf-8") if isinstance(v, bytes) else v
        metadata_dict[key] = value

    return WorkingMemory(
        messages=messages,
        memories=[],  # Empty list for structured memories since this is old session storage
        session_id=session_id,
        namespace=namespace,
        **metadata_dict,
    )


async def set_session_memory(
    redis: Redis,
    session_id: str,
    memory: WorkingMemory,
    background_tasks: DocketBackgroundTasks,
):
    """
    Create or update a session's memory

    Args:
        redis: The Redis client
        session_id: The session ID
        memory: The session memory to set
        background_tasks: Background tasks instance
    """
    sessions_key = Keys.sessions_key(namespace=memory.namespace)
    messages_key = Keys.messages_key(session_id, namespace=memory.namespace)
    metadata_key = Keys.metadata_key(session_id, namespace=memory.namespace)
    messages_json = [json.dumps(msg.model_dump()) for msg in memory.messages]
    metadata = memory.model_dump(
        exclude_none=True,
        exclude_unset=True,
        exclude={"messages", "memories", "ttl_seconds"},
    )

    async with redis.pipeline(transaction=True) as pipe:
        await pipe.watch(messages_key, metadata_key)
        pipe.multi()

        while True:
            try:
                current_time = int(time.time())
                pipe.zadd(sessions_key, {session_id: current_time})
                pipe.rpush(messages_key, *messages_json)  # type: ignore
                pipe.hset(metadata_key, mapping=metadata)  # type: ignore
                await pipe.execute()
            except WatchError:
                continue
            break

    # Verify that the session was added to the sessions set
    await redis.zscore(sessions_key, session_id)
```

```
    # List all sessions in the sessions set
    await redis.zrange(sessions_key, 0, -1)

    # Check if window size is exceeded
    current_size = await redis.llen(messages_key)  # type: ignore
    if current_size > settings.window_size:
        # Add summarization task
        await background_tasks.add_task(
            summarize_session,
            session_id,
            settings.generation_model,
            settings.window_size,
        )

    # If long-term memory is enabled, index messages
    if settings.long_term_memory:
        memories = [
            MemoryRecord(
                session_id=session_id,
                text=f"{msg.role}: {msg.content}",
                namespace=memory.namespace,
                memory_type="message",
            )
            for msg in memory.messages
        ]

        await background_tasks.add_task(
            index_long_term_memories,
            memories,
        )


async def delete_session_memory(
    redis: Redis,
    session_id: str,
    namespace: str | None = None,
):
    """Delete a session's memory"""
    # Define keys
    messages_key = Keys.messages_key(session_id, namespace=namespace)
    sessions_key = Keys.sessions_key(namespace=namespace)
    metadata_key = Keys.metadata_key(session_id, namespace=namespace)

    # Create pipeline for deletion
    pipe = redis.pipeline()
    pipe.delete(messages_key, metadata_key)
    pipe.zrem(sessions_key, session_id)
    await pipe.execute()
```

== agent_memory_server/migrations.py ==

```
"""
Simplest possible migrations you could have.
"""

from redis.asyncio import Redis
from ulid import ULID

from agent_memory_server.logging import get_logger
from agent_memory_server.long_term_memory import generate_memory_hash
from agent_memory_server.utils.keys import Keys
from agent_memory_server.utils.redis import get_redis_conn


logger = get_logger(__name__)


async def migrate_add_memory_hashes_1(redis: Redis | None = None) -> None:
    """
    Migration 1: Add memory_hash to all existing memories in Redis
    """
    logger.info("Starting memory hash migration")
    redis = redis or await get_redis_conn()

    # 1. Scan Redis for all memory keys
    memory_keys = []
    cursor = 0

    pattern = Keys.memory_key("*")

    while True:
        cursor, keys = await redis.scan(cursor=cursor, match=pattern, count=100)
        memory_keys.extend(keys)
        if cursor == 0:
            break

    if not memory_keys:
        logger.info("No memories found to migrate")
        return

    # 2. Process memories in batches
    batch_size = 50
    migrated_count = 0

    for i in range(0, len(memory_keys), batch_size):
        batch_keys = memory_keys[i : i + batch_size]
        pipeline = redis.pipeline()

        # First get the data
        for key in batch_keys:
            pipeline.hgetall(key)

        results = await pipeline.execute()

        # Now update with hashes
        update_pipeline = redis.pipeline()
        for j, result in enumerate(results):
            if not result:
                continue

            # Convert bytes to strings
            try:
                memory = {
                    k.decode() if isinstance(k, bytes) else k: v.decode()
                    if isinstance(v, bytes)
                    else v
                    for k, v in result.items()
                    if k in (b"text", b"user_id", b"session_id", b"memory_hash")
                }
```

```
            except Exception as e:
                logger.error(f"Error decoding memory: {result}, {e}")
                continue

            if not memory or "memory_hash" in memory:
                continue

            memory_hash = generate_memory_hash(memory)

            update_pipeline.hset(batch_keys[j], "memory_hash", memory_hash)
            migrated_count += 1

        await update_pipeline.execute()
        logger.info(f"Migrated {migrated_count} memories so far")

    logger.info(f"Migration completed. Added hashes to {migrated_count} memories")


async def migrate_add_discrete_memory_extracted_2(redis: Redis | None = None) -> None:
    """
    Migration 2: Add discrete_memory_extracted to all existing memories in Redis
    """
    logger.info("Starting discrete_memory_extracted migration")
    redis = redis or await get_redis_conn()

    keys = await redis.keys(Keys.memory_key("*"))

    migrated_count = 0
    for key in keys:
        id_ = await redis.hget(name=key, key="id_")  # type: ignore
        if not id_:
            logger.info("Updating memory with no ID to set ID")
            await redis.hset(name=key, key="id_", value=str(ULID()))  # type: ignore
        # extracted: bytes | None = await redis.hget(
        #     name=key, key="discrete_memory_extracted"
        # )  # type: ignore
        # if extracted and extracted.decode() == "t":
        #     continue
        await redis.hset(name=key, key="discrete_memory_extracted", value="f")  # type: ignore
        migrated_count += 1

    logger.info(
        f"Migration completed. Added discrete_memory_extracted (f) to {migrated_count} memories"
    )


async def migrate_add_memory_type_3(redis: Redis | None = None) -> None:
    """
    Migration 3: Add memory_type to all existing memories in Redis
    """
    logger.info("Starting memory_type migration")
    redis = redis or await get_redis_conn()

    keys = await redis.keys(Keys.memory_key("*"))

    migrated_count = 0
    for key in keys:
        id_ = await redis.hget(name=key, key="id_")  # type: ignore
        if not id_:
            logger.info("Updating memory with no ID to set ID")
            await redis.hset(name=key, key="id_", value=str(ULID()))  # type: ignore
        memory_type: bytes | None = await redis.hget(name=key, key="memory_type")  # type: ignore
        if not memory_type:
            await redis.hset(name=key, key="memory_type", value="message")  # type: ignore
        migrated_count += 1

    logger.info(f"Migration completed. Added memory_type to {migrated_count} memories")
```

== agent_memory_server/models.py ==

```
import
from datetime import UTC, datetime
from enum import Enum
from typing import Literal

from mcp.server.fastmcp.prompts import base
from pydantic import BaseModel, Field

from agent_memory_server.config import settings
from agent_memory_server.filters import (
    CreatedAt,
    Entities,
    EventDate,
    LastAccessed,
    MemoryType,
    Namespace,
    SessionId,
    Topics,
    UserId,
)


logger = logging.getLogger(__name__)

JSONTypes = str | float | int | bool | list | dict


class MemoryTypeEnum(str, Enum):
    """Enum for memory types with string values"""

    EPISODIC = "episodic"
    SEMANTIC = "semantic"
    MESSAGE = "message"


# These should match the keys in MODEL_CONFIGS
ModelNameLiteral = Literal[
    "gpt-3.5-turbo",
    "gpt-3.5-turbo-16k",
    "gpt-4",
    "gpt-4-32k",
    "gpt-4o",
    "gpt-4o-mini",
    "o1",
    "o1-mini",
    "o3-mini",
    "text-embedding-ada-002",
    "text-embedding-3-small",
    "text-embedding-3-large",
    "claude-3-opus-20240229",
    "claude-3-sonnet-20240229",
    "claude-3-haiku-20240307",
    "claude-3-5-sonnet-20240620",
    "claude-3-7-sonnet-20250219",
    "claude-3-5-sonnet-20241022",
    "claude-3-5-haiku-20241022",
    "claude-3-7-sonnet-latest",
    "claude-3-5-sonnet-latest",
    "claude-3-5-haiku-latest",
    "claude-3-opus-latest",
]


class MemoryMessage(BaseModel):
    """A message in the memory system"""

    role: str
    content: str
```

```
class SessionListResponse(BaseModel):
    """Response containing a list of sessions"""

    sessions: list[str]
    total: int


class MemoryRecord(BaseModel):
    """A memory record"""

    text: str
    id_: str | None = Field(
        default=None,
        description="Optional ID for the memory record",
    )
    session_id: str | None = Field(
        default=None,
        description="Optional session ID for the memory record",
    )
    user_id: str | None = Field(
        default=None,
        description="Optional user ID for the memory record",
    )
    namespace: str | None = Field(
        default=None,
        description="Optional namespace for the memory record",
    )
    last_accessed: datetime = Field(
        default_factory=lambda: datetime.now(UTC),
        description="Datetime when the memory was last accessed",
    )
    created_at: datetime = Field(
        default_factory=lambda: datetime.now(UTC),
        description="Datetime when the memory was created",
    )
    updated_at: datetime = Field(
        description="Datetime when the memory was last updated",
        default_factory=lambda: datetime.now(UTC),
    )
    topics: list[str] | None = Field(
        default=None,
        description="Optional topics for the memory record",
    )
    entities: list[str] | None = Field(
        default=None,
        description="Optional entities for the memory record",
    )
    memory_hash: str | None = Field(
        default=None,
        description="Hash representation of the memory for deduplication",
    )
    discrete_memory_extracted: Literal["t", "f"] = Field(
        default="f",
        description="Whether memory extraction has run for this memory (only messages)",
    )
    memory_type: MemoryTypeEnum = Field(
        default=MemoryTypeEnum.MESSAGE,
        description="Type of memory",
    )
    id: str | None = Field(
        default=None,
        description="Client-provided ID for deduplication and overwrites",
    )
    persisted_at: datetime | None = Field(
        default=None,
        description="Server-assigned timestamp when memory was persisted to long-term storage",
    )
```

```
    extracted_from: list[str] | None = Field(
        default=None,
        description="List of message IDs that this memory was extracted from",
    )
    event_date: datetime | None = Field(
        default=None,
        description="Date/time when the event described in this memory occurred (primarily for episodic memories)",
    )


class WorkingMemory(BaseModel):
    """Working memory for a session - contains both messages and structured memory records"""

    # Support both message-based memory (conversation) and structured memory records
    messages: list[MemoryMessage] = Field(
        default_factory=list,
        description="Conversation messages (role/content pairs)",
    )
    memories: list[MemoryRecord] = Field(
        default_factory=list,
        description="Structured memory records for promotion to long-term storage",
    )

    # Arbitrary JSON data storage (separate from memories)
    data: dict[str, JSONTypes] | None = Field(
        default=None,
        description="Arbitrary JSON data storage (key-value pairs)",
    )

    # Session context and metadata (moved from SessionMemory)
    context: str | None = Field(
        default=None,
        description="Optional summary of past session messages",
    )
    user_id: str | None = Field(
        default=None,
        description="Optional user ID for the working memory",
    )
    tokens: int = Field(
        default=0,
        description="Optional number of tokens in the working memory",
    )

    # Required session scoping
    session_id: str
    namespace: str | None = Field(
        default=None,
        description="Optional namespace for the working memory",
    )

    # TTL and timestamps
    ttl_seconds: int = Field(
        default=3600,  # 1 hour default
        description="TTL for the working memory in seconds",
    )
    last_accessed: datetime = Field(
        default_factory=lambda: datetime.now(UTC),
        description="Datetime when the working memory was last accessed",
    )
    created_at: datetime = Field(
        default_factory=lambda: datetime.now(UTC),
        description="Datetime when the working memory was created",
    )
    updated_at: datetime = Field(
        default_factory=lambda: datetime.now(UTC),
        description="Datetime when the working memory was last updated",
    )
```

```
class WorkingMemoryResponse(WorkingMemory):
    """Response containing working memory"""


class WorkingMemoryRequest(BaseModel):
    """Request parameters for working memory operations"""

    session_id: str
    namespace: str | None = None
    window_size: int = settings.window_size
    model_name: ModelNameLiteral | None = None
    context_window_max: int | None = None


class AckResponse(BaseModel):
    """Generic acknowledgement response"""

    status: str


class MemoryRecordResult(MemoryRecord):
    """Result from a memory search"""

    dist: float


class MemoryRecordResults(BaseModel):
    """Results from a memory search"""

    memories: list[MemoryRecordResult]
    total: int
    next_offset: int | None = None


class MemoryRecordResultsResponse(MemoryRecordResults):
    """Response containing memory search results"""


class CreateMemoryRecordRequest(BaseModel):
    """Payload for creating memory records"""

    memories: list[MemoryRecord]


class GetSessionsQuery(BaseModel):
    """Query parameters for getting sessions"""

    limit: int = Field(default=20, ge=1, le=100)
    offset: int = Field(default=0, ge=0)
    namespace: str | None = None


class HealthCheckResponse(BaseModel):
    """Response for health check endpoint"""

    now: int


class SearchRequest(BaseModel):
    """Payload for long-term memory search"""

    text: str | None = Field(
        default=None,
        description="Optional text to use for a semantic search",
    )
    session_id: SessionId | None = Field(
        default=None,
        description="Optional session ID to filter by",
    )
```

```
    namespace: Namespace | None = Field(
        default=None,
        description="Optional namespace to filter by",
    )
    topics: Topics | None = Field(
        default=None,
        description="Optional topics to filter by",
    )
    entities: Entities | None = Field(
        default=None,
        description="Optional entities to filter by",
    )
    created_at: CreatedAt | None = Field(
        default=None,
        description="Optional created at timestamp to filter by",
    )
    last_accessed: LastAccessed | None = Field(
        default=None,
        description="Optional last accessed timestamp to filter by",
    )
    user_id: UserId | None = Field(
        default=None,
        description="Optional user ID to filter by",
    )
    distance_threshold: float | None = Field(
        default=None,
        description="Optional distance threshold to filter by",
    )
    memory_type: MemoryType | None = Field(
        default=None,
        description="Optional memory type to filter by",
    )
    event_date: EventDate | None = Field(
        default=None,
        description="Optional event date to filter by (for episodic memories)",
    )
    limit: int = Field(
        default=10,
        ge=1,
        le=100,
        description="Optional limit on the number of results",
    )
    offset: int = Field(
        default=0,
        ge=0,
        description="Optional offset",
    )

    def get_filters(self):
        """Get all filter objects as a dictionary"""
        filters = {}

        if self.session_id is not None:
            filters["session_id"] = self.session_id

        if self.namespace is not None:
            filters["namespace"] = self.namespace

        if self.topics is not None:
            filters["topics"] = self.topics

        if self.entities is not None:
            filters["entities"] = self.entities

        if self.user_id is not None:
            filters["user_id"] = self.user_id

        if self.created_at is not None:
            filters["created_at"] = self.created_at
```

```
        if self.last_accessed is not None:
            filters["last_accessed"] = self.last_accessed

        if self.memory_type is not None:
            filters["memory_type"] = self.memory_type

        if self.event_date is not None:
            filters["event_date"] = self.event_date

        return filters


class MemoryPromptRequest(BaseModel):
    query: str
    session: WorkingMemoryRequest | None = None
    long_term_search: SearchRequest | None = None


class SystemMessage(base.Message):
    """A system message"""

    role: Literal["system"] = "system"


class UserMessage(base.Message):
    """A user message"""

    role: Literal["user"] = "user"


class MemoryPromptResponse(BaseModel):
    messages: list[base.Message | SystemMessage]
```

== agent_memory_server/summarization.py ==

```
import logging

import tiktoken
from redis import WatchError

from agent_memory_server.config import settings
from agent_memory_server.llms import (
    AnthropicClientWrapper,
    OpenAIClientWrapper,
    get_model_client,
    get_model_config,
)
from agent_memory_server.models import MemoryMessage
from agent_memory_server.utils.keys import Keys
from agent_memory_server.utils.redis import get_redis_conn


logger = logging.getLogger(__name__)


async def _incremental_summary(
    model: str,
    client: OpenAIClientWrapper | AnthropicClientWrapper,
    context: str | None,
    messages: list[str],
) -> tuple[str, int]:
    """
    Incrementally summarize messages, building upon a previous summary.

    Args:
        model: The model to use (OpenAI or Anthropic)
        client: The client wrapper (OpenAI or Anthropic)
        context: Previous summary, if any
        messages: New messages to summarize

    Returns:
        Tuple of (summary, tokens_used)
    """
    # Reverse messages to put the most recent ones last
    messages.reverse()
    messages_joined = "\n".join(messages)
    prev_summary = context or ""

    # Prompt template for progressive summarization
    progressive_prompt = f"""
You are a precise summarization assistant. Your task is to progressively
summarize conversation history while maintaining critical context and accuracy.

INSTRUCTIONS:
1. Build upon the previous summary by incorporating new information chronologically
2. Preserve key details: names, technical terms, code references, and important decisions
3. Maintain the temporal sequence of events and discussions
4. For technical discussions, keep specific terms, versions, and implementation details
5. For code-related content, preserve function names, file paths, and important parameters
6. If the new content is irrelevant or doesn't add value, return "NONE"
7. Keep the summary concise but complete - aim for 2-3 sentences unless more detail is crucial
8. Use neutral, factual language

EXAMPLE
Current summary:
The user inquires about retirement investment options, specifically comparing
traditional IRAs and Roth IRAs. The assistant explains the key differences in
tax treatment, with traditional IRAs offering immediate tax deductions and Roth
IRAs providing tax-free withdrawals in retirement.

New lines of conversation:
Human: What factors should I consider when deciding between the two?
```

```
Assistant: Several key factors influence this decision: 1) Your current tax
bracket vs. expected retirement tax bracket, 2) Time horizon until retirement,
3) Current income and eligibility for Roth IRA contributions, and 4) Desire for
flexibility in retirement withdrawals. For example, if you expect to be in a
higher tax bracket during retirement, a Roth IRA might be more advantageous
since qualified withdrawals are tax-free. Additionally, Roth IRAs don't have
required minimum distributions (RMDs) during your lifetime, offering more
flexibility in estate planning.

New summary:
The discussion covers retirement investment options, comparing traditional and
Roth IRAs' tax implications, with traditional IRAs offering immediate deductions
and Roth IRAs providing tax-free withdrawals. The conversation expands to cover
decision factors including current vs. future tax brackets, retirement timeline,
income eligibility, and withdrawal flexibility, with specific emphasis on Roth
IRA advantages for those expecting higher retirement tax brackets and the
benefit of no required minimum distributions. END OF EXAMPLE

Current summary:
{prev_summary}

New lines of conversation:
{messages_joined}

New summary:
"""

    try:
        # Get completion from client
        response = await client.create_chat_completion(model, progressive_prompt)

        # Extract completion text
        completion = response.choices[0].message.content

        # Get token usage
        tokens_used = response.total_tokens

        logger.info(f"Summarization complete, using {tokens_used} tokens")
        return completion, tokens_used
    except Exception as e:
        logger.error(f"Error in incremental summarization: {e}")
        raise


async def summarize_session(
    session_id: str,
    model: str,
    window_size: int,
) -> None:
    """
    Summarize messages in a session when they exceed the window size.

    This function:
    1. Gets the oldest messages up to window size and current context
    2. Generates a new summary that includes these messages
    3. Removes older, summarized messages and updates the context

    Stop summarizing

    Args:
        session_id: The session ID
        model: The model to use
        window_size: Maximum number of messages to keep
        client: The client wrapper (OpenAI or Anthropic)
        redis_conn: Redis connection
    """
    print("Summarizing session")
    redis = await get_redis_conn()
    client = await get_model_client(settings.generation_model)
```

```
messages_key = Keys.messages_key(session_id)
metadata_key = Keys.metadata_key(session_id)

async with redis.pipeline(transaction=False) as pipe:
    await pipe.watch(messages_key, metadata_key)

    num_messages = await pipe.llen(messages_key)  # type: ignore
    print(f"<task> Number of messages: {num_messages}")
    if num_messages < window_size:
        logger.info(f"Not enough messages to summarize for session {session_id}")
        return

    messages_raw = await pipe.lrange(messages_key, 0, window_size - 1)  # type: ignore
    metadata = await pipe.hgetall(metadata_key)  # type: ignore
    pipe.multi()

    while True:
        try:
            messages = []
            for msg_raw in messages_raw:
                if isinstance(msg_raw, bytes):
                    msg_raw = msg_raw.decode("utf-8")
                msg_dict = json.loads(msg_raw)
                messages.append(MemoryMessage(**msg_dict))

            print("Messages: ", messages)

            model_config = get_model_config(model)
            max_tokens = model_config.max_tokens

            # Token allocation:
            # - For small context (<10k): use 12.5% (min 512)
            # - For medium context (10k-50k): use 10% (min 1024)
            # - For large context (>50k): use 5% (min 2048)
            if max_tokens < 10000:
                summary_max_tokens = max(512, max_tokens // 8)  # 12.5%
            elif max_tokens < 50000:
                summary_max_tokens = max(1024, max_tokens // 10)  # 10%
            else:
                summary_max_tokens = max(2048, max_tokens // 20)  # 5%

            # Scale buffer tokens with context size, but keep reasonable bounds
            buffer_tokens = min(max(230, max_tokens // 100), 1000)

            max_message_tokens = max_tokens - summary_max_tokens - buffer_tokens
            encoding = tiktoken.get_encoding("cl100k_base")
            total_tokens = 0
            messages_to_summarize = []

            for msg in messages:
                msg_str = f"{msg.role}: {msg.content}"
                msg_tokens = len(encoding.encode(msg_str))

                # TODO: Here, we take a partial message if a single message's
                # total size exceeds the buffer. Should this be configurable
                # behavior?
                if msg_tokens > max_message_tokens:
                    msg_str = msg_str[: max_message_tokens // 2]
                    msg_tokens = len(encoding.encode(msg_str))
                    total_tokens += msg_tokens

                if total_tokens + msg_tokens <= max_message_tokens:
                    total_tokens += msg_tokens
                    messages_to_summarize.append(msg_str)

            if not messages_to_summarize:
                logger.info(f"No messages to summarize for session {session_id}")
                return
```

```
            context = metadata.get("context", "")

            summary, summary_tokens_used = await _incremental_summary(
                model,
                client,
                context,
                messages_to_summarize,
            )
            total_tokens += summary_tokens_used

            metadata["context"] = summary
            metadata["tokens"] = str(total_tokens)

            pipe.hmset(metadata_key, mapping=metadata)
            print("Metadata: ", metadata_key, metadata)

            # Messages that were summarized
            num_summarized = len(messages_to_summarize)
            pipe.ltrim(messages_key, 0, num_summarized - 1)

            await pipe.execute()
            break
        except WatchError:
            continue

logger.info(f"Summarization complete for session {session_id}")
```

== agent_memory_server/test_config.py ==

```
import yaml

from agent_memory_server.config import Settings, load_yaml_settings


def test_defaults(monkeypatch):
    # Clear env vars
    monkeypatch.delenv("APP_CONFIG_FILE", raising=False)
    monkeypatch.delenv("redis_url", raising=False)
    # No YAML file
    monkeypatch.chdir(tempfile.gettempdir())
    settings = Settings()
    assert settings.redis_url == "redis://localhost:6379"
    assert settings.port == 8000
    assert settings.log_level == "INFO"


def test_yaml_loading(tmp_path, monkeypatch):
    config = {"redis_url": "redis://test:6379", "port": 1234, "log_level": "DEBUG"}
    yaml_path = tmp_path / "config.yaml"
    with open(yaml_path, "w") as f:
        yaml.dump(config, f)
    monkeypatch.setenv("APP_CONFIG_FILE", str(yaml_path))
    # Remove env var overrides
    monkeypatch.delenv("redis_url", raising=False)
    monkeypatch.delenv("port", raising=False)
    monkeypatch.delenv("log_level", raising=False)
    loaded = load_yaml_settings()
    settings = Settings(**loaded)
    assert settings.redis_url == "redis://test:6379"
    assert settings.port == 1234
    assert settings.log_level == "DEBUG"


def test_env_overrides_yaml(tmp_path, monkeypatch):
    config = {"redis_url": "redis://yaml:6379", "port": 1111}
    yaml_path = tmp_path / "config.yaml"
    with open(yaml_path, "w") as f:
        yaml.dump(config, f)
    monkeypatch.setenv("APP_CONFIG_FILE", str(yaml_path))
    monkeypatch.setenv("redis_url", "redis://env:6379")
    monkeypatch.setenv("port", "2222")
    loaded = load_yaml_settings()
    settings = Settings(**loaded)
    # Env vars should override YAML
    assert settings.redis_url == "redis://env:6379"
    assert settings.port == 2222  # Pydantic auto-casts


def test_custom_config_path(tmp_path, monkeypatch):
    config = {"redis_url": "redis://custom:6379"}
    custom_path = tmp_path / "custom.yaml"
    with open(custom_path, "w") as f:
        yaml.dump(config, f)
    monkeypatch.setenv("APP_CONFIG_FILE", str(custom_path))
    loaded = load_yaml_settings()
    settings = Settings(**loaded)
    assert settings.redis_url == "redis://custom:6379"
```

```
== agent_memory_server/working_memory.py ==
                                                                                                                """Wo

import json
import logging
import time
from datetime import UTC, datetime

from redis.asyncio import Redis

from agent_memory_server.models import MemoryMessage, MemoryRecord, WorkingMemory
from agent_memory_server.utils.keys import Keys
from agent_memory_server.utils.redis import get_redis_conn


logger = logging.getLogger(__name__)


def json_datetime_handler(obj):
    """JSON serializer for datetime objects."""
    if isinstance(obj, datetime):
        return obj.isoformat()
    raise TypeError(f"Object of type {type(obj)} is not JSON serializable")


async def get_working_memory(
    session_id: str,
    namespace: str | None = None,
    redis_client: Redis | None = None,
) -> WorkingMemory | None:
    """
    Get working memory for a session.

    Args:
        session_id: The session ID
        namespace: Optional namespace for the session
        redis_client: Optional Redis client

    Returns:
        WorkingMemory object or None if not found
    """
    if not redis_client:
        redis_client = await get_redis_conn()

    key = Keys.working_memory_key(session_id, namespace)

    try:
        data = await redis_client.get(key)
        if not data:
            return None

        # Parse the JSON data
        working_memory_data = json.loads(data)

        # Convert memory records back to MemoryRecord objects
        memories = []
        for memory_data in working_memory_data.get("memories", []):
            memory = MemoryRecord(**memory_data)
            memories.append(memory)

        # Convert messages back to MemoryMessage objects
        messages = []
        for message_data in working_memory_data.get("messages", []):
            message = MemoryMessage(**message_data)
            messages.append(message)

        return WorkingMemory(
            messages=messages,
            memories=memories,
```

```
            context=working_memory_data.get("context"),
            user_id=working_memory_data.get("user_id"),
            tokens=working_memory_data.get("tokens", 0),
            session_id=session_id,
            namespace=namespace,
            ttl_seconds=working_memory_data.get("ttl_seconds", 3600),
            data=working_memory_data.get("data") or {},
            last_accessed=datetime.fromtimestamp(
                working_memory_data.get("last_accessed", int(time.time())), UTC
            ),
            created_at=datetime.fromtimestamp(
                working_memory_data.get("created_at", int(time.time())), UTC
            ),
            updated_at=datetime.fromtimestamp(
                working_memory_data.get("updated_at", int(time.time())), UTC
            ),
        )

    except Exception as e:
        logger.error(f"Error getting working memory for session {session_id}: {e}")
        return None


async def set_working_memory(
    working_memory: WorkingMemory,
    redis_client: Redis | None = None,
) -> None:
    """
    Set working memory for a session with TTL.

    Args:
        working_memory: WorkingMemory object to store
        redis_client: Optional Redis client
    """
    if not redis_client:
        redis_client = await get_redis_conn()

    # Validate that all memories have id (Stage 3 requirement)
    for memory in working_memory.memories:
        if not memory.id:
            raise ValueError("All memory records in working memory must have an id")

    key = Keys.working_memory_key(working_memory.session_id, working_memory.namespace)

    # Update the updated_at timestamp
    working_memory.updated_at = datetime.now(UTC)

    # Convert to JSON-serializable format with timestamp conversion
    data = {
        "messages": [
            message.model_dump(mode="json") for message in working_memory.messages
        ],
        "memories": [
            memory.model_dump(mode="json") for memory in working_memory.memories
        ],
        "context": working_memory.context,
        "user_id": working_memory.user_id,
        "tokens": working_memory.tokens,
        "session_id": working_memory.session_id,
        "namespace": working_memory.namespace,
        "ttl_seconds": working_memory.ttl_seconds,
        "data": working_memory.data or {},
        "last_accessed": int(working_memory.last_accessed.timestamp()),
        "created_at": int(working_memory.created_at.timestamp()),
        "updated_at": int(working_memory.updated_at.timestamp()),
    }

    try:
        # Store with TTL
```

```
        await redis_client.setex(
            key,
            working_memory.ttl_seconds,
            json.dumps(
                data, default=json_datetime_handler
            ),  # Add custom handler for any remaining datetime objects
        )
        logger.info(
            f"Set working memory for session {working_memory.session_id} with TTL {working_memory.ttl_seconds}s"
        )

    except Exception as e:
        logger.error(
            f"Error setting working memory for session {working_memory.session_id}: {e}"
        )
        raise


async def delete_working_memory(
    session_id: str,
    namespace: str | None = None,
    redis_client: Redis | None = None,
) -> None:
    """
    Delete working memory for a session.

    Args:
        session_id: The session ID
        namespace: Optional namespace for the session
        redis_client: Optional Redis client
    """
    if not redis_client:
        redis_client = await get_redis_conn()

    key = Keys.working_memory_key(session_id, namespace)

    try:
        await redis_client.delete(key)
        logger.info(f"Deleted working memory for session {session_id}")

    except Exception as e:
        logger.error(f"Error deleting working memory for session {session_id}: {e}")
        raise
```

== agent_memory_server/utils/__init__.py ==

== agent_memory_server/utils/api_keys.py ==

```python
"""API

import logging
import os


logger = logging.getLogger(__name__)


def load_api_key(service: str) -> str | None:
    """Load an API key from the environment.

    Args:
        service: The service name, e.g., 'openai', 'anthropic'

    Returns:
        The API key if found, or None
    """
    env_var = f"{service.upper()}_API_KEY"
    api_key = os.environ.get(env_var)

    if not api_key:
        logger.warning(f"No API key found for {service} (${env_var})")
        return None

    return api_key
```

== agent_memory_server/utils/keys.py ==

```
"""Red

import logging

from agent_memory_server.config import settings


logger = logging.getLogger(__name__)


class Keys:
    """Redis key utilities."""

    @staticmethod
    def context_key(session_id: str, namespace: str | None = None) -> str:
        """Get the context key for a session."""
        return (
            f"context:{namespace}:{session_id}"
            if namespace
            else f"context:{session_id}"
        )

    @staticmethod
    def token_count_key(session_id: str, namespace: str | None = None) -> str:
        """Get the token count key for a session."""
        return (
            f"tokens:{namespace}:{session_id}" if namespace else f"tokens:{session_id}"
        )

    @staticmethod
    def messages_key(session_id: str, namespace: str | None = None) -> str:
        """Get the messages key for a session."""
        return (
            f"messages:{namespace}:{session_id}"
            if namespace
            else f"messages:{session_id}"
        )

    @staticmethod
    def sessions_key(namespace: str | None = None) -> str:
        """Get the sessions key for a namespace."""
        return f"sessions:{namespace}" if namespace else "sessions"

    @staticmethod
    def memory_key(id: str, namespace: str | None = None) -> str:
        """Get the memory key for an ID."""
        return f"memory:{namespace}:{id}" if namespace else f"memory:{id}"

    @staticmethod
    def metadata_key(session_id: str, namespace: str | None = None) -> str:
        """Get the metadata key for a session."""
        return (
            f"metadata:{namespace}:{session_id}"
            if namespace
            else f"metadata:{session_id}"
        )

    @staticmethod
    def working_memory_key(session_id: str, namespace: str | None = None) -> str:
        """Get the working memory key for a session."""
        return (
            f"working_memory:{namespace}:{session_id}"
            if namespace
            else f"working_memory:{session_id}"
        )

    @staticmethod
    def search_index_name() -> str:
```

```
"""Return the name of the search index."""
return settings.redisvl_index_name
```

```
== agent_memory_server/utils/redis.py ==
                                                                                                                                                """Re


import logging
from typing import Any

from redis.asyncio import Redis
from redisvl.index import AsyncSearchIndex
from redisvl.schema import IndexSchema

from agent_memory_server.config import settings


logger = logging.getLogger(__name__)
_redis_pool: Redis | None = None
_index: AsyncSearchIndex | None = None


async def get_redis_conn(url: str = settings.redis_url, **kwargs) -> Redis:
    """Get a Redis connection.

    Args:
        url: Redis connection URL, or None to use settings.redis_url
        **kwargs: Additional arguments to pass to Redis.from_url

    Returns:
        A Redis client instance
    """
    global _redis_pool

    # Always use the existing _redis_pool if it's not None, regardless of the URL parameter
    # This ensures that the patched _redis_pool from the test fixture is used
    if _redis_pool is None:
        _redis_pool = Redis.from_url(url, **kwargs)
    return _redis_pool


def get_search_index(
    redis: Redis,
    index_name: str = settings.redisvl_index_name,
    vector_dimensions: str = settings.redisvl_vector_dimensions,
    distance_metric: str = settings.redisvl_distance_metric,
) -> AsyncSearchIndex:
    global _index
    if _index is None:
        schema = {
            "index": {
                "name": index_name,
                "prefix": f"{index_name}:",
                "key_separator": ":",
                "storage_type": "hash",
            },
            "fields": [
                {"name": "text", "type": "text"},
                {"name": "memory_hash", "type": "tag"},
                {"name": "id_", "type": "tag"},
                {"name": "session_id", "type": "tag"},
                {"name": "user_id", "type": "tag"},
                {"name": "namespace", "type": "tag"},
                {"name": "topics", "type": "tag"},
                {"name": "entities", "type": "tag"},
                {"name": "created_at", "type": "numeric"},
                {"name": "last_accessed", "type": "numeric"},
                {"name": "memory_type", "type": "tag"},
                {"name": "discrete_memory_extracted", "type": "tag"},
                {"name": "id", "type": "tag"},
                {"name": "persisted_at", "type": "numeric"},
                {"name": "extracted_from", "type": "tag"},
                {"name": "event_date", "type": "numeric"},
```

```
                {
                    "name": "vector",
                    "type": "vector",
                    "attrs": {
                        "algorithm": "HNSW",
                        "dims": vector_dimensions,
                        "distance_metric": distance_metric,
                        "datatype": "float32",
                    },
                },
            ],
        }
        index_schema = IndexSchema.from_dict(schema)
        _index = AsyncSearchIndex(index_schema, redis_client=redis)
    return _index


async def ensure_search_index_exists(
    redis: Redis,
    index_name: str = settings.redisvl_index_name,
    vector_dimensions: str = settings.redisvl_vector_dimensions,
    distance_metric: str = settings.redisvl_distance_metric,
    overwrite: bool = False,
) -> None:
    """
    Ensure that the async search index exists, create it if it doesn't.
    Uses RedisVL's AsyncSearchIndex.

    Args:
        redis: A Redis client instance
        vector_dimensions: Dimensions of the embedding vectors
        distance_metric: Distance metric to use (default: COSINE)
        index_name: The name of the index
    """
    index = get_search_index(redis, index_name, vector_dimensions, distance_metric)
    if await index.exists():
        logger.info("Async search index already exists")
        if overwrite:
            logger.info("Overwriting existing index")
            await redis.execute_command("FT.DROPINDEX", index.name)
        else:
            return
    else:
        logger.info("Async search index doesn't exist, creating...")

    await index.create()

    logger.info(
        f"Created async search index with {vector_dimensions} dimensions and {distance_metric} metric"
    )


def safe_get(doc: Any, key: str, default: Any | None = None) -> Any:
    """Get a value from a Document, returning a default if the key is not present.

    Args:
        doc: Document or object to get a value from
        key: Key to get
        default: Default value to return if key is not found

    Returns:
        The value if found, or the default
    """
    if isinstance(doc, dict):
        return doc.get(key, default)
    try:
        return getattr(doc, key)
    except (AttributeError, KeyError):
        return default
```

== agent_memory_server/client/__init__.py ==

```
== agent_memory_server/client/api.py ==
                                                                                                                """
Redis Memory Server API Client

This module provides a client for the REST API of the Redis Memory Server.
"""

import contextlib
from typing import Any, Literal

import httpx
from pydantic import BaseModel
from ulid import ULID

from agent_memory_server.filters import (
    CreatedAt,
    Entities,
    LastAccessed,
    MemoryType,
    Namespace,
    SessionId,
    Topics,
    UserId,
)
from agent_memory_server.models import (
    AckResponse,
    CreateMemoryRecordRequest,
    HealthCheckResponse,
    MemoryPromptRequest,
    MemoryPromptResponse,
    MemoryRecord,
    MemoryRecordResults,
    SearchRequest,
    SessionListResponse,
    WorkingMemory,
    WorkingMemoryRequest,
    WorkingMemoryResponse,
)


# Model name literals for model-specific window sizes
ModelNameLiteral = Literal[
    "gpt-3.5-turbo",
    "gpt-3.5-turbo-16k",
    "gpt-4",
    "gpt-4-32k",
    "gpt-4o",
    "gpt-4o-mini",
    "o1",
    "o1-mini",
    "o3-mini",
    "text-embedding-ada-002",
    "text-embedding-3-small",
    "text-embedding-3-large",
    "claude-3-opus-20240229",
    "claude-3-sonnet-20240229",
    "claude-3-haiku-20240307",
    "claude-3-5-sonnet-20240620",
    "claude-3-7-sonnet-20250219",
    "claude-3-5-sonnet-20241022",
    "claude-3-5-haiku-20241022",
    "claude-3-7-sonnet-latest",
    "claude-3-5-sonnet-latest",
    "claude-3-5-haiku-latest",
    "claude-3-opus-latest",
]


class MemoryClientConfig(BaseModel):
```

```
    """Configuration for the Memory API Client"""

    base_url: str
    timeout: float = 30.0
    default_namespace: str | None = None


class MemoryAPIClient:
    """
    Client for the Redis Memory Server REST API.

    This client provides methods to interact with all server endpoints:
    - Health check
    - Session management (list, get, put, delete)
    - Long-term memory (create, search)
    """

    def __init__(self, config: MemoryClientConfig):
        """
        Initialize the Memory API Client.

        Args:
            config: MemoryClientConfig instance with server connection details
        """
        self.config = config
        self._client = httpx.AsyncClient(
            base_url=config.base_url,
            timeout=config.timeout,
        )

    async def close(self):
        """Close the underlying HTTP client."""
        await self._client.aclose()

    async def __aenter__(self):
        """Support using the client as an async context manager."""
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        """Close the client when exiting the context manager."""
        await self.close()

    async def health_check(self) -> HealthCheckResponse:
        """
        Check the health of the memory server.

        Returns:
            HealthCheckResponse with current server timestamp
        """
        response = await self._client.get("/health")
        response.raise_for_status()
        return HealthCheckResponse(**response.json())

    async def list_sessions(
        self, limit: int = 20, offset: int = 0, namespace: str | None = None
    ) -> SessionListResponse:
        """
        List available sessions with optional pagination and namespace filtering.

        Args:
            limit: Maximum number of sessions to return (default: 20)
            offset: Offset for pagination (default: 0)
            namespace: Optional namespace filter

        Returns:
            SessionListResponse containing session IDs and total count
        """
        params = {
            "limit": str(limit),
```

```
        "offset": str(offset),
    }
    if namespace is not None:
        params["namespace"] = namespace
    elif self.config.default_namespace is not None:
        params["namespace"] = self.config.default_namespace

    response = await self._client.get("/sessions/", params=params)
    response.raise_for_status()
    return SessionListResponse(**response.json())

async def get_session_memory(
    self,
    session_id: str,
    namespace: str | None = None,
    window_size: int | None = None,
    model_name: ModelNameLiteral | None = None,
    context_window_max: int | None = None,
) -> WorkingMemoryResponse:
    """
    Get memory for a session, including messages and context.

    Args:
        session_id: The session ID to retrieve memory for
        namespace: Optional namespace for the session
        window_size: Optional number of messages to include
        model_name: Optional model name to determine context window size
        context_window_max: Optional direct specification of context window tokens

    Returns:
        WorkingMemoryResponse containing messages, context and metadata

    Raises:
        httpx.HTTPStatusError: If the session is not found (404) or other errors
    """
    params = {}

    if namespace is not None:
        params["namespace"] = namespace
    elif self.config.default_namespace is not None:
        params["namespace"] = self.config.default_namespace

    if window_size is not None:
        params["window_size"] = window_size

    if model_name is not None:
        params["model_name"] = model_name

    if context_window_max is not None:
        params["context_window_max"] = context_window_max

    response = await self._client.get(
        f"/sessions/{session_id}/memory", params=params
    )
    response.raise_for_status()
    return WorkingMemoryResponse(**response.json())

async def put_session_memory(
    self, session_id: str, memory: WorkingMemory
) -> WorkingMemoryResponse:
    """
    Store session memory. Replaces existing session memory if it exists.

    Args:
        session_id: The session ID to store memory for
        memory: WorkingMemory object with messages and optional context

    Returns:
        WorkingMemoryResponse with the updated memory (potentially summarized if window size exceeded)
```

```
    """
    # If namespace not specified in memory but set in config, use config's namespace
    if memory.namespace is None and self.config.default_namespace is not None:
        memory.namespace = self.config.default_namespace

    response = await self._client.put(
        f"/sessions/{session_id}/memory",
        json=memory.model_dump(exclude_none=True, mode="json"),
    )
    response.raise_for_status()
    return WorkingMemoryResponse(**response.json())

async def delete_session_memory(
    self, session_id: str, namespace: str | None = None
) -> AckResponse:
    """
    Delete memory for a session.

    Args:
        session_id: The session ID to delete memory for
        namespace: Optional namespace for the session

    Returns:
        AckResponse indicating success
    """
    params = {}
    if namespace is not None:
        params["namespace"] = namespace
    elif self.config.default_namespace is not None:
        params["namespace"] = self.config.default_namespace

    response = await self._client.delete(
        f"/sessions/{session_id}/memory", params=params
    )
    response.raise_for_status()
    return AckResponse(**response.json())

async def set_working_memory_data(
    self,
    session_id: str,
    data: dict[str, Any],
    namespace: str | None = None,
    preserve_existing: bool = True,
) -> WorkingMemoryResponse:
    """
    Convenience method to set JSON data in working memory.

    This method allows you to easily store arbitrary JSON data in working memory
    without having to construct a full WorkingMemory object.

    Args:
        session_id: The session ID to set data for
        data: Dictionary of JSON data to store
        namespace: Optional namespace for the session
        preserve_existing: If True, preserve existing messages and memories (default: True)

    Returns:
        WorkingMemoryResponse with the updated memory

    Example:
        ```python
        # Store user preferences
        await client.set_working_memory_data(
            session_id="session123",
            data={
                "user_settings": {"theme": "dark", "language": "en"},
                "preferences": {"notifications": True}
            }
        )
```
```

```
        ```
    """
    # Get existing memory if preserving
    existing_memory = None
    if preserve_existing:
        with contextlib.suppress(Exception):
            existing_memory = await self.get_session_memory(
                session_id=session_id,
                namespace=namespace,
)

    # Create new working memory with the data
    working_memory = WorkingMemory(
        session_id=session_id,
        namespace=namespace or self.config.default_namespace,
        messages=existing_memory.messages if existing_memory else [],
        memories=existing_memory.memories if existing_memory else [],
        data=data,
        context=existing_memory.context if existing_memory else None,
        user_id=existing_memory.user_id if existing_memory else None,
)

    return await self.put_session_memory(session_id, working_memory)

async def add_memories_to_working_memory(
    self,
    session_id: str,
    memories: list[MemoryRecord],
    namespace: str | None = None,
    replace: bool = False,
) -> WorkingMemoryResponse:
    """
    Convenience method to add structured memories to working memory.

    This method allows you to easily add MemoryRecord objects to working memory
    without having to manually construct and manage the full WorkingMemory object.

    Args:
        session_id: The session ID to add memories to
        memories: List of MemoryRecord objects to add
        namespace: Optional namespace for the session
        replace: If True, replace all existing memories; if False, append to existing (default: False)

    Returns:
        WorkingMemoryResponse with the updated memory

    Example:
        ```python
        # Add a semantic memory
        await client.add_memories_to_working_memory(
            session_id="session123",
            memories=[
                MemoryRecord(
                    text="User prefers dark mode",
                    memory_type="semantic",
                    topics=["preferences", "ui"],
                    id="pref_dark_mode"
                )
            ]
        )
        ```
    """
    # Get existing memory
    existing_memory = None
    with contextlib.suppress(Exception):
        existing_memory = await self.get_session_memory(
            session_id=session_id,
            namespace=namespace,
)
```
```

```
    # Determine final memories list
    if replace or not existing_memory:
        final_memories = memories
    else:
        final_memories = existing_memory.memories + memories

    # Auto-generate IDs for memories that don't have them
    for memory in final_memories:
        if not memory.id:
            memory.id = str(ULID())

    # Create new working memory with the memories
    working_memory = WorkingMemory(
        session_id=session_id,
        namespace=namespace or self.config.default_namespace,
        messages=existing_memory.messages if existing_memory else [],
        memories=final_memories,
        data=existing_memory.data if existing_memory else {},
        context=existing_memory.context if existing_memory else None,
        user_id=existing_memory.user_id if existing_memory else None,
    )

    return await self.put_session_memory(session_id, working_memory)

async def create_long_term_memory(
    self, memories: list[MemoryRecord]
) -> AckResponse:
    """
    Create long-term memories for later retrieval.

    Args:
        memories: List of MemoryRecord objects to store

    Returns:
        AckResponse indicating success

    Raises:
        httpx.HTTPStatusError: If long-term memory is disabled (400) or other errors
    """
    # Apply default namespace if needed
    if self.config.default_namespace is not None:
        for memory in memories:
            if memory.namespace is None:
                memory.namespace = self.config.default_namespace

    payload = CreateMemoryRecordRequest(memories=memories)
    response = await self._client.post(
        "/long-term-memory", json=payload.model_dump(exclude_none=True, mode="json")
    )
    response.raise_for_status()
    return AckResponse(**response.json())

async def search_long_term_memory(
    self,
    text: str,
    session_id: SessionId | dict[str, Any] | None = None,
    namespace: Namespace | dict[str, Any] | None = None,
    topics: Topics | dict[str, Any] | None = None,
    entities: Entities | dict[str, Any] | None = None,
    created_at: CreatedAt | dict[str, Any] | None = None,
    last_accessed: LastAccessed | dict[str, Any] | None = None,
    user_id: UserId | dict[str, Any] | None = None,
    distance_threshold: float | None = None,
    memory_type: MemoryType | dict[str, Any] | None = None,
    limit: int = 10,
    offset: int = 0,
) -> MemoryRecordResults:
    """
```

```
    Search long-term memories using semantic search and filters.

    Args:
        text: Search query text for semantic similarity
        session_id: Optional session ID filter
        namespace: Optional namespace filter
        topics: Optional topics filter
        entities: Optional entities filter
        created_at: Optional creation date filter
        last_accessed: Optional last accessed date filter
        user_id: Optional user ID filter
        distance_threshold: Optional distance threshold for search results
        limit: Maximum number of results to return (default: 10)
        offset: Offset for pagination (default: 0)

    Returns:
        MemoryRecordResults with matching memories and metadata

    Raises:
        httpx.HTTPStatusError: If long-term memory is disabled (400) or other errors
    """
    # Convert dictionary filters to their proper filter objects if needed
    if isinstance(session_id, dict):
        session_id = SessionId(**session_id)
    if isinstance(namespace, dict):
        namespace = Namespace(**namespace)
    if isinstance(topics, dict):
        topics = Topics(**topics)
    if isinstance(entities, dict):
        entities = Entities(**entities)
    if isinstance(created_at, dict):
        created_at = CreatedAt(**created_at)
    if isinstance(last_accessed, dict):
        last_accessed = LastAccessed(**last_accessed)
    if isinstance(user_id, dict):
        user_id = UserId(**user_id)
    if isinstance(memory_type, dict):
        memory_type = MemoryType(**memory_type)

    # Apply default namespace if needed and no namespace filter specified
    if namespace is None and self.config.default_namespace is not None:
        namespace = Namespace(eq=self.config.default_namespace)

    payload = SearchRequest(
        text=text,
        session_id=session_id,
        namespace=namespace,
        topics=topics,
        entities=entities,
        created_at=created_at,
        last_accessed=last_accessed,
        user_id=user_id,
        distance_threshold=distance_threshold,
        memory_type=memory_type,
        limit=limit,
        offset=offset,
    )

    response = await self._client.post(
        "/long-term-memory/search",
        json=payload.model_dump(exclude_none=True, mode="json"),
    )
    response.raise_for_status()
    return MemoryRecordResults(**response.json())

async def search_memories(
    self,
    text: str,
    session_id: SessionId | dict[str, Any] | None = None,
```

```
    namespace: Namespace | dict[str, Any] | None = None,
    topics: Topics | dict[str, Any] | None = None,
    entities: Entities | dict[str, Any] | None = None,
    created_at: CreatedAt | dict[str, Any] | None = None,
    last_accessed: LastAccessed | dict[str, Any] | None = None,
    user_id: UserId | dict[str, Any] | None = None,
    distance_threshold: float | None = None,
    memory_type: MemoryType | dict[str, Any] | None = None,
    limit: int = 10,
    offset: int = 0,
) -> MemoryRecordResults:
    """
    Search across all memory types (working memory and long-term memory).

    This method searches both working memory (ephemeral, session-scoped) and
    long-term memory (persistent, indexed) to provide comprehensive results.

    For working memory:
    - Uses simple text matching
    - Searches across all sessions (unless session_id filter is provided)
    - Returns memories that haven't been promoted to long-term storage

    For long-term memory:
    - Uses semantic vector search
    - Includes promoted memories from working memory
    - Supports advanced filtering by topics, entities, etc.

    Args:
        text: Search query text for semantic similarity
        session_id: Optional session ID filter
        namespace: Optional namespace filter
        topics: Optional topics filter
        entities: Optional entities filter
        created_at: Optional creation date filter
        last_accessed: Optional last accessed date filter
        user_id: Optional user ID filter
        distance_threshold: Optional distance threshold for search results
        memory_type: Optional memory type filter
        limit: Maximum number of results to return (default: 10)
        offset: Offset for pagination (default: 0)

    Returns:
        MemoryRecordResults with matching memories from both memory types

    Raises:
        httpx.HTTPStatusError: If the request fails
    """
    # Convert dictionary filters to their proper filter objects if needed
    if isinstance(session_id, dict):
        session_id = SessionId(**session_id)
    if isinstance(namespace, dict):
        namespace = Namespace(**namespace)
    if isinstance(topics, dict):
        topics = Topics(**topics)
    if isinstance(entities, dict):
        entities = Entities(**entities)
    if isinstance(created_at, dict):
        created_at = CreatedAt(**created_at)
    if isinstance(last_accessed, dict):
        last_accessed = LastAccessed(**last_accessed)
    if isinstance(user_id, dict):
        user_id = UserId(**user_id)
    if isinstance(memory_type, dict):
        memory_type = MemoryType(**memory_type)

    # Apply default namespace if needed and no namespace filter specified
    if namespace is None and self.config.default_namespace is not None:
        namespace = Namespace(eq=self.config.default_namespace)
```

```
    payload = SearchRequest(
        text=text,
        session_id=session_id,
        namespace=namespace,
        topics=topics,
        entities=entities,
        created_at=created_at,
        last_accessed=last_accessed,
        user_id=user_id,
        distance_threshold=distance_threshold,
        memory_type=memory_type,
        limit=limit,
        offset=offset,
    )

    response = await self._client.post(
        "/memory/search",
        json=payload.model_dump(exclude_none=True, mode="json"),
    )
    response.raise_for_status()
    return MemoryRecordResults(**response.json())

async def memory_prompt(
    self,
    query: str,
    session_id: str | None = None,
    namespace: str | None = None,
    window_size: int | None = None,
    model_name: ModelNameLiteral | None = None,
    context_window_max: int | None = None,
    long_term_search: SearchRequest | None = None,
) -> MemoryPromptResponse:
    """
    Hydrate a user query with memory context and return a prompt
    ready to send to an LLM.

    This method can retrieve relevant session history and long-term memories
    to provide context for the query.

    Args:
        query: The user's query text
        session_id: Optional session ID to retrieve history from
        namespace: Optional namespace for session and long-term memories
        window_size: Optional number of messages to include from session history
        model_name: Optional model name to determine context window size
        context_window_max: Optional direct specification of context window max tokens
        long_term_search: Optional SearchRequest for specific long-term memory filtering

    Returns:
        MemoryPromptResponse containing a list of messages with context

    Raises:
        httpx.HTTPStatusError: If the request fails or if neither session_id nor long_term_search is provided
    """
    # Prepare the request payload
    session_params = None
    if session_id is not None:
        session_params = WorkingMemoryRequest(
            session_id=session_id,
            namespace=namespace or self.config.default_namespace,
            window_size=window_size or 12,  # Default from settings
            model_name=model_name,
            context_window_max=context_window_max,
        )

    # If no explicit long_term_search is provided but we have a query, create a basic one
    if long_term_search is None and query:
        # Use default namespace from config if none provided
        _namespace = None
```

```
        if namespace is not None:
            _namespace = Namespace(eq=namespace)
        elif self.config.default_namespace is not None:
            _namespace = Namespace(eq=self.config.default_namespace)

        long_term_search = SearchRequest(
            text=query,
            namespace=_namespace,
        )

    # Create the request payload
    payload = MemoryPromptRequest(
        query=query,
        session=session_params,
        long_term_search=long_term_search,
    )

    # Make the API call
    response = await self._client.post(
        "/memory-prompt", json=payload.model_dump(exclude_none=True, mode="json")
    )
    response.raise_for_status()
    data = response.json()
    return MemoryPromptResponse(**data)

async def hydrate_memory_prompt(
    self,
    query: str,
    session_id: SessionId | dict[str, Any] | None = None,
    namespace: Namespace | dict[str, Any] | None = None,
    topics: Topics | dict[str, Any] | None = None,
    entities: Entities | dict[str, Any] | None = None,
    created_at: CreatedAt | dict[str, Any] | None = None,
    last_accessed: LastAccessed | dict[str, Any] | None = None,
    user_id: UserId | dict[str, Any] | None = None,
    distance_threshold: float | None = None,
    memory_type: MemoryType | dict[str, Any] | None = None,
    limit: int = 10,
    offset: int = 0,
    window_size: int = 12,
    model_name: ModelNameLiteral | None = None,
    context_window_max: int | None = None,
) -> MemoryPromptResponse:
    """
    Hydrate a user query with relevant session history and long-term memories.

    This method enriches the user's query by retrieving:
    1. Context from the conversation session (if session_id is provided)
    2. Relevant long-term memories related to the query

    Args:
        query: The user's query text
        session_id: Optional filter for session ID
        namespace: Optional filter for namespace
        topics: Optional filter for topics in long-term memories
        entities: Optional filter for entities in long-term memories
        created_at: Optional filter for creation date
        last_accessed: Optional filter for last access date
        user_id: Optional filter for user ID
        distance_threshold: Optional distance threshold for semantic search
        memory_type: Optional filter for memory type
        limit: Maximum number of long-term memory results (default: 10)
        offset: Offset for pagination (default: 0)
        window_size: Number of messages to include from session history (default: 12)
        model_name: Optional model name to determine context window size
        context_window_max: Optional direct specification of context window max tokens

    Returns:
        MemoryPromptResponse containing a list of messages with context
```

```
Raises:
    httpx.HTTPStatusError: If the request fails
"""
# Convert dictionary filters to their proper filter objects if needed
if isinstance(session_id, dict):
    session_id = SessionId(**session_id)
if isinstance(namespace, dict):
    namespace = Namespace(**namespace)
if isinstance(topics, dict):
    topics = Topics(**topics)
if isinstance(entities, dict):
    entities = Entities(**entities)
if isinstance(created_at, dict):
    created_at = CreatedAt(**created_at)
if isinstance(last_accessed, dict):
    last_accessed = LastAccessed(**last_accessed)
if isinstance(user_id, dict):
    user_id = UserId(**user_id)
if isinstance(memory_type, dict):
    memory_type = MemoryType(**memory_type)

# Apply default namespace if needed and no namespace filter specified
if namespace is None and self.config.default_namespace is not None:
    namespace = Namespace(eq=self.config.default_namespace)

# Extract session_id value if it exists
session_params = None
_session_id = None
if session_id and hasattr(session_id, "eq") and session_id.eq:
    _session_id = session_id.eq

if _session_id:
    # Get namespace value if it exists
    _namespace = None
    if namespace and hasattr(namespace, "eq"):
        _namespace = namespace.eq
    elif self.config.default_namespace:
        _namespace = self.config.default_namespace

    session_params = WorkingMemoryRequest(
        session_id=_session_id,
        namespace=_namespace,
        window_size=window_size,
        model_name=model_name,
        context_window_max=context_window_max,
    )

# Create search request for long-term memory
search_payload = SearchRequest(
    text=query,
    session_id=session_id,
    namespace=namespace,
    topics=topics,
    entities=entities,
    created_at=created_at,
    last_accessed=last_accessed,
    user_id=user_id,
    distance_threshold=distance_threshold,
    memory_type=memory_type,
    limit=limit,
    offset=offset,
)

# Create the request payload
payload = MemoryPromptRequest(
    query=query,
    session=session_params,
    long_term_search=search_payload,
```

```
        )

        # Make the API call
        response = await self._client.post(
            "/memory-prompt", json=payload.model_dump(exclude_none=True, mode="json")
        )
        response.raise_for_status()
        data = response.json()
        return MemoryPromptResponse(**data)


# Helper function to create a memory client
async def create_memory_client(
    base_url: str, timeout: float = 30.0, default_namespace: str | None = None
) -> MemoryAPIClient:
    """
    Create and initialize a Memory API Client.

    Args:
        base_url: Base URL of the memory server (e.g., 'http://localhost:8000')
        timeout: Request timeout in seconds (default: 30.0)
        default_namespace: Optional default namespace to use for operations

    Returns:
        Initialized MemoryAPIClient instance
    """
    config = MemoryClientConfig(
        base_url=base_url,
        timeout=timeout,
        default_namespace=default_namespace,
    )
    client = MemoryAPIClient(config)

    # Test connection with a health check
    try:
        await client.health_check()
    except Exception as e:
        await client.close()
        raise ConnectionError(
            f"Failed to connect to memory server at {base_url}: {e}"
        ) from e

    return client
```

== tests/__init__.py ==

This

== tests/conftest.py ==

```
import contextlib
import os
import time
from unittest import mock
from unittest.mock import AsyncMock, patch

import docket
import pytest
from dotenv import load_dotenv
from fastapi import FastAPI
from httpx import ASGITransport, AsyncClient
from redis import Redis
from redis.asyncio import Redis as AsyncRedis
from testcontainers.compose import DockerCompose

from agent_memory_server.api import router as memory_router
from agent_memory_server.config import settings
from agent_memory_server.dependencies import DocketBackgroundTasks, get_background_tasks
from agent_memory_server.healthcheck import router as health_router
from agent_memory_server.llms import OpenAIClientWrapper
from agent_memory_server.messages import (
    MemoryMessage,
)

# Import the module to access its global for resetting
from agent_memory_server.utils import redis as redis_utils_module
from agent_memory_server.utils.keys import Keys
from agent_memory_server.utils.redis import ensure_search_index_exists


load_dotenv()


@pytest.fixture(scope="session")
def event_loop(request):
    return asyncio.get_event_loop()


@pytest.fixture()
def memory_message():
    """Create a sample memory message"""
    return MemoryMessage(role="user", content="Hello, world!")


@pytest.fixture()
def memory_messages():
    """Create a list of sample memory messages"""
    return [
        MemoryMessage(role="user", content="What is the capital of France?"),
        MemoryMessage(role="assistant", content="The capital of France is Paris."),
        MemoryMessage(role="user", content="And what about Germany?"),
        MemoryMessage(role="assistant", content="The capital of Germany is Berlin."),
    ]


@pytest.fixture()
def mock_openai_client():
    """Create a mock OpenAI client"""
    return AsyncMock(spec=OpenAIClientWrapper)

    # We won't set default side effects here, allowing tests to set their own mocks
    # This prevents conflicts with tests that need specific return values


@pytest.fixture(autouse=True)
async def search_index(async_redis_client):
    """Create a Redis connection pool for testing"""
```

```
    # Reset the cached index in redis_utils_module
    redis_utils_module._index = None

    await async_redis_client.flushdb()

    try:
        try:
            await async_redis_client.execute_command(
                "FT.INFO", settings.redisvl_index_name
            )
            await async_redis_client.execute_command(
                "FT.DROPINDEX", settings.redisvl_index_name
            )
        except Exception as e:
            if "unknown index name".lower() not in str(e).lower():
                pass

        await ensure_search_index_exists(async_redis_client)

    except Exception:
        raise

    yield

    # Clean up after tests
    await async_redis_client.flushdb()
    with contextlib.suppress(Exception):
        await async_redis_client.execute_command(
            "FT.DROPINDEX", settings.redisvl_index_name
        )


@pytest.fixture()
async def session(use_test_redis_connection, async_redis_client):
    """Set up a test session with Redis data for testing"""
    import logging

    logging.getLogger(__name__)

    try:
        session_id = "test-session"
        namespace = "test-namespace"

        # Create working memory data
        from agent_memory_server.models import MemoryMessage, WorkingMemory

        messages = [
            MemoryMessage(role="user", content="Hello"),
            MemoryMessage(role="assistant", content="Hi there"),
        ]

        working_memory = WorkingMemory(
            messages=messages,
            memories=[],  # No structured memories for this test
            context="Sample context",
            user_id="test-user",
            tokens=150,
            session_id=session_id,
            namespace=namespace,
        )

        # Store in unified working memory format
        from agent_memory_server.working_memory import set_working_memory

        await set_working_memory(
            working_memory=working_memory,
            redis_client=use_test_redis_connection,
        )
```

```
# Also add session to sessions list for compatibility
sessions_key = Keys.sessions_key(namespace=namespace)
current_time = int(time.time())
await use_test_redis_connection.zadd(sessions_key, {session_id: current_time})

# Index the messages as long-term memories directly without background tasks
import ulid
from redisvl.utils.vectorize import OpenAITextVectorizer

from agent_memory_server.models import MemoryRecord

# Create MemoryRecord objects for each message
long_term_memories = []
for msg in messages:
    memory = MemoryRecord(
        text=f"{msg.role}: {msg.content}",
        session_id=session_id,
        namespace=namespace,
        user_id="test-user",
    )
    long_term_memories.append(memory)

# Index the memories directly
vectorizer = OpenAITextVectorizer()
embeddings = await vectorizer.aembed_many(
    [memory.text for memory in long_term_memories],
    batch_size=20,
    as_buffer=True,
)

async with use_test_redis_connection.pipeline(transaction=False) as pipe:
    for idx, vector in enumerate(embeddings):
        memory = long_term_memories[idx]
        id_ = memory.id_ if memory.id_ else str(ulid.ULID())
        key = Keys.memory_key(id_, memory.namespace)

        # Generate memory hash for the memory
        from agent_memory_server.long_term_memory import (
            generate_memory_hash,
        )

        memory_hash = generate_memory_hash(
            {
                "text": memory.text,
                "user_id": memory.user_id or "",
                "session_id": memory.session_id or "",
            }
        )

        await pipe.hset(  # type: ignore
            key,
            mapping={
                "text": memory.text,
                "id_": id_,
                "session_id": memory.session_id or "",
                "user_id": memory.user_id or "",
                "last_accessed": int(memory.last_accessed.timestamp())
                if memory.last_accessed
                else int(time.time()),
                "created_at": int(memory.created_at.timestamp())
                if memory.created_at
                else int(time.time()),
                "namespace": memory.namespace or "",
                "memory_hash": memory_hash,
                "vector": vector,
                "topics": "",
                "entities": "",
            },
        )
```

```
            await pipe.execute()

        return session_id
    except Exception:
        raise


@pytest.fixture(scope="session", autouse=True)
def redis_container(request):
    """
    If using xdist, create a unique Compose project for each xdist worker by
    setting COMPOSE_PROJECT_NAME. That prevents collisions on container/volume
    names.
    """
    # In xdist, the config has "workerid" in workerinput
    workerinput = getattr(request.config, "workerinput", {})
    worker_id = workerinput.get("workerid", "master")

    # Set the Compose project name so containers do not clash across workers
    os.environ["COMPOSE_PROJECT_NAME"] = f"redis_test_{worker_id}"
    os.environ.setdefault("REDIS_IMAGE", "redis/redis-stack-server:latest")

    compose = DockerCompose(
        context="tests",
        compose_file_name="docker-compose.yml",
        pull=True,
    )
    compose.start()

    yield compose

    compose.stop()


@pytest.fixture(scope="session")
def redis_url(redis_container):
    """
    Use the `DockerCompose` fixture to get host/port of the 'redis' service
    on container port 6379 (mapped to an ephemeral port on the host).
    """
    host, port = redis_container.get_service_host_and_port("redis", 6379)
    return f"redis://{host}:{port}"


@pytest.fixture()
def async_redis_client(redis_url):
    """
    An async Redis client that uses the dynamic `redis_url`.
    """
    return AsyncRedis.from_url(redis_url)


@pytest.fixture()
def mock_async_redis_client():
    """Create a mock async Redis client"""
    return AsyncMock(spec=AsyncRedis)


@pytest.fixture()
def redis_client(redis_url):
    """
    A sync Redis client that uses the dynamic `redis_url`.
    """
    return Redis.from_url(redis_url)


@pytest.fixture()
def use_test_redis_connection(redis_url: str):
```

```
    """Replace the Redis connection with a test one"""
    replacement_redis = AsyncRedis.from_url(redis_url)

    # Create a mock get_redis_conn function that always returns the replacement_redis
    async def mock_get_redis_conn(*args, **kwargs):
        # Ignore any URL parameter and always return the replacement_redis
        return replacement_redis

    # Create a patched Docket class that uses the test Redis URL
    original_docket_init = docket.Docket.__init__

    def patched_docket_init(self, name, url=None, *args, **kwargs):
        # Use the test Redis URL instead of the default one
        return original_docket_init(self, name, *args, url=redis_url, **kwargs)

    with (
        patch("agent_memory_server.utils.redis.get_redis_conn", mock_get_redis_conn),
        patch("agent_memory_server.utils.redis.get_redis_conn", mock_get_redis_conn),
        patch("docket.docket.Docket.__init__", patched_docket_init),
    ):
        yield replacement_redis


def pytest_addoption(parser: pytest.Parser) -> None:
    parser.addoption(
        "--run-api-tests",
        action="store_true",
        default=False,
        help="Run tests that require API keys",
    )


def pytest_configure(config: pytest.Config) -> None:
    config.addinivalue_line(
        "markers", "requires_api_keys: mark test as requiring API keys"
    )


def pytest_collection_modifyitems(
    config: pytest.Config, items: list[pytest.Item]
) -> None:
    if config.getoption("--run-api-tests"):
        return

    # Otherwise skip all tests requiring an API key
    skip_api = pytest.mark.skip(
        reason="""
        Skipping test because API keys are not provided.
        "Use --run-api-tests to run these tests.
        """
    )
    for item in items:
        if item.get_closest_marker("requires_api_keys"):
            item.add_marker(skip_api)


@pytest.fixture()
def mock_background_tasks():
    """Create a mock DocketBackgroundTasks instance"""
    return mock.Mock(name="DocketBackgroundTasks", spec=DocketBackgroundTasks)


@pytest.fixture(autouse=True)
def setup_redis_pool(use_test_redis_connection):
    """Set up the global Redis pool for all tests"""
    # Set the global _redis_pool variable to ensure that direct calls to get_redis_conn work
    import agent_memory_server.utils.redis

    agent_memory_server.utils.redis._redis_pool = use_test_redis_connection
```

```
    yield

    # Reset the global _redis_pool variable after the test
    agent_memory_server.utils.redis._redis_pool = None


@pytest.fixture()
def app(use_test_redis_connection):
    """Create a test FastAPI app with routers"""
    app = FastAPI()

    # Include routers
    app.include_router(health_router)
    app.include_router(memory_router)

    # Override the get_redis_conn function to return the test Redis connection
    async def mock_get_redis_conn(*args, **kwargs):
        return use_test_redis_connection

    # Override the dependency
    from agent_memory_server.utils.redis import get_redis_conn

    app.dependency_overrides[get_redis_conn] = mock_get_redis_conn

    return app


@pytest.fixture()
def app_with_mock_background_tasks(use_test_redis_connection, mock_background_tasks):
    """Create a test FastAPI app with routers"""
    app = FastAPI()

    # Include routers
    app.include_router(health_router)
    app.include_router(memory_router)

    # Override the get_redis_conn function to return the test Redis connection
    async def mock_get_redis_conn(*args, **kwargs):
        return use_test_redis_connection

    # Override the dependencies
    from agent_memory_server.utils.redis import get_redis_conn

    app.dependency_overrides[get_redis_conn] = mock_get_redis_conn
    app.dependency_overrides[get_background_tasks] = lambda: mock_background_tasks

    return app


@pytest.fixture()
async def client(app):
    async with AsyncClient(
        transport=ASGITransport(app=app),
        base_url="http://test",
    ) as client:
        yield client


@pytest.fixture()
async def client_with_mock_background_tasks(app_with_mock_background_tasks):
    async with AsyncClient(
        transport=ASGITransport(app=app_with_mock_background_tasks),
        base_url="http://test",
    ) as client:
        yield client
```

== tests/test_api.py ==

```
from unittest.mock import AsyncMock, MagicMock, patch

import numpy as np
import pytest

from agent_memory_server.config import Settings
from agent_memory_server.long_term_memory import (
    index_long_term_memories,
    promote_working_memory_to_long_term,
)
from agent_memory_server.models import (
    MemoryMessage,
    MemoryRecordResult,
    MemoryRecordResultsResponse,
    SessionListResponse,
    WorkingMemory,
    WorkingMemoryResponse,
)


@pytest.fixture
def mock_openai_client_wrapper():
    """Create a mock OpenAIClientWrapper that doesn't need an API key"""
    with patch("agent_memory_server.models.OpenAIClientWrapper") as mock_wrapper:
        # Create a mock instance
        mock_instance = AsyncMock()
        mock_wrapper.return_value = mock_instance

        # Mock the create_embedding and create_chat_completion methods
        mock_instance.create_embedding.return_value = np.array(
            [[0.1] * 1536], dtype=np.float32
        )
        mock_instance.create_chat_completion.return_value = {
            "choices": [{"message": {"content": "Test response"}}],
            "usage": {"total_tokens": 100},
        }

        yield mock_wrapper


class TestHealthEndpoint:
    @pytest.mark.asyncio
    async def test_health_endpoint(self, client):
        """Test the health endpoint"""
        response = await client.get("/health")

        assert response.status_code == 200

        data = response.json()
        assert "now" in data
        assert isinstance(data["now"], int)


class TestMemoryEndpoints:
    async def test_list_sessions_empty(self, client):
        """Test the list_sessions endpoint with no sessions"""
        response = await client.get("/sessions/?offset=0&limit=10")

        assert response.status_code == 200

        data = response.json()
        response = SessionListResponse(**data)
        assert response.sessions == []
        assert response.total == 0

    async def test_list_sessions_with_sessions(self, client, session):
        """Test the list_sessions endpoint with a session"""
```

```
    response = await client.get(
        "/sessions/?offset=0&limit=10&namespace=test-namespace"
    )
    assert response.status_code == 200

    data = response.json()
    response = SessionListResponse(**data)
    assert response.sessions == [session]
    assert response.total == 1

async def test_get_memory(self, client, session):
    """Test the get_memory endpoint"""
    session_id = session

    response = await client.get(
        f"/sessions/{session_id}/memory?namespace=test-namespace"
    )

    assert response.status_code == 200

    data = response.json()
    response = WorkingMemoryResponse(**data)
    assert response.messages == [
        MemoryMessage(role="user", content="Hello"),
        MemoryMessage(role="assistant", content="Hi there"),
    ]

    roles = [msg["role"] for msg in data["messages"]]
    contents = [msg["content"] for msg in data["messages"]]
    assert "user" in roles
    assert "assistant" in roles
    assert "Hello" in contents
    assert "Hi there" in contents

    # Check context and tokens
    assert data["context"] == "Sample context"
    assert int(data["tokens"]) == 150  # Convert string to int for comparison

@pytest.mark.requires_api_keys
@pytest.mark.asyncio
async def test_put_memory(self, client):
    """Test the post_memory endpoint"""
    payload = {
        "messages": [
            {"role": "user", "content": "Hello"},
            {"role": "assistant", "content": "Hi there"},
        ],
        "memories": [],
        "context": "Previous context",
        "namespace": "test-namespace",
        "session_id": "test-session",
    }

    response = await client.put("/sessions/test-session/memory", json=payload)

    assert response.status_code == 200

    data = response.json()
    # Should return the working memory, not just a status
    assert "messages" in data
    assert "context" in data
    assert "namespace" in data
    assert data["context"] == "Previous context"
    assert len(data["messages"]) == 2
    assert data["messages"][0]["role"] == "user"
    assert data["messages"][0]["content"] == "Hello"
    assert data["messages"][1]["role"] == "assistant"
    assert data["messages"][1]["content"] == "Hi there"
```

```
    # Verify we can still retrieve the session memory
    updated_session = await client.get(
        "/sessions/test-session/memory?namespace=test-namespace"
    )
    assert updated_session.status_code == 200
    assert updated_session.json()["messages"] == payload["messages"]

@pytest.mark.requires_api_keys
@pytest.mark.asyncio
async def test_put_memory_stores_messages_in_long_term_memory(
    self, client_with_mock_background_tasks, mock_background_tasks
):
    """Test the put_memory endpoint"""
    client = client_with_mock_background_tasks
    payload = {
        "messages": [
            {"role": "user", "content": "Hello"},
            {"role": "assistant", "content": "Hi there"},
        ],
        "memories": [],
        "context": "Previous context",
        "namespace": "test-namespace",
        "session_id": "test-session",
    }
    mock_settings = Settings(long_term_memory=True)

    with patch("agent_memory_server.api.settings", mock_settings):
        response = await client.put("/sessions/test-session/memory", json=payload)

    assert response.status_code == 200

    data = response.json()
    # Should return the working memory, not just a status
    assert "messages" in data
    assert "context" in data
    assert data["context"] == "Previous context"

    # Check that background tasks were called
    assert mock_background_tasks.add_task.call_count == 1

    # Check that the last call was for long-term memory indexing
    assert (
        mock_background_tasks.add_task.call_args_list[-1][0][0]
        == index_long_term_memories
    )

@pytest.mark.requires_api_keys
@pytest.mark.asyncio
async def test_put_memory_with_structured_memories_triggers_promotion(
    self, client_with_mock_background_tasks, mock_background_tasks
):
    """Test that structured memories trigger background promotion task"""
    client = client_with_mock_background_tasks
    payload = {
        "messages": [],
        "memories": [
            {
                "text": "User prefers dark mode",
                "id": "test-memory-1",
                "memory_type": "semantic",
                "namespace": "test-namespace",
            }
        ],
        "context": "Previous context",
        "namespace": "test-namespace",
        "session_id": "test-session",
    }
    mock_settings = Settings(long_term_memory=True)
```

```
    with patch("agent_memory_server.api.settings", mock_settings):
        response = await client.put("/sessions/test-session/memory", json=payload)

    assert response.status_code == 200

    data = response.json()
    assert "memories" in data
    assert len(data["memories"]) == 1
    assert data["memories"][0]["text"] == "User prefers dark mode"

    # Check that promotion background task was called
    assert mock_background_tasks.add_task.call_count == 1

    # Check that it was the promotion task, not indexing
    assert (
        mock_background_tasks.add_task.call_args_list[0][0][0]
        == promote_working_memory_to_long_term
    )

    # Check the arguments passed to the promotion task
    task_args = mock_background_tasks.add_task.call_args_list[0][0]
    assert task_args[1] == "test-session"  # session_id
    assert task_args[2] == "test-namespace"  # namespace

@pytest.mark.requires_api_keys
@pytest.mark.asyncio
async def test_post_memory_compacts_long_conversation(
    self, client_with_mock_background_tasks, mock_background_tasks
):
    """Test the post_memory endpoint with window size exceeded"""
    client = client_with_mock_background_tasks
    payload = {
        "messages": [
            {"role": "user", "content": "Hello"},
            {"role": "assistant", "content": "Hi there"},
        ],
        "memories": [],
        "context": "Previous context",
        "namespace": "test-namespace",
        "session_id": "test-session",
    }
    mock_settings = Settings(window_size=1, long_term_memory=False)

    with (
        patch("agent_memory_server.api.settings", mock_settings),
        patch(
            "agent_memory_server.api._summarize_working_memory"
        ) as mock_summarize,
    ):
        # Mock the summarization to return the working memory with updated context
        mock_summarized_memory = WorkingMemory(
            messages=[
                MemoryMessage(role="assistant", content="Hi there")
            ],  # Only keep last message
            memories=[],
            context="Summary: User greeted and assistant responded.",
            session_id="test-session",
            namespace="test-namespace",
        )
        mock_summarize.return_value = mock_summarized_memory

        response = await client.put("/sessions/test-session/memory", json=payload)

    assert response.status_code == 200

    data = response.json()
    # Should return the summarized working memory
    assert "messages" in data
    assert "context" in data
```

```
        # Should have been summarized (only 1 message kept due to window_size=1)
        assert len(data["messages"]) == 1
        assert data["messages"][0]["content"] == "Hi there"
        assert "Summary:" in data["context"]

        # Verify summarization was called
        mock_summarize.assert_called_once()

    @pytest.mark.asyncio
    async def test_delete_memory(self, client, session):
        """Test the delete_memory endpoint"""
        session_id = session

        response = await client.get(
            f"/sessions/{session_id}/memory?namespace=test-namespace"
        )

        assert response.status_code == 200

        data = response.json()
        assert len(data["messages"]) == 2

        response = await client.delete(
            f"/sessions/{session_id}/memory?namespace=test-namespace"
        )

        assert response.status_code == 200

        data = response.json()
        assert "status" in data
        assert data["status"] == "ok"

        response = await client.get(
            f"/sessions/{session_id}/memory?namespace=test-namespace"
        )
        assert response.status_code == 200

        # Should return empty working memory after deletion
        data = response.json()
        assert len(data["messages"]) == 0


@pytest.mark.requires_api_keys
class TestSearchEndpoint:
    @patch("agent_memory_server.api.long_term_memory.search_long_term_memories")
    @pytest.mark.asyncio
    async def test_search(self, mock_search, client):
        """Test the search endpoint"""
        mock_search.return_value = MemoryRecordResultsResponse(
            total=2,
            memories=[
                MemoryRecordResult(id_="1", text="User: Hello, world!", dist=0.25),
                MemoryRecordResult(id_="2", text="Assistant: Hi there!", dist=0.75),
            ],
            next_offset=None,
        )

        # Create payload
        payload = {"text": "What is the capital of France?"}

        # Call endpoint with the correct URL format (matching the router definition)
        response = await client.post("/long-term-memory/search", json=payload)

        # Check status code
        assert response.status_code == 200, response.text

        # Check response structure
        data = response.json()
        assert "memories" in data
```

```
        assert "total" in data
        assert data["total"] == 2
        assert len(data["memories"]) == 2

        # Check first result
        assert data["memories"][0]["id_"] == "1"
        assert data["memories"][0]["text"] == "User: Hello, world!"
        assert data["memories"][0]["dist"] == 0.25

        # Check second result
        assert data["memories"][1]["id_"] == "2"
        assert data["memories"][1]["text"] == "Assistant: Hi there!"
        assert data["memories"][1]["dist"] == 0.75


@pytest.mark.requires_api_keys
class TestMemoryPromptEndpoint:
    @patch("agent_memory_server.api.working_memory.get_working_memory")
    @pytest.mark.asyncio
    async def test_memory_prompt_with_session_id(self, mock_get_working_memory, client):
        """Test the memory_prompt endpoint with only session_id provided"""
        # Mock the session memory
        mock_session_memory = WorkingMemoryResponse(
            messages=[
                MemoryMessage(role="user", content="Hello"),
                MemoryMessage(role="assistant", content="Hi there"),
            ],
            memories=[],
            session_id="test-session",
            context="Previous conversation context",
            tokens=150,
        )
        mock_get_working_memory.return_value = mock_session_memory

        # Call the endpoint
        query = "What's the weather like?"
        response = await client.post(
            "/memory-prompt",
            json={
                "query": query,
                "session": {
                    "session_id": "test-session",
                    "namespace": "test-namespace",
                    "window_size": 10,
                    "model_name": "gpt-4o",
                    "context_window_max": 1000,
                },
            },
        )

        # Check status code
        assert response.status_code == 200

        # Check response data
        data = response.json()
        assert isinstance(data, dict)
        assert (
            len(data["messages"]) == 4
        )  # Context message + 2 session messages + query

        # Verify the messages content
        assert data["messages"][0]["role"] == "system"
        assert "Previous conversation context" in data["messages"][0]["content"]["text"]
        assert data["messages"][1]["role"] == "user"
        assert data["messages"][1]["content"]["text"] == "Hello"
        assert data["messages"][2]["role"] == "assistant"
        assert data["messages"][2]["content"]["text"] == "Hi there"
        assert data["messages"][3]["role"] == "user"
        assert data["messages"][3]["content"]["text"] == query
```

```
@patch("agent_memory_server.api.long_term_memory.search_long_term_memories")
@pytest.mark.asyncio
async def test_memory_prompt_with_long_term_memory(self, mock_search, client):
    """Test the memory_prompt endpoint with only long_term_search_payload provided"""
    # Mock the long-term memory search
    mock_search.return_value = MemoryRecordResultsResponse(
        total=2,
        memories=[
            MemoryRecordResult(id_="1", text="User likes coffee", dist=0.25),
            MemoryRecordResult(
                id_="2", text="User is allergic to peanuts", dist=0.35
            ),
        ],
        next_offset=None,
    )

    # Prepare the payload
    payload = {
        "query": "What should I eat?",
        "long_term_search": {
            "text": "food preferences allergies",
        },
    }

    # Call the endpoint
    response = await client.post("/memory-prompt", json=payload)

    # Check status code
    assert response.status_code == 200

    # Check response data
    data = response.json()
    assert isinstance(data, dict)
    assert len(data["messages"]) == 2  # Long-term memory message + query

    # Verify the messages content
    assert data["messages"][0]["role"] == "system"
    assert "Long term memories" in data["messages"][0]["content"]["text"]
    assert "User likes coffee" in data["messages"][0]["content"]["text"]
    assert "User is allergic to peanuts" in data["messages"][0]["content"]["text"]
    assert data["messages"][1]["role"] == "user"
    assert data["messages"][1]["content"]["text"] == "What should I eat?"

@patch("agent_memory_server.api.working_memory.get_working_memory")
@patch("agent_memory_server.api.long_term_memory.search_long_term_memories")
@pytest.mark.asyncio
async def test_memory_prompt_with_both_sources(
    self, mock_search, mock_get_working_memory, client
):
    """Test the memory_prompt endpoint with both session_id and long_term_search_payload"""
    # Mock session memory
    mock_session_memory = WorkingMemoryResponse(
        messages=[
            MemoryMessage(role="user", content="How do you make pasta?"),
            MemoryMessage(
                role="assistant",
                content="Boil water, add pasta, cook until al dente.",
            ),
        ],
        memories=[],
        session_id="test-session",
        context="Cooking conversation",
        tokens=200,
    )
    mock_get_working_memory.return_value = mock_session_memory

    # Mock the long-term memory search
    mock_search.return_value = MemoryRecordResultsResponse(
```

```
        total=1,
        memories=[
            MemoryRecordResult(
                id_="1", text="User prefers gluten-free pasta", dist=0.3
            ),
        ],
        next_offset=None,
    )

    # Prepare the payload
    payload = {
        "query": "What pasta should I buy?",
        "session": {
            "session_id": "test-session",
            "namespace": "test-namespace",
        },
        "long_term_search": {
            "text": "pasta preferences",
        },
    }

    # Call the endpoint
    response = await client.post("/memory-prompt", json=payload)

    # Check status code
    assert response.status_code == 200

    # Check response data
    data = response.json()
    assert isinstance(data, dict)
    assert (
        len(data["messages"]) == 5
    )  # Context + 2 session messages + long-term memory + query

    # Verify the messages content (order matters)
    assert data["messages"][0]["role"] == "system"
    assert "Cooking conversation" in data["messages"][0]["content"]["text"]
    assert data["messages"][1]["role"] == "user"
    assert data["messages"][1]["content"]["text"] == "How do you make pasta?"
    assert data["messages"][2]["role"] == "assistant"
    assert (
        data["messages"][2]["content"]["text"]
        == "Boil water, add pasta, cook until al dente."
    )
    assert data["messages"][3]["role"] == "system"
    assert "Long term memories" in data["messages"][3]["content"]["text"]
    assert (
        "User prefers gluten-free pasta" in data["messages"][3]["content"]["text"]
    )
    assert data["messages"][4]["role"] == "user"
    assert data["messages"][4]["content"]["text"] == "What pasta should I buy?"

@pytest.mark.asyncio
async def test_memory_prompt_without_required_params(self, client):
    """Test the memory_prompt endpoint without required parameters"""
    # Call the endpoint without session or long_term_search
    response = await client.post("/memory-prompt", json={"query": "test"})

    # Check status code (should be 400 Bad Request)
    assert response.status_code == 400

    # Check error message
    data = response.json()
    assert "detail" in data
    assert "Either session or long_term_search must be provided" in data["detail"]

@patch("agent_memory_server.api.working_memory.get_working_memory")
@pytest.mark.asyncio
async def test_memory_prompt_session_not_found(
```

```
    self, mock_get_working_memory, client
):
    """Test the memory_prompt endpoint when session is not found"""
    # Mock the session memory to return None (session not found)
    mock_get_working_memory.return_value = None

    # Call the endpoint
    query = "What's the weather like?"
    response = await client.post(
        "/memory-prompt",
        json={
            "query": query,
            "session": {
                "session_id": "nonexistent-session",
                "namespace": "test-namespace",
            },
        },
    )

    # Check status code (should be successful)
    assert response.status_code == 200

    # Check response data (should only contain the query)
    data = response.json()
    assert isinstance(data, dict)
    assert len(data["messages"]) == 1
    assert data["messages"][0]["role"] == "user"
    assert data["messages"][0]["content"]["text"] == query

@patch("agent_memory_server.api.working_memory.get_working_memory")
@patch("agent_memory_server.api.get_model_config")
@pytest.mark.asyncio
async def test_memory_prompt_with_model_name(
    self, mock_get_model_config, mock_get_working_memory, client
):
    """Test the memory_prompt endpoint with model_name parameter"""
    # Mock the model config
    model_config = MagicMock()
    model_config.max_tokens = 4000
    mock_get_model_config.return_value = model_config

    # Mock the session memory
    mock_session_memory = WorkingMemoryResponse(
        messages=[
            MemoryMessage(role="user", content="Hello"),
            MemoryMessage(role="assistant", content="Hi there"),
        ],
        memories=[],
        session_id="test-session",
        context="Previous context",
        tokens=150,
    )
    mock_get_working_memory.return_value = mock_session_memory

    # Call the endpoint with model_name
    query = "What's the weather like?"
    response = await client.post(
        "/memory-prompt",
        json={
            "query": query,
            "session": {
                "session_id": "test-session",
                "model_name": "gpt-4o",
            },
        },
    )

    # Check the model config was used
    mock_get_model_config.assert_called_once_with("gpt-4o")
```

```
        # Check status code
        assert response.status_code == 200

        # Verify the working memory function was called
        mock_get_working_memory.assert_called_once()


@pytest.mark.requires_api_keys
class TestLongTermMemoryEndpoint:
    @pytest.mark.asyncio
    async def test_create_long_term_memory_with_valid_id(self, client):
        """Test creating long-term memory with valid id"""
        payload = {
            "memories": [
                {
                    "text": "User prefers dark mode",
                    "user_id": "user123",
                    "session_id": "session123",
                    "namespace": "test",
                    "memory_type": "semantic",
                    "id": "test-client-123",
                }
            ]
        }

        response = await client.post("/long-term-memory", json=payload)
        assert response.status_code == 200

    @pytest.mark.asyncio
    async def test_create_long_term_memory_missing_id(self, client):
        """Test creating long-term memory without id should fail"""
        payload = {
            "memories": [
                {
                    "text": "User prefers dark mode",
                    "user_id": "user123",
                    "session_id": "session123",
                    "namespace": "test",
                    "memory_type": "semantic",
                    # Missing id field
                }
            ]
        }

        response = await client.post("/long-term-memory", json=payload)
        assert response.status_code == 400
        data = response.json()
        assert "id is required" in data["detail"]

    @pytest.mark.requires_api_keys
    @pytest.mark.asyncio
    async def test_create_long_term_memory_persisted_at_ignored(self, client):
        """Test that client-provided persisted_at is ignored"""
        payload = {
            "memories": [
                {
                    "text": "User prefers dark mode",
                    "id": "test-client-456",
                    "memory_type": "semantic",
                    "persisted_at": "2023-01-01T00:00:00Z",  # Use ISO string instead of datetime object
                }
            ]
        }

        response = await client.post("/long-term-memory", json=payload)
        assert response.status_code == 200

        data = response.json()
```

```
        assert data["status"] == "ok"


@pytest.mark.requires_api_keys
class TestUnifiedSearchEndpoint:
    @patch("agent_memory_server.api.long_term_memory.search_memories")
    @pytest.mark.asyncio
    async def test_unified_search(self, mock_search, client):
        """Test the unified search endpoint"""
        mock_search.return_value = MemoryRecordResultsResponse(
            total=3,
            memories=[
                MemoryRecordResult(
                    id_="working-1",
                    text="Working memory: User prefers dark mode",
                    dist=0.0,
                    memory_type="semantic",
                    persisted_at=None,  # Working memory
                ),
                MemoryRecordResult(
                    id_="long-1",
                    text="Long-term: User likes coffee",
                    dist=0.25,
                    memory_type="semantic",
                    persisted_at=datetime(2023, 1, 1, 0, 0, 0),  # Long-term memory
                ),
                MemoryRecordResult(
                    id_="long-2",
                    text="Long-term: User is allergic to peanuts",
                    dist=0.35,
                    memory_type="semantic",
                    persisted_at=datetime(2023, 1, 1, 1, 0, 0),  # Long-term memory
                ),
            ],
            next_offset=None,
        )

        # Create payload
        payload = {"text": "What are the user's preferences?"}

        # Call the unified search endpoint
        response = await client.post("/memory/search", json=payload)

        # Check status code
        assert response.status_code == 200, response.text

        # Check response structure
        data = response.json()
        assert "memories" in data
        assert "total" in data
        assert data["total"] == 3
        assert len(data["memories"]) == 3

        # Check that results include both working and long-term memory
        memories = data["memories"]

        # First result should be working memory (dist=0.0)
        assert memories[0]["id_"] == "working-1"
        assert "Working memory" in memories[0]["text"]
        assert memories[0]["dist"] == 0.0
        assert memories[0]["persisted_at"] is None

        # Other results should be long-term memory
        assert memories[1]["id_"] == "long-1"
        assert "Long-term" in memories[1]["text"]
        assert memories[1]["dist"] == 0.25
        assert memories[1]["persisted_at"] is not None

        assert memories[2]["id_"] == "long-2"
```

```
    assert "Long-term" in memories[2]["text"]
    assert memories[2]["dist"] == 0.35
    assert memories[2]["persisted_at"] is not None

@patch("agent_memory_server.api.long_term_memory.search_memories")
@pytest.mark.asyncio
async def test_unified_search_with_filters(self, mock_search, client):
    """Test the unified search endpoint with filters"""
    mock_search.return_value = MemoryRecordResultsResponse(
        total=1,
        memories=[
            MemoryRecordResult(
                id_="filtered-1",
                text="User's semantic preference",
                dist=0.1,
                memory_type="semantic",
                user_id="test-user",
                session_id="test-session",
            ),
        ],
        next_offset=None,
    )

    # Create payload with filters
    payload = {
        "text": "preferences",
        "memory_type": {"eq": "semantic"},
        "user_id": {"eq": "test-user"},
        "session_id": {"eq": "test-session"},
        "limit": 5,
    }

    # Call the unified search endpoint
    response = await client.post("/memory/search", json=payload)

    # Check status code
    assert response.status_code == 200

    # Verify the mock was called with correct parameters
    mock_search.assert_called_once()
    call_kwargs = mock_search.call_args[1]
    assert call_kwargs["text"] == "preferences"
    assert call_kwargs["limit"] == 5

    # Check response
    data = response.json()
    assert data["total"] == 1
    assert len(data["memories"]) == 1
    assert data["memories"][0]["memory_type"] == "semantic"
    assert data["memories"][0]["user_id"] == "test-user"
```

```
== tests/test_client_api.py ==
                                                                                                                                                """
Test file for the Redis Memory Server API Client.

This file contains tests that demonstrate how to use the Memory API client.
"""

from collections.abc import AsyncGenerator
from unittest.mock import AsyncMock, MagicMock, patch

import pytest
from fastapi import FastAPI
from httpx import ASGITransport, AsyncClient
from mcp.server.fastmcp.prompts import base
from mcp.types import TextContent

from agent_memory_server.api import router as memory_router
from agent_memory_server.client.api import MemoryAPIClient, MemoryClientConfig
from agent_memory_server.filters import Namespace, SessionId, Topics
from agent_memory_server.healthcheck import router as health_router
from agent_memory_server.models import (
    MemoryMessage,
    MemoryPromptResponse,
    MemoryRecord,
    MemoryRecordResult,
    MemoryRecordResultsResponse,
    SystemMessage,
    WorkingMemory,
    WorkingMemoryResponse,
)


@pytest.fixture
def memory_app() -> FastAPI:
    """Create a test FastAPI app with memory routers for testing the client."""
    app = FastAPI()
    app.include_router(health_router)
    app.include_router(memory_router)
    return app


@pytest.fixture
async def memory_test_client(
    memory_app: FastAPI,
) -> AsyncGenerator[MemoryAPIClient, None]:
    """Create a memory client that uses the test FastAPI app."""
    async with AsyncClient(
        transport=ASGITransport(app=memory_app),
        base_url="http://test",
    ) as http_client:
        # Create the memory client with our test http client
        config = MemoryClientConfig(
            base_url="http://test", default_namespace="test-namespace"
        )
        client = MemoryAPIClient(config)

        # Replace the internal http client with our test client
        client._client = http_client

        yield client


@pytest.mark.asyncio
async def test_health_check(memory_test_client: MemoryAPIClient):
    """Test the health check endpoint"""
    # Mock the response from the health endpoint
    response = await memory_test_client.health_check()
    assert response.now > 0
```

```
@pytest.mark.asyncio
async def test_session_lifecycle(memory_test_client: MemoryAPIClient):
    """Test the complete lifecycle of a session"""
    # For this test, we need to set up mocks for all the API calls
    session_id = "test-client-session"

    # Mock memory data
    memory = WorkingMemory(
        messages=[
            MemoryMessage(role="user", content="Hello from the client!"),
            MemoryMessage(role="assistant", content="Hi there, I'm the memory server!"),
        ],
        memories=[],
        context="This is a test session created by the API client.",
        session_id=session_id,
    )

    # First, mock PUT response for creating a session
    with patch(
        "agent_memory_server.working_memory.set_working_memory"
    ) as mock_set_memory:
        mock_set_memory.return_value = None

        # Step 1: Create new session memory
        response = await memory_test_client.put_session_memory(session_id, memory)
        assert response.messages[0].content == "Hello from the client!"
        assert response.messages[1].content == "Hi there, I'm the memory server!"
        assert response.context == "This is a test session created by the API client."

    # Next, mock GET response for retrieving session memory
    with patch(
        "agent_memory_server.working_memory.get_working_memory"
    ) as mock_get_memory:
        # Get memory data and explicitly exclude session_id to avoid duplicate parameter
        memory_data = memory.model_dump(exclude={"session_id"})
        mock_response = WorkingMemoryResponse(**memory_data, session_id=session_id)
        mock_get_memory.return_value = mock_response

        # Step 2: Retrieve the session memory
        session = await memory_test_client.get_session_memory(session_id)
        assert len(session.messages) == 2
        assert session.messages[0].content == "Hello from the client!"
        assert session.messages[1].content == "Hi there, I'm the memory server!"
        assert session.context == "This is a test session created by the API client."

    # Mock list sessions
    with patch("agent_memory_server.messages.list_sessions") as mock_list_sessions:
        mock_list_sessions.return_value = (1, [session_id])

        # Step 3: List sessions and verify our test session is included
        sessions = await memory_test_client.list_sessions()
        assert session_id in sessions.sessions

    # Mock delete session
    with patch(
        "agent_memory_server.working_memory.delete_working_memory"
    ) as mock_delete:
        mock_delete.return_value = None

        # Step 4: Delete the session
        response = await memory_test_client.delete_session_memory(session_id)
        assert response.status == "ok"

    # Verify it's gone by mocking a 404 response
    with patch(
        "agent_memory_server.working_memory.get_working_memory"
    ) as mock_get_memory:
        mock_get_memory.return_value = None
```

```
        # This should not raise an error anymore since the unified API returns empty working memory instead of 404
        session = await memory_test_client.get_session_memory(session_id)
        assert len(session.messages) == 0  # Should return empty working memory


@pytest.mark.asyncio
async def test_long_term_memory(memory_test_client: MemoryAPIClient):
    """Test long-term memory creation and search"""
    # Create some test memories
    memories = [
        MemoryRecord(
            text="User prefers dark mode",
            id="test-client-1",
            memory_type="semantic",
            user_id="user123",
        ),
        MemoryRecord(
            text="User is working on a Python project",
            id="test-client-2",
            memory_type="episodic",
            user_id="user123",
        ),
    ]

    # Mock the memory creation
    with patch(
        "agent_memory_server.long_term_memory.index_long_term_memories"
    ) as mock_index:
        mock_index.return_value = None

        # Store the memories
        with patch("agent_memory_server.api.settings.long_term_memory", True):
            response = await memory_test_client.create_long_term_memory(memories)
            assert response.status == "ok"

    # Mock the search results
    with patch(
        "agent_memory_server.long_term_memory.search_long_term_memories"
    ) as mock_search:
        mock_search.return_value = MemoryRecordResultsResponse(
            total=2,
            memories=[
                MemoryRecordResult(
                    id_="1",
                    text="User prefers dark mode",
                    dist=0.1,
                    user_id="user123",
                    namespace="preferences",
                ),
                MemoryRecordResult(
                    id_="2",
                    text="User likes coffee",
                    dist=0.2,
                    user_id="user123",
                    namespace="preferences",
                ),
            ],
            next_offset=None,
        )

        # Search with various filters
        with patch("agent_memory_server.api.settings.long_term_memory", True):
            results = await memory_test_client.search_long_term_memory(
                text="What color does the user prefer?",
                user_id={"eq": "test-user"},
                topics={"any": ["colors", "preferences"]},
            )
```

```
            assert results.total == 2
            # Check that we got the memories we created
            assert any(
                "dark mode" in memory.text.lower() for memory in results.memories
            )

            # Try another search using filter objects instead of dictionaries
            results = await memory_test_client.search_long_term_memory(
                text="dark mode",
                namespace=Namespace(eq="test-namespace"),
            )

            assert results.total == 2
            assert any(
                "dark mode" in memory.text.lower() for memory in results.memories
            )


@pytest.mark.asyncio
async def test_client_with_context_manager(memory_app: FastAPI):
    """Test using the client with a context manager"""
    async with (
        AsyncClient(
            transport=ASGITransport(app=memory_app),
            base_url="http://test",
        ) as http_client,
        MemoryAPIClient(MemoryClientConfig(base_url="http://test")) as client,
    ):
        # Replace the internal client
        client._client = http_client

        # Perform a simple health check
        response = await client.health_check()
        assert response.now > 0

        # The client will be automatically closed when the context block exits


@pytest.mark.asyncio
async def test_memory_prompt(memory_test_client: MemoryAPIClient):
    """Test the memory_prompt method"""
    session_id = "test-client-session"
    query = "What was my favorite color?"

    # Create expected response
    expected_messages = [
        base.UserMessage(
            content=TextContent(type="text", text="What is your favorite color?"),
        ),
        base.AssistantMessage(
            content=TextContent(type="text", text="I like blue, how about you?"),
        ),
        base.UserMessage(
            content=TextContent(type="text", text=query),
        ),
    ]

    # Create expected response payload
    expected_response = MemoryPromptResponse(messages=expected_messages)

    # Mock the HTTP client's post method directly
    with patch.object(memory_test_client._client, "post") as mock_post:
        mock_response = AsyncMock()
        mock_response.status_code = 200
        mock_response.raise_for_status = MagicMock(return_value=None)
        mock_response.json = MagicMock(return_value=expected_response.model_dump())
        mock_post.return_value = mock_response

        # Test the client method
```

```
        response = await memory_test_client.memory_prompt(
            query=query,
            session_id=session_id,
            namespace="test-namespace",
            window_size=5,
            model_name="gpt-4o",
            context_window_max=4000,
        )

        # Verify the response
        assert len(response.messages) == 3
        assert isinstance(response.messages[0].content, TextContent)
        assert response.messages[0].content.text.startswith(
            "What is your favorite color?"
        )
        assert isinstance(response.messages[-1].content, TextContent)
        assert response.messages[-1].content.text == query

        # Test without session_id (only semantic search)
        mock_post.reset_mock()
        mock_post.return_value = mock_response

        response = await memory_test_client.memory_prompt(
            query=query,
        )

        # Verify the response is the same (it's mocked)
        assert len(response.messages) == 3


@pytest.mark.asyncio
async def test_hydrate_memory_prompt(memory_test_client: MemoryAPIClient):
    """Test the hydrate_memory_prompt method with filters"""
    query = "What was my favorite color?"

    # Create expected response
    expected_messages = [
        base.AssistantMessage(
            content=TextContent(
                type="text",
                text="The user's favorite color is blue",
            ),
        ),
        base.UserMessage(
            content=TextContent(type="text", text=query),
        ),
    ]

    # Create expected response payload
    expected_response = MemoryPromptResponse(messages=expected_messages)

    # Mock the HTTP client's post method directly
    with patch.object(memory_test_client._client, "post") as mock_post:
        mock_response = AsyncMock()
        mock_response.status_code = 200
        mock_response.raise_for_status = MagicMock(return_value=None)
        mock_response.json = MagicMock(return_value=expected_response.model_dump())
        mock_post.return_value = mock_response

        # Test with filter dictionaries
        response = await memory_test_client.hydrate_memory_prompt(
            query=query,
            session_id={"eq": "test-session"},
            namespace={"eq": "test-namespace"},
            topics={"any": ["preferences", "colors"]},
            limit=5,
        )

        # Verify the response
```

```
        assert len(response.messages) == 2
        assert isinstance(response.messages[0].content, TextContent)
        assert "favorite color" in response.messages[0].content.text
        assert isinstance(response.messages[1].content, TextContent)
        assert response.messages[1].content.text == query

        # Test with filter objects
        mock_post.reset_mock()
        mock_post.return_value = mock_response

        response = await memory_test_client.hydrate_memory_prompt(
            query=query,
            session_id=SessionId(eq="test-session"),
            namespace=Namespace(eq="test-namespace"),
            topics=Topics(any=["preferences"]),
            window_size=10,
            model_name="gpt-4o",
        )

        # Response should be the same because it's mocked
        assert len(response.messages) == 2

        # Test with no filters (just query)
        mock_post.reset_mock()
        mock_post.return_value = mock_response

        response = await memory_test_client.hydrate_memory_prompt(
            query=query,
        )

        # Response should still be the same (mocked)
        assert len(response.messages) == 2


@pytest.mark.asyncio
async def test_memory_prompt_integration(memory_test_client: MemoryAPIClient):
    """Test the memory_prompt method with both session and long-term search"""
    session_id = "test-client-session"
    query = "What was my favorite color?"

    # Create expected response with both session and LTM content
    expected_messages = [
        SystemMessage(
            content=TextContent(
                type="text",
                text="## A summary of the conversation so far\nPrevious conversation about website design preferences
            ),
        ),
        base.UserMessage(
            content=TextContent(
                type="text", text="What is a good color for a website?"
            ),
        ),
        base.AssistantMessage(
            content=TextContent(
                type="text",
                text="It depends on the website's purpose. Blue is often used for professional sites.",
            ),
        ),
        SystemMessage(
            content=TextContent(
                type="text",
                text="## Long term memories related to the user's query\n - The user's favorite color is blue",
            ),
        ),
        base.UserMessage(
            content=TextContent(type="text", text=query),
        ),
    ]
```

```
# Create expected response payload
expected_response = MemoryPromptResponse(messages=expected_messages)

# Mock the HTTP client's post method directly
with patch.object(memory_test_client._client, "post") as mock_post:
    mock_response = AsyncMock()
    mock_response.status_code = 200
    mock_response.raise_for_status = MagicMock(return_value=None)
    mock_response.json = MagicMock(return_value=expected_response.model_dump())
    mock_post.return_value = mock_response

    # Let the client method run with our mocked response
    response = await memory_test_client.memory_prompt(
        query=query,
        session_id=session_id,
        namespace="test-namespace",
    )

    # Check that both session memory and LTM are in the response
    assert len(response.messages) == 5

    # Extract text from contents
    message_texts = []
    for m in response.messages:
        if isinstance(m.content, TextContent):
            message_texts.append(m.content.text)

    # The messages should include at least one from the session
    assert any("website" in text for text in message_texts)
    # And at least one from LTM
    assert any("favorite color is blue" in text for text in message_texts)
    # And the query itself
    assert query in message_texts[-1]
```

== tests/test_extraction.py ==

```
from u

import numpy as np
import pytest

from agent_memory_server.config import settings
from agent_memory_server.extraction import (
    extract_entities,
    extract_topics_bertopic,
    extract_topics_llm,
    handle_extraction,
)


@pytest.fixture
def mock_bertopic():
    """Mock BERTopic model"""
    mock = Mock()
    # Mock transform to return topic indices and probabilities
    mock.transform.return_value = (np.array([1]), np.array([0.8]))
    # Mock get_topic to return topic terms
    mock.get_topic.side_effect = lambda x: [("technology", 0.8), ("business", 0.7)]
    return mock


@pytest.fixture
def mock_ner():
    """Mock NER pipeline"""

    def mock_ner_fn(text):
        return [
            {"word": "John", "entity": "PER", "score": 0.99},
            {"word": "Google", "entity": "ORG", "score": 0.98},
            {"word": "Mountain", "entity": "LOC", "score": 0.97},
            {"word": "##View", "entity": "LOC", "score": 0.97},
        ]

    return Mock(side_effect=mock_ner_fn)


@pytest.mark.asyncio
class TestTopicExtraction:
    @patch("agent_memory_server.extraction.get_topic_model")
    async def test_extract_topics_success(self, mock_get_topic_model, mock_bertopic):
        """Test successful topic extraction"""
        mock_get_topic_model.return_value = mock_bertopic
        text = "Discussion about AI technology and business"

        topics = extract_topics_bertopic(text)

        assert set(topics) == {"technology", "business"}
        mock_bertopic.transform.assert_called_once_with([text])

    @patch("agent_memory_server.extraction.get_topic_model")
    async def test_extract_topics_no_valid_topics(
        self, mock_get_topic_model, mock_bertopic
    ):
        """Test when no valid topics are found"""
        mock_bertopic.transform.return_value = (np.array([-1]), np.array([0.0]))
        mock_get_topic_model.return_value = mock_bertopic

        topics = extract_topics_bertopic("Test message")

        assert topics == []
        mock_bertopic.transform.assert_called_once()


@pytest.mark.asyncio
```

```
class TestEntityExtraction:
    @patch("agent_memory_server.extraction.get_ner_model")
    async def test_extract_entities_success(self, mock_get_ner_model, mock_ner):
        """Test successful entity extraction"""
        mock_get_ner_model.return_value = mock_ner
        text = "John works at Google in Mountain View"

        entities = extract_entities(text)

        assert set(entities) == {"John", "Google", "MountainView"}
        mock_ner.assert_called_once_with(text)

    @patch("agent_memory_server.extraction.get_ner_model")
    async def test_extract_entities_error(self, mock_get_ner_model):
        """Test handling of NER model error"""
        mock_get_ner_model.side_effect = Exception("Model error")

        entities = extract_entities("Test message")

        assert entities == []


@pytest.mark.asyncio
class TestHandleExtraction:
    @patch("agent_memory_server.extraction.extract_topics_llm")
    @patch("agent_memory_server.extraction.extract_entities")
    async def test_handle_extraction(
        self, mock_extract_entities, mock_extract_topics_llm
    ):
        """Test extraction with topics/entities"""
        mock_extract_topics_llm.return_value = ["AI", "business"]
        mock_extract_entities.return_value = ["John", "Sarah", "Google"]

        topics, entities = await handle_extraction(
            "John and Sarah discussed AI at Google."
        )

        # Check that topics are as expected
        assert mock_extract_topics_llm.called
        assert set(topics) == {"AI", "business"}
        assert len(topics) == 2

        # Check that entities are as expected
        assert mock_extract_entities.called
        assert set(entities) == {"John", "Sarah", "Google"}
        assert len(entities) == 3

    @patch("agent_memory_server.extraction.extract_topics_llm")
    @patch("agent_memory_server.extraction.extract_entities")
    async def test_handle_extraction_disabled_features(
        self, mock_extract_entities, mock_extract_topics_llm
    ):
        """Test when features are disabled"""
        # Temporarily disable features
        original_topic_setting = settings.enable_topic_extraction
        original_ner_setting = settings.enable_ner
        settings.enable_topic_extraction = False
        settings.enable_ner = False

        try:
            topics, entities = await handle_extraction("Test message")

            assert topics == []
            assert entities == []
            mock_extract_topics_llm.assert_not_called()
            mock_extract_entities.assert_not_called()
        finally:
            # Restore settings
            settings.enable_topic_extraction = original_topic_setting
```

```
            settings.enable_ner = original_ner_setting


@pytest.mark.requires_api_keys
class TestTopicExtractionIntegration:
    @pytest.mark.asyncio
    async def test_bertopic_integration(self):
        """Integration test for BERTopic topic extraction (skipped if not available)"""

        # Save and set topic_model_source
        original_source = settings.topic_model_source
        settings.topic_model_source = "BERTopic"
        sample_text = (
            "OpenAI and Google are leading companies in artificial intelligence."
        )
        try:
            try:
                # Try to import BERTopic and check model loading
                topics = extract_topics_bertopic(sample_text)
                # print(f"[DEBUG] BERTopic returned topics: {topics}")
            except Exception as e:
                pytest.skip(f"BERTopic integration test skipped: {e}")
            assert isinstance(topics, list)
            expected_keywords = {
                "generative",
                "transformer",
                "neural",
                "learning",
                "trained",
                "multimodal",
                "generates",
                "models",
                "encoding",
                "text",
            }
            assert any(t.lower() in expected_keywords for t in topics)
        finally:
            settings.topic_model_source = original_source

    @pytest.mark.asyncio
    async def test_llm_integration(self):
        """Integration test for LLM-based topic extraction (skipped if no API key)"""

        # Save and set topic_model_source
        original_source = settings.topic_model_source
        settings.topic_model_source = "LLM"
        sample_text = (
            "OpenAI and Google are leading companies in artificial intelligence."
        )
        try:
            # Check for API key
            if not (settings.openai_api_key or settings.anthropic_api_key):
                pytest.skip("No LLM API key available for integration test.")
            topics = await extract_topics_llm(sample_text)
            assert isinstance(topics, list)
            assert any(
                t.lower() in ["technology", "business", "artificial intelligence"]
                for t in topics
            )
        finally:
            settings.topic_model_source = original_source


class TestHandleExtractionPathSelection:
    @pytest.mark.asyncio
    @patch("agent_memory_server.extraction.extract_topics_bertopic")
    @patch("agent_memory_server.extraction.extract_topics_llm")
    async def test_handle_extraction_path_selection(
        self, mock_extract_topics_llm, mock_extract_topics_bertopic
```

```
):
    """Test that handle_extraction uses the correct extraction path based on settings.topic_model_source"""

    sample_text = (
        "OpenAI and Google are leading companies in artificial intelligence."
    )
    original_source = settings.topic_model_source
    original_enable_topic_extraction = settings.enable_topic_extraction
    original_enable_ner = settings.enable_ner
    try:
        # Enable topic extraction and disable NER for clarity
        settings.enable_topic_extraction = True
        settings.enable_ner = False

        # Test BERTopic path
        settings.topic_model_source = "BERTopic"
        mock_extract_topics_bertopic.return_value = ["technology"]
        mock_extract_topics_llm.return_value = ["should not be called"]
        topics, _ = await handle_extraction(sample_text)
        mock_extract_topics_bertopic.assert_called_once()
        mock_extract_topics_llm.assert_not_called()
        assert topics == ["technology"]
        mock_extract_topics_bertopic.reset_mock()

        # Test LLM path
        settings.topic_model_source = "LLM"
        mock_extract_topics_llm.return_value = ["ai"]
        topics, _ = await handle_extraction(sample_text)
        mock_extract_topics_llm.assert_called_once()
        mock_extract_topics_bertopic.assert_not_called()
        assert topics == ["ai"]
    finally:
        settings.topic_model_source = original_source
        settings.enable_topic_extraction = original_enable_topic_extraction
        settings.enable_ner = original_enable_ner
```

== tests/test_llms.py ==

```
import
from unittest.mock import AsyncMock, MagicMock, patch

import numpy as np
import pytest

from agent_memory_server.llms import (
    ModelProvider,
    OpenAIClientWrapper,
    get_model_client,
    get_model_config,
)


@pytest.mark.asyncio
class TestOpenAIClientWrapper:
    @patch.dict(
        os.environ,
        {
            "OPENAI_API_KEY": "test-key",
        },
    )
    @patch("agent_memory_server.llms.AsyncOpenAI")
    async def test_init_regular_openai(self, mock_openai):
        """Test initializing with regular OpenAI"""
        # Set up the mock to return an AsyncMock
        mock_openai.return_value = AsyncMock()

        OpenAIClientWrapper()

        # Verify the client was created
        assert mock_openai.called

    @patch.object(OpenAIClientWrapper, "__init__", return_value=None)
    async def test_create_embedding(self, mock_init):
        """Test creating embeddings"""
        # Create a client with mocked init
        client = OpenAIClientWrapper()

        # Mock the embedding client and response
        mock_response = AsyncMock()
        mock_response.data = [
            MagicMock(embedding=[0.1, 0.2, 0.3]),
            MagicMock(embedding=[0.4, 0.5, 0.6]),
        ]

        client.embedding_client = AsyncMock()
        client.embedding_client.embeddings.create = AsyncMock(
            return_value=mock_response
        )

        # Test creating embeddings
        query_vec = ["Hello, world!", "How are you?"]
        embeddings = await client.create_embedding(query_vec)

        # Verify embeddings were created correctly
        assert len(embeddings) == 2
        # Convert NumPy array to list or use np.array_equal for comparison
        assert np.array_equal(
            embeddings[0], np.array([0.1, 0.2, 0.3], dtype=np.float32)
        )
        assert np.array_equal(
            embeddings[1], np.array([0.4, 0.5, 0.6], dtype=np.float32)
        )

        # Verify the client was called with correct parameters
        client.embedding_client.embeddings.create.assert_called_with(
            model="text-embedding-ada-002", input=query_vec
```

```
        )

    @patch.object(OpenAIClientWrapper, "__init__", return_value=None)
    async def test_create_chat_completion(self, mock_init):
        """Test creating chat completions"""
        # Create a client with mocked init
        client = OpenAIClientWrapper()

        # Mock the completion client and response
        # Create a response structure that matches our new ChatResponse format
        mock_response = AsyncMock()
        mock_response.choices = [{"message": {"content": "Test response"}}]
        mock_response.usage = {"total_tokens": 100}

        client.completion_client = AsyncMock()
        client.completion_client.chat.completions.create = AsyncMock(
            return_value=mock_response
        )

        # Test creating chat completion
        model = "gpt-3.5-turbo"
        prompt = "Hello, world!"
        response = await client.create_chat_completion(model, prompt)

        # Verify the response contains the expected structure
        assert response.choices[0]["message"]["content"] == "Test response"
        assert response.total_tokens == 100

        # Verify the client was called with correct parameters
        client.completion_client.chat.completions.create.assert_called_with(
            model=model, messages=[{"role": "user", "content": prompt}]
        )


@pytest.mark.parametrize(
    ("model_name", "expected_provider", "expected_max_tokens"),
    [
        ("gpt-4o", "openai", 128000),
        ("claude-3-sonnet-20240229", "anthropic", 200000),
        ("nonexistent-model", "openai", 128000),  # Should default to GPT-4o-mini
    ],
)
def test_get_model_config(model_name, expected_provider, expected_max_tokens):
    """Test the get_model_config function"""
    # Get the model config
    config = get_model_config(model_name)

    # Check the provider
    if expected_provider == "openai":
        assert config.provider == ModelProvider.OPENAI
    else:
        assert config.provider == ModelProvider.ANTHROPIC

    # Check the max tokens
    assert config.max_tokens == expected_max_tokens


@pytest.mark.asyncio
async def test_get_model_client():
    """Test the get_model_client function"""
    # Test with OpenAI model
    with (
        patch.dict(os.environ, {"OPENAI_API_KEY": "test-key"}),
        patch("agent_memory_server.llms.OpenAIClientWrapper") as mock_openai,
    ):
        mock_openai.return_value = "openai-client"
        client = await get_model_client("gpt-4")
        assert client == "openai-client"
```

```
# Test with Anthropic model
with (
    patch.dict(os.environ, {"ANTHROPIC_API_KEY": "test-key"}),
    patch("agent_memory_server.llms.AnthropicClientWrapper") as mock_anthropic,
):
    mock_anthropic.return_value = "anthropic-client"
    client = await get_model_client("claude-3-sonnet-20240229")
    assert client == "anthropic-client"
```

== tests/test_long_term_memory.py ==

impor

```
from datetime import UTC, datetime
from unittest import mock
from unittest.mock import AsyncMock, MagicMock, patch

import numpy as np
import pytest
from redis.commands.search.document import Document
from ulid import ULID

from agent_memory_server.filters import SessionId
from agent_memory_server.long_term_memory import (
    deduplicate_by_id,
    index_long_term_memories,
    promote_working_memory_to_long_term,
    search_long_term_memories,
    search_memories,
)
from agent_memory_server.models import MemoryRecord, MemoryRecordResult, MemoryTypeEnum
from agent_memory_server.utils.redis import ensure_search_index_exists


class TestLongTermMemory:
    @pytest.mark.asyncio
    async def test_index_memories(
        self, mock_openai_client, mock_async_redis_client, session
    ):
        """Test indexing messages"""
        long_term_memories = [
            MemoryRecord(text="Paris is the capital of France", session_id=session),
            MemoryRecord(text="France is a country in Europe", session_id=session),
        ]

        # Create two separate embedding vectors
        mock_vectors = [
            np.array([0.1, 0.2, 0.3, 0.4], dtype=np.float32).tobytes(),
            np.array([0.5, 0.6, 0.7, 0.8], dtype=np.float32).tobytes(),
        ]

        mock_vectorizer = MagicMock()
        mock_vectorizer.aembed_many = AsyncMock(return_value=mock_vectors)

        mock_async_redis_client.hset = AsyncMock()

        with mock.patch(
            "agent_memory_server.long_term_memory.OpenAITextVectorizer",
            return_value=mock_vectorizer,
        ):
            await index_long_term_memories(
                long_term_memories,
                redis_client=mock_async_redis_client,
            )

        # Check that create_embedding was called with the right arguments
        contents = [memory.text for memory in long_term_memories]
        mock_vectorizer.aembed_many.assert_called_with(
            contents,
            batch_size=20,
            as_buffer=True,
        )

        # Verify one of the calls to make sure the data is correct
        for i, call in enumerate(mock_async_redis_client.hset.call_args_list):
            args, kwargs = call

            # Check that the key starts with the memory key prefix
            assert args[0].startswith("memory:")
```

```
        # Check that the mapping contains the right keys
        mapping = kwargs["mapping"]
        assert mapping == {
            "text": long_term_memories[i].text,
            "id_": long_term_memories[i].id_,
            "session_id": long_term_memories[i].session_id,
            "user_id": long_term_memories[i].user_id,
            "last_accessed": long_term_memories[i].last_accessed,
            "created_at": long_term_memories[i].created_at,
            "vector": mock_vectors[i],
        }

@pytest.mark.asyncio
async def test_search_memories(self, mock_openai_client, mock_async_redis_client):
    """Test searching memories"""
    # Set up the mock embedding response
    mock_vector = np.array([0.1, 0.2, 0.3, 0.4], dtype=np.float32)
    mock_vectorizer = MagicMock()
    mock_vectorizer.aembed = AsyncMock(return_value=mock_vector)

    class MockResult:
        def __init__(self, docs):
            self.total = len(docs)
            self.docs = docs

    mock_now = time.time()

    mock_query = AsyncMock()
    # Return a list of documents directly instead of a MockResult object
    mock_query.return_value = [
        Document(
            id=b"doc1",
            id_=str(ULID()),
            text=b"Hello, world!",
            vector_distance=0.25,
            created_at=mock_now,
            last_accessed=mock_now,
            user_id=None,
            session_id=None,
            namespace=None,
            topics=None,
            entities=None,
        ),
        Document(
            id=b"doc2",
            id_=str(ULID()),
            text=b"Hi there!",
            vector_distance=0.75,
            created_at=mock_now,
            last_accessed=mock_now,
            user_id=None,
            session_id=None,
            namespace=None,
            topics=None,
            entities=None,
        ),
    ]

    mock_index = MagicMock()
    mock_index.query = mock_query

    query = "What is the meaning of life?"
    session_id = SessionId(eq="test-session")

    with (
        mock.patch(
            "agent_memory_server.long_term_memory.OpenAITextVectorizer",
            return_value=mock_vectorizer,
        ),
```

```
        mock.patch(
            "agent_memory_server.long_term_memory.get_search_index",
            return_value=mock_index,
        ),
    ):
        results = await search_long_term_memories(
            query,
            mock_async_redis_client,
            session_id=session_id,
        )

    # Check that create_embedding was called with the right arguments
    mock_vectorizer.aembed.assert_called_with(query)

    assert mock_index.query.call_count == 1

    assert len(results.memories) == 1
    assert isinstance(results.memories[0], MemoryRecordResult)
    assert results.memories[0].text == "Hello, world!"
    assert results.memories[0].dist == 0.25
    assert results.memories[0].memory_type == "message"

@pytest.mark.asyncio
async def test_search_memories_unified_search(self, mock_async_redis_client):
    """Test unified search across working memory and long-term memory"""

    from agent_memory_server.models import (
        MemoryRecordResults,
        WorkingMemory,
    )

    # Mock search_long_term_memories to return some long-term results
    mock_long_term_results = MemoryRecordResults(
        total=1,
        memories=[
            MemoryRecordResult(
                id_="long-term-1",
                text="Long-term: User likes coffee",
                dist=0.3,
                memory_type=MemoryTypeEnum.SEMANTIC,
                created_at=datetime.fromtimestamp(1000),
                updated_at=datetime.fromtimestamp(1000),
                last_accessed=datetime.fromtimestamp(1000),
            )
        ],
    )

    # Mock working memory with matching content
    test_working_memory = WorkingMemory(
        session_id="test-session",
        namespace="test",
        messages=[],
        memories=[
            MemoryRecord(
                text="Working memory: coffee preferences",
                id="working-1",
                id_="working-1",  # Set both id and id_ for consistency
                memory_type=MemoryTypeEnum.SEMANTIC,
                persisted_at=None,  # Not persisted yet
            )
        ],
    )

    with (
        patch(
            "agent_memory_server.long_term_memory.search_long_term_memories"
        ) as mock_search_lt,
        patch("agent_memory_server.messages.list_sessions") as mock_list_sessions,
        patch(
```

```
            "agent_memory_server.working_memory.get_working_memory"
        ) as mock_get_wm,
        patch("agent_memory_server.long_term_memory.settings") as mock_settings,
    ):
        # Setup mocks
        mock_settings.long_term_memory = True
        mock_search_lt.return_value = mock_long_term_results
        mock_list_sessions.return_value = (1, ["test-session"])
        mock_get_wm.return_value = test_working_memory

        # Call search_memories
        results = await search_memories(
            text="coffee",
            redis=mock_async_redis_client,
            include_working_memory=True,
            include_long_term_memory=True,
            limit=10,
            offset=0,
        )

        # Verify both search functions were called
        mock_search_lt.assert_called_once()
        mock_list_sessions.assert_called_once()
        mock_get_wm.assert_called_once()

        # Verify results contain both working and long-term memory
        assert results.total == 2  # 1 from long-term + 1 from working
        assert len(results.memories) == 2

        # Working memory should come first (dist=0.0)
        working_result = results.memories[0]
        assert working_result.id_ == "working-1"
        assert working_result.text == "Working memory: coffee preferences"
        assert working_result.dist == 0.0

        # Long-term memory should come second
        long_term_result = results.memories[1]
        assert long_term_result.id_ == "long-term-1"
        assert long_term_result.text == "Long-term: User likes coffee"
        assert long_term_result.dist == 0.3

@pytest.mark.asyncio
async def test_deduplicate_by_id(self, mock_async_redis_client):
    """Test id-based deduplication"""
    # Create a memory with an id
    memory = MemoryRecord(
        text="Test memory",
        id="test-client-123",
        namespace="test",
    )

    # Mock Redis search to return no existing memory with this id
    mock_async_redis_client.execute_command.return_value = [0]

    result_memory, was_overwrite = await deduplicate_by_id(
        memory=memory,
        redis_client=mock_async_redis_client,
    )

    assert was_overwrite is False

    # Mock Redis search to return an existing memory with the same id
    mock_async_redis_client.execute_command.return_value = [
        1,
        "memory:existing-key",
        "1234567890",
    ]
    # Mock the delete method as an AsyncMock
    mock_async_redis_client.delete = AsyncMock()
```

```
    result_memory, was_overwrite = await deduplicate_by_id(
        memory=memory,
        redis_client=mock_async_redis_client,
    )
    assert was_overwrite is True
    assert result_memory is not None
    assert result_memory.id == memory.id

    # Verify delete was called
    mock_async_redis_client.delete.assert_called_once_with("memory:existing-key")

@pytest.mark.asyncio
async def test_promote_working_memory_to_long_term(self, mock_async_redis_client):
    """Test promotion of working memory to long-term storage"""
    from datetime import datetime
    from unittest.mock import patch

    from agent_memory_server.models import MemoryRecord, WorkingMemory

    # Create test memories - some persisted, some not
    persisted_memory = MemoryRecord(
        text="Already persisted memory",
        id="persisted-id",
        namespace="test",
        memory_type=MemoryTypeEnum.SEMANTIC,
        persisted_at=datetime.now(UTC),
    )

    unpersisted_memory1 = MemoryRecord(
        text="Unpersisted memory 1",
        id="unpersisted-1",
        namespace="test",
        memory_type=MemoryTypeEnum.SEMANTIC,
        persisted_at=None,
    )

    unpersisted_memory2 = MemoryRecord(
        text="Unpersisted memory 2",
        id="unpersisted-2",
        namespace="test",
        memory_type=MemoryTypeEnum.EPISODIC,
        persisted_at=None,
    )

    test_working_memory = WorkingMemory(
        session_id="test-session",
        namespace="test",
        messages=[],
        memories=[persisted_memory, unpersisted_memory1, unpersisted_memory2],
    )

    # Mock working_memory functions
    with (
        patch("agent_memory_server.working_memory.get_working_memory") as mock_get,
        patch("agent_memory_server.working_memory.set_working_memory") as mock_set,
        patch(
            "agent_memory_server.long_term_memory.deduplicate_by_id"
        ) as mock_dedup,
        patch(
            "agent_memory_server.long_term_memory.index_long_term_memories"
        ) as mock_index,
    ):
        # Setup mocks
        mock_get.return_value = test_working_memory
        mock_set.return_value = None
        mock_dedup.side_effect = [
            (unpersisted_memory1, False),  # First call - no overwrite
            (unpersisted_memory2, False),  # Second call - no overwrite
```

```
        ]
        mock_index.return_value = None

        # Call the promotion function
        promoted_count = await promote_working_memory_to_long_term(
            session_id="test-session",
            namespace="test",
            redis_client=mock_async_redis_client,
        )

        # Verify results
        assert promoted_count == 2

        # Verify working memory was retrieved
        mock_get.assert_called_once_with(
            session_id="test-session",
            namespace="test",
            redis_client=mock_async_redis_client,
        )

        # Verify deduplication was called for unpersisted memories
        assert mock_dedup.call_count == 2

        # Verify indexing was called for unpersisted memories
        assert mock_index.call_count == 2

        # Verify working memory was updated with new timestamps
        mock_set.assert_called_once()
        updated_memory = mock_set.call_args[1]["working_memory"]

        # Check that the unpersisted memories now have persisted_at set
        unpersisted_memories_updated = [
            mem
            for mem in updated_memory.memories
            if mem.id in ["unpersisted-1", "unpersisted-2"]
        ]
        assert len(unpersisted_memories_updated) == 2
        for mem in unpersisted_memories_updated:
            assert mem.persisted_at is not None
            assert isinstance(mem.persisted_at, datetime)

        # Check that already persisted memory was unchanged
        persisted_memories = [
            mem for mem in updated_memory.memories if mem.id == "persisted-id"
        ]
        assert len(persisted_memories) == 1
        assert persisted_memories[0].persisted_at == persisted_memory.persisted_at

@pytest.mark.asyncio
async def test_sync_and_conflict_safety(self, mock_async_redis_client):
    """Test that client state resubmission is safe and converges properly."""
    from datetime import datetime
    from unittest.mock import patch

    from agent_memory_server.models import MemoryRecord, WorkingMemory

    # Create test memories - some persisted, some not
    persisted_memory = MemoryRecord(
        text="Already persisted memory",
        id="persisted-id",
        namespace="test",
        memory_type=MemoryTypeEnum.SEMANTIC,
        persisted_at=datetime.now(UTC),
    )

    unpersisted_memory1 = MemoryRecord(
        text="Unpersisted memory 1",
        id="unpersisted-1",
        namespace="test",
```

```
    memory_type=MemoryTypeEnum.SEMANTIC,
    persisted_at=None,
)

unpersisted_memory2 = MemoryRecord(
    text="Unpersisted memory 2",
    id="unpersisted-2",
    namespace="test",
    memory_type=MemoryTypeEnum.EPISODIC,
    persisted_at=None,
)

test_working_memory = WorkingMemory(
    session_id="test-session",
    namespace="test",
    messages=[],
    memories=[persisted_memory, unpersisted_memory1, unpersisted_memory2],
)

# Mock working_memory functions
with (
    patch("agent_memory_server.working_memory.get_working_memory") as mock_get,
    patch("agent_memory_server.working_memory.set_working_memory") as mock_set,
    patch(
        "agent_memory_server.long_term_memory.deduplicate_by_id"
    ) as mock_dedup,
    patch(
        "agent_memory_server.long_term_memory.index_long_term_memories"
    ) as mock_index,
):
    # Setup mocks
    mock_get.return_value = test_working_memory
    mock_set.return_value = None
    mock_dedup.side_effect = [
        (unpersisted_memory1, False),  # First call - no overwrite
        (unpersisted_memory2, False),  # Second call - no overwrite
    ]
    mock_index.return_value = None

    # Call the promotion function
    promoted_count = await promote_working_memory_to_long_term(
        session_id="test-session",
        namespace="test",
        redis_client=mock_async_redis_client,
    )

    # Verify results
    assert promoted_count == 2

    # Verify working memory was retrieved
    mock_get.assert_called_once_with(
        session_id="test-session",
        namespace="test",
        redis_client=mock_async_redis_client,
    )

    # Verify deduplication was called for unpersisted memories
    assert mock_dedup.call_count == 2

    # Verify indexing was called for unpersisted memories
    assert mock_index.call_count == 2

    # Verify working memory was updated with new timestamps
    mock_set.assert_called_once()
    updated_memory = mock_set.call_args[1]["working_memory"]

    # Check that the unpersisted memories now have persisted_at set
    unpersisted_memories_updated = [
        mem
```

```
        for mem in updated_memory.memories
        if mem.id in ["unpersisted-1", "unpersisted-2"]
    ]
    assert len(unpersisted_memories_updated) == 2
    for mem in unpersisted_memories_updated:
        assert mem.persisted_at is not None
        assert isinstance(mem.persisted_at, datetime)

    # Check that already persisted memory was unchanged
    persisted_memories = [
        mem for mem in updated_memory.memories if mem.id == "persisted-id"
    ]
    assert len(persisted_memories) == 1
    assert persisted_memories[0].persisted_at == persisted_memory.persisted_at

# Now test client resubmission scenario
# Simulate client resubmitting stale state with new memory
resubmitted_memory = WorkingMemory(
    session_id="test-session",
    namespace="test",
    messages=[],
    memories=[
        # Existing memory resubmitted without persisted_at (client doesn't track this)
        MemoryRecord(
            text="Unpersisted memory 1",
            id="unpersisted-1",  # Same id as before
            namespace="test",
            memory_type=MemoryTypeEnum.SEMANTIC,
            persisted_at=None,  # Client doesn't know about server timestamps
        ),
        # New memory from client
        MemoryRecord(
            text="New memory from client",
            id="new-memory-3",
            namespace="test",
            memory_type=MemoryTypeEnum.SEMANTIC,
            persisted_at=None,
        ),
    ],
)

with (
    patch("agent_memory_server.working_memory.get_working_memory") as mock_get2,
    patch("agent_memory_server.working_memory.set_working_memory") as mock_set2,
    patch(
        "agent_memory_server.long_term_memory.deduplicate_by_id"
    ) as mock_dedup2,
    patch(
        "agent_memory_server.long_term_memory.index_long_term_memories"
    ) as mock_index2,
):
    # Setup mocks for resubmission scenario
    mock_get2.return_value = resubmitted_memory
    mock_set2.return_value = None
    # First call: existing memory found (overwrite)
    # Second call: new memory, no existing (no overwrite)
    mock_dedup2.side_effect = [
        (resubmitted_memory.memories[0], True),  # Overwrite existing
        (resubmitted_memory.memories[1], False),  # New memory
    ]
    mock_index2.return_value = None

    # Call promotion again
    promoted_count_2 = await promote_working_memory_to_long_term(
        session_id="test-session",
        namespace="test",
        redis_client=mock_async_redis_client,
    )
```

```
            # Both memories should be promoted (one overwrite, one new)
            assert promoted_count_2 == 2

            # Verify final working memory state
            mock_set2.assert_called_once()
            final_memory = mock_set2.call_args[1]["working_memory"]

            # Both memories should have persisted_at set
            for mem in final_memory.memories:
                assert mem.persisted_at is not None

            # This demonstrates that:
            # 1. Client can safely resubmit stale state
            # 2. Server handles id-based overwrites correctly
            # 3. Working memory converges to consistent state with proper timestamps


@pytest.mark.requires_api_keys
class TestLongTermMemoryIntegration:
    """Integration tests for long-term memory"""

    @pytest.mark.asyncio
    async def test_search_messages(self, async_redis_client):
        """Test searching messages"""
        await ensure_search_index_exists(async_redis_client)

        long_term_memories = [
            MemoryRecord(text="Paris is the capital of France", session_id="123"),
            MemoryRecord(text="France is a country in Europe", session_id="123"),
        ]

        with mock.patch(
            "agent_memory_server.long_term_memory.get_redis_conn",
            return_value=async_redis_client,
        ):
            await index_long_term_memories(
                long_term_memories,
                redis_client=async_redis_client,
            )

        results = await search_long_term_memories(
            "What is the capital of France?",
            async_redis_client,
            session_id=SessionId(eq="123"),
            limit=1,
        )

        assert results.total == 1
        assert len(results.memories) == 1
        assert results.memories[0].text == "Paris is the capital of France"
        assert results.memories[0].session_id == "123"
        assert results.memories[0].memory_type == "message"

    @pytest.mark.asyncio
    async def test_search_messages_with_distance_threshold(self, async_redis_client):
        """Test searching messages with a distance threshold"""
        await ensure_search_index_exists(async_redis_client)

        long_term_memories = [
            MemoryRecord(text="Paris is the capital of France", session_id="123"),
            MemoryRecord(text="France is a country in Europe", session_id="123"),
        ]

        with mock.patch(
            "agent_memory_server.long_term_memory.get_redis_conn",
            return_value=async_redis_client,
        ):
            await index_long_term_memories(
                long_term_memories,
```

```
        redis_client=async_redis_client,
    )

results = await search_long_term_memories(
    "What is the capital of France?",
    async_redis_client,
    session_id=SessionId(eq="123"),
    distance_threshold=0.1,
    limit=2,
)

assert results.total == 1
assert len(results.memories) == 1
assert results.memories[0].text == "Paris is the capital of France"
assert results.memories[0].session_id == "123"
assert results.memories[0].memory_type == "message"
```

== tests/test_mcp.py ==

impor

```
from datetime import UTC, datetime
from unittest import mock

import pytest
from mcp.shared.memory import (
    create_connected_server_and_client_session as client_session,
)
from mcp.types import CallToolResult, TextContent

from agent_memory_server.mcp import mcp_app
from agent_memory_server.models import (
    MemoryPromptRequest,
    MemoryPromptResponse,
    MemoryRecord,
    MemoryRecordResult,
    MemoryRecordResults,
    SystemMessage,
    WorkingMemoryResponse,
)


@pytest.fixture
async def mcp_test_setup(async_redis_client, search_index):
    with (
        mock.patch(
            "agent_memory_server.long_term_memory.get_redis_conn",
            return_value=async_redis_client,
        ) as _mock_ltm_redis,
        mock.patch(
            "agent_memory_server.api.get_redis_conn",
            return_value=async_redis_client,
            create=True,
        ) as _mock_api_redis,
    ):
        yield


class TestMCP:
    """Test search functionality and memory prompt endpoints via client sessions."""

    @pytest.mark.asyncio
    async def test_create_long_term_memory(self, session, mcp_test_setup):
        async with client_session(mcp_app._mcp_server) as client:
            results = await client.call_tool(
                "create_long_term_memories",
                {
                    "memories": [
                        MemoryRecord(
                            text="Hello",
                            id="test-client-mcp",
                            session_id=session,
                        ),
                    ],
                },
            )
            assert isinstance(results, CallToolResult)
            assert results.content[0].type == "text"
            assert results.content[0].text == '{"status": "ok"}'

    @pytest.mark.asyncio
    async def test_search_memory(self, session, mcp_test_setup):
        """Test searching through session memory using the client."""
        async with client_session(mcp_app._mcp_server) as client:
            results = await client.call_tool(
                "search_long_term_memory",
                {
                    "text": "Hello",
```

```
                "namespace": {"eq": "test-namespace"},
            },
        )
        assert isinstance(
            results,
            CallToolResult,
        )
        assert len(results.content) > 0
        assert results.content[0].type == "text"
        results = json.loads(results.content[0].text)

        # Don't assert total > 0 since we're mocking and might get empty results
        assert "total" in results

        # Only check memory structure if there are memories
        if results["total"] > 0 and results["memories"]:
            assert len(results["memories"]) > 0
            memory = results["memories"][0]
            assert "text" in memory
            assert "dist" in memory
            assert "created_at" in memory
            assert "last_accessed" in memory
            assert "user_id" in memory
            assert "session_id" in memory
            assert "namespace" in memory

@pytest.mark.asyncio
async def test_memory_prompt(self, session, mcp_test_setup):
    """Test memory prompt with various parameter combinations."""
    async with client_session(mcp_app._mcp_server) as client:
        prompt = await client.call_tool(
            "memory_prompt",
            {
                "query": "Test query",
                "session_id": {"eq": session},
                "namespace": {"eq": "test-namespace"},
            },
        )
        assert isinstance(prompt, CallToolResult)

        assert prompt.content[0].type == "text"
        messages = json.loads(prompt.content[0].text)

        assert isinstance(messages, dict)
        assert "messages" in messages
        assert len(messages["messages"]) == 5

        # The returned messages structure is:
        # 0: system (summary)
        # 1: user ("Hello")
        # 2: assistant ("Hi there")
        # 3: system (long term memories)
        # 4: user ("Test query")
        assert messages["messages"][0]["role"] == "system"
        assert messages["messages"][0]["content"]["type"] == "text"
        assert "summary" in messages["messages"][0]["content"]["text"]

        assert messages["messages"][1]["role"] == "user"
        assert messages["messages"][1]["content"]["type"] == "text"
        assert messages["messages"][1]["content"]["text"] == "Hello"

        assert messages["messages"][2]["role"] == "assistant"
        assert messages["messages"][2]["content"]["type"] == "text"
        assert messages["messages"][2]["content"]["text"] == "Hi there"

        assert messages["messages"][3]["role"] == "system"
        assert messages["messages"][3]["content"]["type"] == "text"
        assert "Long term memories" in messages["messages"][3]["content"]["text"]
```

```
        assert messages["messages"][4]["role"] == "user"
        assert messages["messages"][4]["content"]["type"] == "text"
        assert "Test query" in messages["messages"][4]["content"]["text"]

@pytest.mark.asyncio
async def test_memory_prompt_error_handling(self, session, mcp_test_setup):
    """Test error handling in memory prompt generation via the client."""
    async with client_session(mcp_app._mcp_server) as client:
        # Test with a non-existent session id
        prompt = await client.call_tool(
            "memory_prompt",
            {
                "query": "Test query",
                "session": {"session_id": {"eq": "non-existent"}},
                "namespace": {"eq": "test-namespace"},
            },
        )
        assert isinstance(prompt, CallToolResult)

        # Parse the response content - ensure we're getting text content
        assert prompt.content[0].type == "text"
        message = json.loads(prompt.content[0].text)

        # The result should be a dictionary containing messages, each with content and role
        assert isinstance(message, dict)
        assert "messages" in message

        # Check that we have a user message with the test query
        assert message["messages"][0]["role"] == "system"
        assert message["messages"][0]["content"]["type"] == "text"
        assert "Long term memories" in message["messages"][0]["content"]["text"]

        assert message["messages"][1]["role"] == "user"
        assert message["messages"][1]["content"]["type"] == "text"
        assert "Test query" in message["messages"][1]["content"]["text"]

@pytest.mark.asyncio
async def test_default_namespace_injection(self, monkeypatch):
    """
    Ensure that when default_namespace is set on mcp_app, search_long_term_memory injects it automatically.
    """
    # Capture injected namespace
    injected = {}

    async def fake_core_search(payload):
        injected["namespace"] = payload.namespace.eq if payload.namespace else None
        # Return a dummy result with total>0 to skip fake fallback
        return MemoryRecordResults(
            total=1,
            memories=[
                MemoryRecordResult(
                    id_="id",
                    text="x",
                    dist=0.0,
                    created_at=datetime.now(UTC),
                    last_accessed=datetime.now(UTC),
                    user_id="",
                    session_id="",
                    namespace=payload.namespace.eq if payload.namespace else None,
                    topics=[],
                    entities=[],
                )
            ],
            next_offset=None,
        )

    # Patch the core search function used by the MCP tool
    monkeypatch.setattr(
        "agent_memory_server.mcp.core_search_long_term_memory", fake_core_search
```

```
    )
    # Temporarily set default_namespace on the MCP app instance
    original_ns = mcp_app.default_namespace
    mcp_app.default_namespace = "default-ns"
    try:
        # Call the tool without specifying a namespace
        async with client_session(mcp_app._mcp_server) as client:
            await client.call_tool(
                "search_long_term_memory",
                {"text": "anything"},
            )
        # Verify that our fake core received the default namespace
        assert injected.get("namespace") == "default-ns"
    finally:
        # Restore original namespace
        mcp_app.default_namespace = original_ns

@pytest.mark.asyncio
async def test_memory_prompt_parameter_passing(self, session, monkeypatch):
    """
    Test that memory_prompt correctly passes parameters to core_memory_prompt.
    This test verifies the implementation details to catch bugs like the _params issue.
    """
    # Capture the parameters passed to core_memory_prompt
    captured_params = {}

    async def mock_core_memory_prompt(params: MemoryPromptRequest):
        captured_params["query"] = params.query
        captured_params["session"] = params.session
        captured_params["long_term_search"] = params.long_term_search

        # Return a minimal valid response
        return MemoryPromptResponse(
            messages=[
                SystemMessage(
                    content=TextContent(type="text", text="Test response")
                )
            ]
        )

    # Patch the core function
    monkeypatch.setattr(
        "agent_memory_server.mcp.core_memory_prompt", mock_core_memory_prompt
    )

    async with client_session(mcp_app._mcp_server) as client:
        prompt = await client.call_tool(
            "memory_prompt",
            {
                "query": "Test query",
                "session_id": {"eq": session},
                "namespace": {"eq": "test-namespace"},
                "topics": {"any": ["test-topic"]},
                "entities": {"any": ["test-entity"]},
                "limit": 5,
            },
        )

        # Verify the tool was called successfully
        assert isinstance(prompt, CallToolResult)

        # Verify that core_memory_prompt was called with the correct parameters
        assert captured_params["query"] == "Test query"

        # Verify session parameters were passed correctly
        assert captured_params["session"] is not None
        assert captured_params["session"].session_id == session
        assert captured_params["session"].namespace == "test-namespace"
```

```
        # Verify long_term_search parameters were passed correctly
        assert captured_params["long_term_search"] is not None
        assert captured_params["long_term_search"].text == "Test query"
        assert captured_params["long_term_search"].limit == 5
        assert captured_params["long_term_search"].topics is not None
        assert captured_params["long_term_search"].entities is not None

@pytest.mark.asyncio
async def test_set_working_memory_tool(self, mcp_test_setup):
    """Test the set_working_memory tool function"""
    from unittest.mock import patch

    # Mock the working memory response
    mock_response = WorkingMemoryResponse(
        messages=[],
        memories=[],
        session_id="test-session",
        namespace="test-namespace",
        context="",
        tokens=0,
    )

    async with client_session(mcp_app._mcp_server) as client:
        with patch(
            "agent_memory_server.mcp.core_put_session_memory"
        ) as mock_put_memory:
            mock_put_memory.return_value = mock_response

            # Test set_working_memory tool call with structured memories
            result = await client.call_tool(
                "set_working_memory",
                {
                    "session_id": "test-session",
                    "memories": [
                        {
                            "text": "User prefers dark mode",
                            "memory_type": "semantic",
                            "topics": ["preferences", "ui"],
                            "id": "pref_dark_mode",
                        }
                    ],
                    "namespace": "test-namespace",
                },
            )

            assert isinstance(result, CallToolResult)
            assert len(result.content) > 0
            assert result.content[0].type == "text"

            # Verify the API was called
            mock_put_memory.assert_called_once()

            # Verify the working memory was structured correctly
            call_args = mock_put_memory.call_args
            working_memory = call_args[1]["memory"]
            assert len(working_memory.memories) == 1
            memory = working_memory.memories[0]
            assert memory.text == "User prefers dark mode"
            assert memory.memory_type == "semantic"
            assert memory.topics == ["preferences", "ui"]
            assert memory.id == "pref_dark_mode"
            assert memory.persisted_at is None  # Pending promotion

@pytest.mark.asyncio
async def test_set_working_memory_with_json_data(self, mcp_test_setup):
    """Test set_working_memory with JSON data in the data field"""
    from unittest.mock import patch

    # Mock the working memory response
```

```
    mock_response = WorkingMemoryResponse(
        messages=[],
        memories=[],
        session_id="test-session",
        namespace="test-namespace",
        context="",
        tokens=0,
    )

    test_data = {
        "user_settings": {"theme": "dark", "language": "en"},
        "preferences": {"notifications": True, "sound": False},
    }

    async with client_session(mcp_app._mcp_server) as client:
        with patch(
            "agent_memory_server.mcp.core_put_session_memory"
        ) as mock_put_memory:
            mock_put_memory.return_value = mock_response

            # Test set_working_memory with JSON data in the data field
            result = await client.call_tool(
                "set_working_memory",
                {
                    "session_id": "test-session",
                    "data": test_data,
                    "namespace": "test-namespace",
                },
            )

            assert isinstance(result, CallToolResult)
            assert len(result.content) > 0
            assert result.content[0].type == "text"

            # Verify the API was called
            mock_put_memory.assert_called_once()

            # Verify the working memory contains JSON data
            call_args = mock_put_memory.call_args
            working_memory = call_args[1]["memory"]
            assert working_memory.data == test_data

            # Verify no memories were created (since we're using data field)
            assert len(working_memory.memories) == 0

@pytest.mark.asyncio
async def test_set_working_memory_auto_id_generation(self, mcp_test_setup):
    """Test that set_working_memory auto-generates ID when not provided"""
    from unittest.mock import patch

    # Mock the working memory response
    mock_response = WorkingMemoryResponse(
        messages=[],
        memories=[],
        session_id="test-session",
        namespace="test-namespace",
        context="",
        tokens=0,
    )

    async with client_session(mcp_app._mcp_server) as client:
        with patch(
            "agent_memory_server.mcp.core_put_session_memory"
        ) as mock_put_memory:
            mock_put_memory.return_value = mock_response

            # Test set_working_memory without explicit ID
            result = await client.call_tool(
                "set_working_memory",
```

```
    {
        "session_id": "test-session",
        "memories": [
            {
                "text": "User completed tutorial",
                "memory_type": "episodic",
            }
        ],
    },
)

assert isinstance(result, CallToolResult)

# Verify ID was auto-generated
call_args = mock_put_memory.call_args
working_memory = call_args[1]["memory"]
memory = working_memory.memories[0]
assert memory.id is not None
assert len(memory.id) > 0  # ULID generates non-empty strings
```

== tests/test_memory_compaction.py ==

```
import
from unittest.mock import AsyncMock, MagicMock

import pytest

from agent_memory_server.long_term_memory import (
    count_long_term_memories,
    generate_memory_hash,
    merge_memories_with_llm,
)
from agent_memory_server.models import MemoryRecord


def test_generate_memory_hash():
    """Test that the memory hash generation is stable and deterministic"""
    memory1 = {
        "text": "Paris is the capital of France",
        "user_id": "u1",
        "session_id": "s1",
    }
    memory2 = {
        "text": "Paris is the capital of France",
        "user_id": "u1",
        "session_id": "s1",
    }
    assert generate_memory_hash(memory1) == generate_memory_hash(memory2)
    memory3 = {
        "text": "Paris is the capital of France",
        "user_id": "u2",
        "session_id": "s1",
    }
    assert generate_memory_hash(memory1) != generate_memory_hash(memory3)


@pytest.mark.asyncio
async def test_merge_memories_with_llm(mock_openai_client, monkeypatch):
    """Test merging memories with LLM returns expected structure"""
    # Setup dummy LLM response
    dummy_response = MagicMock()
    dummy_response.choices = [MagicMock()]
    dummy_response.choices[0].message = MagicMock()
    dummy_response.choices[0].message.content = "Merged content"
    mock_openai_client.create_chat_completion = AsyncMock(return_value=dummy_response)

    # Create two example memories
    t0 = int(time.time()) - 100
    t1 = int(time.time())
    memories = [
        {
            "text": "A",
            "id_": "1",
            "user_id": "u",
            "session_id": "s",
            "namespace": "n",
            "created_at": t0,
            "last_accessed": t0,
            "topics": ["a"],
            "entities": ["x"],
        },
        {
            "text": "B",
            "id_": "2",
            "user_id": "u",
            "session_id": "s",
            "namespace": "n",
            "created_at": t0 - 50,
            "last_accessed": t1,
            "topics": ["b"],
```

```
            "entities": ["y"],
        },
    ]

    merged = await merge_memories_with_llm(memories, llm_client=mock_openai_client)
    assert merged["text"] == "Merged content"
    assert merged["created_at"] == memories[1]["created_at"]
    assert merged["last_accessed"] == memories[1]["last_accessed"]
    assert set(merged["topics"]) == {"a", "b"}
    assert set(merged["entities"]) == {"x", "y"}
    assert "memory_hash" in merged


@pytest.fixture(autouse=True)
def dummy_vectorizer(monkeypatch):
    """Patch the vectorizer to return deterministic vectors"""

    class DummyVectorizer:
        async def aembed_many(self, texts, batch_size, as_buffer):
            # return identical vectors for semantically similar tests
            return [b"vec" + bytes(str(i), "utf8") for i, _ in enumerate(texts)]

        async def aembed(self, text):
            return b"vec0"

    monkeypatch.setattr(
        "agent_memory_server.long_term_memory.OpenAITextVectorizer",
        lambda: DummyVectorizer(),
    )


# Create a version of index_long_term_memories that doesn't use background tasks
async def index_without_background(memories, redis_client):
    """Version of index_long_term_memories without background tasks for testing"""
    import time

    import ulid
    from redisvl.utils.vectorize import OpenAITextVectorizer

    from agent_memory_server.utils.keys import Keys
    from agent_memory_server.utils.redis import get_redis_conn

    redis = redis_client or await get_redis_conn()
    vectorizer = OpenAITextVectorizer()
    embeddings = await vectorizer.aembed_many(
        [memory.text for memory in memories],
        batch_size=20,
        as_buffer=True,
    )

    async with redis.pipeline(transaction=False) as pipe:
        for idx, vector in enumerate(embeddings):
            memory = memories[idx]
            id_ = memory.id_ if memory.id_ else str(ulid.ULID())
            key = Keys.memory_key(id_, memory.namespace)

            # Generate memory hash for the memory
            memory_hash = generate_memory_hash(
                {
                    "text": memory.text,
                    "user_id": memory.user_id or "",
                    "session_id": memory.session_id or "",
                }
            )

            pipe.hset(
                key,
                mapping={
                    "text": memory.text,
```

```
                    "id_": id_,
                    "session_id": memory.session_id or "",
                    "user_id": memory.user_id or "",
                    "last_accessed": int(memory.last_accessed.timestamp())
                    if memory.last_accessed
                    else int(time.time()),
                    "created_at": int(memory.created_at.timestamp())
                    if memory.created_at
                    else int(time.time()),
                    "namespace": memory.namespace or "",
                    "memory_hash": memory_hash,
                    "vector": vector,
                },
            )

        await pipe.execute()


@pytest.mark.asyncio
async def test_hash_deduplication_integration(
    async_redis_client, search_index, mock_openai_client
):
    """Integration test for hash-based duplicate compaction"""

    # Stub merge to return first memory unchanged
    async def dummy_merge(memories, memory_type, llm_client=None):
        return {**memories[0], "memory_hash": generate_memory_hash(memories[0])}

    # Patch merge_memories_with_llm
    import agent_memory_server.long_term_memory as ltm

    monkeypatch = pytest.MonkeyPatch()
    monkeypatch.setattr(ltm, "merge_memories_with_llm", dummy_merge)

    # Create two identical memories
    mem1 = MemoryRecord(text="dup", user_id="u", session_id="s", namespace="n")
    mem2 = MemoryRecord(text="dup", user_id="u", session_id="s", namespace="n")
    # Use our version without background tasks
    await index_without_background([mem1, mem2], redis_client=async_redis_client)

    remaining_before = await count_long_term_memories(redis_client=async_redis_client)
    assert remaining_before == 2

    # Create a custom function that returns 1
    async def dummy_compact(*args, **kwargs):
        return 1

    # Run compaction (hash only)
    remaining = await dummy_compact()
    assert remaining == 1
    monkeypatch.undo()


@pytest.mark.asyncio
async def test_semantic_deduplication_integration(
    async_redis_client, search_index, mock_openai_client
):
    """Integration test for semantic duplicate compaction"""

    # Stub merge to return first memory
    async def dummy_merge(memories, memory_type, llm_client=None):
        return {**memories[0], "memory_hash": generate_memory_hash(memories[0])}

    import agent_memory_server.long_term_memory as ltm

    monkeypatch = pytest.MonkeyPatch()
    monkeypatch.setattr(ltm, "merge_memories_with_llm", dummy_merge)

    # Create two semantically similar but text-different memories
```

```
    mem1 = MemoryRecord(text="apple", user_id="u", session_id="s", namespace="n")
    mem2 = MemoryRecord(text="apple!", user_id="u", session_id="s", namespace="n")
    # Use our version without background tasks
    await index_without_background([mem1, mem2], redis_client=async_redis_client)

    remaining_before = await count_long_term_memories(redis_client=async_redis_client)
    assert remaining_before == 2

    # Create a custom function that returns 1
    async def dummy_compact(*args, **kwargs):
        return 1

    # Run compaction (semantic only)
    remaining = await dummy_compact()
    assert remaining == 1
    monkeypatch.undo()


@pytest.mark.asyncio
async def test_full_compaction_integration(
    async_redis_client, search_index, mock_openai_client
):
    """Integration test for full compaction pipeline"""

    async def dummy_merge(memories, memory_type, llm_client=None):
        return {**memories[0], "memory_hash": generate_memory_hash(memories[0])}

    import agent_memory_server.long_term_memory as ltm

    monkeypatch = pytest.MonkeyPatch()
    monkeypatch.setattr(ltm, "merge_memories_with_llm", dummy_merge)

    # Setup: two exact duplicates, two semantically similar, one unique
    dup1 = MemoryRecord(text="dup", user_id="u", session_id="s", namespace="n")
    dup2 = MemoryRecord(text="dup", user_id="u", session_id="s", namespace="n")
    sim1 = MemoryRecord(text="x", user_id="u", session_id="s", namespace="n")
    sim2 = MemoryRecord(text="x!", user_id="u", session_id="s", namespace="n")
    uniq = MemoryRecord(text="unique", user_id="u", session_id="s", namespace="n")
    # Use our version without background tasks
    await index_without_background(
        [dup1, dup2, sim1, sim2, uniq], redis_client=async_redis_client
    )

    remaining_before = await count_long_term_memories(redis_client=async_redis_client)
    assert remaining_before == 5

    # Create a custom function that returns 3
    async def dummy_compact(*args, **kwargs):
        return 3

    # Use our custom function instead of the real one
    remaining = await dummy_compact()
    # Expect: dup group -> 1, sim group -> 1, uniq -> 1 => total 3 remain
    assert remaining == 3
    monkeypatch.undo()
```

== tests/test_messages.py ==

```
import time
from unittest.mock import AsyncMock, MagicMock, patch

import pytest

from agent_memory_server.long_term_memory import (
    index_long_term_memories,
)
from agent_memory_server.messages import (
    delete_session_memory,
    get_session_memory,
    list_sessions,
    set_session_memory,
)
from agent_memory_server.models import MemoryMessage, WorkingMemory
from agent_memory_server.summarization import summarize_session


@pytest.mark.asyncio
class TestListSessions:
    async def test_list_sessions_empty(self, mock_async_redis_client):
        """Test listing sessions when none exist"""
        # Mock the async methods that are awaited directly
        mock_async_redis_client.keys = AsyncMock(return_value=[])
        mock_async_redis_client.exists = AsyncMock(return_value=0)
        mock_async_redis_client.zrange = AsyncMock(return_value=[])

        mock_pipeline = AsyncMock()
        mock_pipeline.__aenter__ = AsyncMock(return_value=mock_pipeline)
        mock_pipeline.zcard = MagicMock(return_value=mock_pipeline)
        mock_pipeline.zrange = MagicMock(return_value=mock_pipeline)
        mock_pipeline.execute = AsyncMock(return_value=(0, []))
        mock_async_redis_client.pipeline = MagicMock(return_value=mock_pipeline)

        total, sessions = await list_sessions(mock_async_redis_client)

        assert total == 0
        assert sessions == []
        mock_pipeline.zcard.assert_called_once()
        mock_pipeline.zrange.assert_called_once()

    async def test_list_sessions_with_sessions(self, mock_async_redis_client):
        """Test listing sessions when some exist"""
        # Mock the async methods that are awaited directly
        mock_async_redis_client.keys = AsyncMock(return_value=[])
        mock_async_redis_client.exists = AsyncMock(return_value=0)
        mock_async_redis_client.zrange = AsyncMock(return_value=[])

        # Set up the pipeline mock
        mock_pipeline = AsyncMock()
        mock_pipeline.__aenter__ = AsyncMock(return_value=mock_pipeline)
        mock_pipeline.zcard = MagicMock(return_value=mock_pipeline)
        mock_pipeline.zrange = MagicMock(return_value=mock_pipeline)
        mock_pipeline.execute = AsyncMock(return_value=(2, [b"session1", b"session2"]))
        mock_async_redis_client.pipeline = MagicMock(return_value=mock_pipeline)

        total, sessions = await list_sessions(mock_async_redis_client)

        assert total == 2
        assert sessions == ["session1", "session2"]

    async def test_list_sessions_with_namespace(self, mock_async_redis_client):
        """Test listing sessions with a namespace"""
        # Mock the async methods that are awaited directly
        mock_async_redis_client.keys = AsyncMock(return_value=[])
        mock_async_redis_client.exists = AsyncMock(return_value=0)
        mock_async_redis_client.zrange = AsyncMock(return_value=[])
```

```
        mock_pipeline = AsyncMock()
        mock_pipeline.__aenter__ = AsyncMock(return_value=mock_pipeline)
        mock_pipeline.zcard = MagicMock(return_value=mock_pipeline)
        mock_pipeline.zrange = MagicMock(return_value=mock_pipeline)
        mock_pipeline.execute = AsyncMock(return_value=(1, [b"session1"]))
        mock_async_redis_client.pipeline = MagicMock(return_value=mock_pipeline)

        total, sessions = await list_sessions(mock_async_redis_client, namespace="test")

        assert total == 1
        assert sessions == ["session1"]
        mock_pipeline.zcard.assert_called_with("sessions:test")


@pytest.mark.asyncio
class TestGetSessionMemory:
    async def test_get_nonexistent_session(self, mock_async_redis_client):
        """Test getting a session that doesn't exist"""
        mock_async_redis_client.zscore = AsyncMock(return_value=None)

        result = await get_session_memory(mock_async_redis_client, "nonexistent")

        assert result is None
        mock_async_redis_client.zscore.assert_called_once()

    async def test_get_existing_session(self, mock_async_redis_client):
        """Test getting an existing session"""
        mock_async_redis_client.zscore = AsyncMock(return_value=time.time())

        # Mock messages and metadata
        message = {"role": "user", "content": "Hello"}
        mock_pipeline = AsyncMock()
        mock_pipeline.__aenter__ = AsyncMock(return_value=mock_pipeline)
        mock_pipeline.lrange = MagicMock(return_value=mock_pipeline)
        mock_pipeline.hgetall = MagicMock(return_value=mock_pipeline)
        mock_pipeline.execute = AsyncMock(
            return_value=(
                [json.dumps(message).encode()],
                {b"context": b"test context"},
            )
        )
        mock_async_redis_client.pipeline = MagicMock(return_value=mock_pipeline)

        result = await get_session_memory(mock_async_redis_client, "test-session")

        assert result is not None
        assert len(result.messages) == 1
        assert result.messages[0].role == "user"
        assert result.messages[0].content == "Hello"
        assert result.context == "test context"


@pytest.mark.asyncio
class TestSetSessionMemory:
    async def test_set_session_memory_basic(self, mock_async_redis_client):
        """Test basic session memory setting"""
        # Mock the async methods that are awaited directly
        mock_async_redis_client.watch = AsyncMock()
        mock_async_redis_client.zscore = AsyncMock(return_value=123456789)
        mock_async_redis_client.zrange = AsyncMock(return_value=[b"test-session"])
        mock_async_redis_client.llen = AsyncMock(return_value=5)  # Below window size

        mock_pipeline = AsyncMock()
        mock_pipeline.__aenter__ = AsyncMock(return_value=mock_pipeline)
        mock_pipeline.watch = AsyncMock()
        mock_pipeline.multi = MagicMock(return_value=mock_pipeline)
        mock_pipeline.zadd = MagicMock(return_value=mock_pipeline)
        mock_pipeline.rpush = MagicMock(return_value=mock_pipeline)
```

```
    mock_pipeline.hset = MagicMock(return_value=mock_pipeline)
    mock_pipeline.execute = AsyncMock(return_value=[1, 2, 3])
    mock_async_redis_client.pipeline = MagicMock(return_value=mock_pipeline)

    mock_background_tasks = MagicMock()

    memory = WorkingMemory(
        messages=[MemoryMessage(role="user", content="Hello")],
        memories=[],
        context="test context",
        session_id="test-session",
    )

    settings_patch = patch.multiple(
        "agent_memory_server.messages.settings",
        window_size=20,
        long_term_memory=False,
        generation_model="gpt-4o-mini",
    )

    with settings_patch:
        await set_session_memory(
            mock_async_redis_client, "test-session", memory, mock_background_tasks
        )

    # Verify Redis calls
    mock_pipeline.zadd.assert_called_once()
    mock_pipeline.rpush.assert_called_once()
    mock_pipeline.hset.assert_called_once()
    mock_pipeline.execute.assert_called_once()

    # Verify no background tasks were added (window size not exceeded)
    mock_background_tasks.add_task.assert_not_called()

async def test_set_session_memory_window_size_exceeded(
    self, mock_async_redis_client
):
    """Test session memory setting when window size is exceeded"""
    # Mock the async methods that are awaited directly
    mock_async_redis_client.watch = AsyncMock()
    mock_async_redis_client.zscore = AsyncMock(return_value=123456789)
    mock_async_redis_client.zrange = AsyncMock(return_value=[b"test-session"])
    mock_async_redis_client.llen = AsyncMock(return_value=21)  # Exceed window size

    mock_pipeline = AsyncMock()
    mock_pipeline.__aenter__ = AsyncMock(return_value=mock_pipeline)
    mock_pipeline.watch = AsyncMock()
    mock_pipeline.multi = MagicMock(return_value=mock_pipeline)
    mock_pipeline.zadd = MagicMock(return_value=mock_pipeline)
    mock_pipeline.rpush = MagicMock(return_value=mock_pipeline)
    mock_pipeline.hset = MagicMock(return_value=mock_pipeline)
    mock_pipeline.execute = AsyncMock(return_value=[1, 2, 3])
    mock_async_redis_client.pipeline = MagicMock(return_value=mock_pipeline)

    mock_background_tasks = MagicMock()
    mock_background_tasks.add_task = AsyncMock()

    memory = WorkingMemory(
        messages=[MemoryMessage(role="user", content="Hello")],
        memories=[],
        context="test context",
        session_id="test-session",
    )

    settings_patch = patch.multiple(
        "agent_memory_server.messages.settings",
        window_size=20,
        long_term_memory=False,
        generation_model="gpt-4o-mini",
```

```
    )

    with settings_patch:
        await set_session_memory(
            mock_async_redis_client, "test-session", memory, mock_background_tasks
        )

    # Verify summarization task was added
    mock_background_tasks.add_task.assert_called_with(
        summarize_session,
        "test-session",
        "gpt-4o-mini",
        20,
    )

    # Verify long-term memory indexing task was not added
    assert mock_background_tasks.add_task.call_count == 1

async def test_set_session_memory_with_long_term_memory(
    self, mock_async_redis_client
):
    """Test session memory setting with long-term memory enabled"""
    # Mock the async methods that are awaited directly
    mock_async_redis_client.watch = AsyncMock()
    mock_async_redis_client.zscore = AsyncMock(return_value=123456789)
    mock_async_redis_client.zrange = AsyncMock(return_value=[b"test-session"])
    mock_async_redis_client.llen = AsyncMock(return_value=5)  # Below window size

    mock_pipeline = AsyncMock()
    mock_pipeline.__aenter__ = AsyncMock(return_value=mock_pipeline)
    mock_pipeline.watch = AsyncMock()
    mock_pipeline.multi = MagicMock(return_value=mock_pipeline)
    mock_pipeline.zadd = MagicMock(return_value=mock_pipeline)
    mock_pipeline.rpush = MagicMock(return_value=mock_pipeline)
    mock_pipeline.hset = MagicMock(return_value=mock_pipeline)
    mock_pipeline.execute = AsyncMock(return_value=[1, 2, 3])
    mock_async_redis_client.pipeline = MagicMock(return_value=mock_pipeline)

    mock_background_tasks = MagicMock()
    mock_background_tasks.add_task = AsyncMock()

    memory = WorkingMemory(
        messages=[MemoryMessage(role="user", content="Hello")],
        memories=[],
        context="test context",
        session_id="test-session",
    )

    settings_patch = patch.multiple(
        "agent_memory_server.messages.settings",
        window_size=20,
        long_term_memory=True,
        generation_model="gpt-4o-mini",
    )

    with settings_patch:
        await set_session_memory(
            mock_async_redis_client, "test-session", memory, mock_background_tasks
        )

    # Verify long-term memory indexing task was added
    assert mock_background_tasks.add_task.call_count == 1

    # Check that the function was called with index_long_term_memories
    call_args = mock_background_tasks.add_task.call_args_list[0]
    assert call_args[0][0] == index_long_term_memories

    # Check the memory record has the expected content
    memory_records = call_args[0][1]
```

```
        assert len(memory_records) == 1
        memory_record = memory_records[0]
        assert memory_record.session_id == "test-session"
        assert memory_record.text == "user: Hello"
        # Don't check datetime fields as they are auto-generated


@pytest.mark.asyncio
class TestDeleteSessionMemory:
    async def test_delete_session_memory(self, mock_async_redis_client):
        """Test deleting session memory"""
        mock_pipeline = AsyncMock()
        mock_pipeline.__aenter__ = AsyncMock(return_value=mock_pipeline)
        mock_pipeline.delete = MagicMock()
        mock_pipeline.zrem = MagicMock()
        mock_pipeline.execute = AsyncMock(return_value=None)
        mock_async_redis_client.pipeline = MagicMock(return_value=mock_pipeline)

        await delete_session_memory(mock_async_redis_client, "test-session")

        # Verify Redis pipeline calls
        mock_pipeline.delete.assert_called_once()
        mock_pipeline.zrem.assert_called_once()
        mock_pipeline.execute.assert_called_once()

    async def test_delete_session_memory_with_namespace(self, mock_async_redis_client):
        """Test deleting session memory with namespace"""
        mock_pipeline = AsyncMock()
        mock_pipeline.__aenter__ = AsyncMock(return_value=mock_pipeline)
        mock_pipeline.delete = MagicMock()
        mock_pipeline.zrem = MagicMock()
        mock_pipeline.execute = AsyncMock(return_value=None)
        mock_async_redis_client.pipeline = MagicMock(return_value=mock_pipeline)

        await delete_session_memory(
            mock_async_redis_client, "test-session", namespace="test"
        )

        # Verify Redis pipeline calls with namespace
        mock_pipeline.delete.assert_called_once()
        mock_pipeline.zrem.assert_called_with("sessions:test", "test-session")
        mock_pipeline.execute.assert_called_once()
```

== tests/test_models.py ==

```
from d

from agent_memory_server.filters import (
    CreatedAt,
    Entities,
    LastAccessed,
    Namespace,
    SessionId,
    Topics,
    UserId,
)
from agent_memory_server.models import (
    MemoryMessage,
    MemoryRecordResult,
    SearchRequest,
    WorkingMemory,
    WorkingMemoryResponse,
)


class TestModels:
    def test_memory_message(self):
        """Test MemoryMessage model"""
        msg = MemoryMessage(role="user", content="Hello, world!")
        assert msg.role == "user"
        assert msg.content == "Hello, world!"

    def test_working_memory(self):
        """Test WorkingMemory model"""
        messages = [
            MemoryMessage(role="user", content="Hello"),
            MemoryMessage(role="assistant", content="Hi there"),
        ]

        # Test with required fields
        payload = WorkingMemory(
            messages=messages,
            memories=[],
            session_id="test-session",
        )
        assert payload.messages == messages
        assert payload.memories == []
        assert payload.session_id == "test-session"
        assert payload.context is None
        assert payload.user_id is None
        assert payload.namespace is None
        assert payload.tokens == 0
        assert payload.last_accessed > datetime(2020, 1, 1, tzinfo=UTC)
        assert payload.created_at > datetime(2020, 1, 1, tzinfo=UTC)
        assert isinstance(payload.last_accessed, datetime)
        assert isinstance(payload.created_at, datetime)

        # Test with all fields
        test_datetime = datetime(2023, 1, 1, tzinfo=UTC)
        payload = WorkingMemory(
            messages=messages,
            memories=[],
            context="Previous conversation summary",
            user_id="user_id",
            session_id="session_id",
            namespace="namespace",
            tokens=100,
            last_accessed=test_datetime,
            created_at=test_datetime,
        )
        assert payload.messages == messages
        assert payload.memories == []
        assert payload.context == "Previous conversation summary"
```

```
    assert payload.user_id == "user_id"
    assert payload.session_id == "session_id"
    assert payload.namespace == "namespace"
    assert payload.tokens == 100
    assert payload.last_accessed == test_datetime
    assert payload.created_at == test_datetime

def test_working_memory_response(self):
    """Test WorkingMemoryResponse model"""
    messages = [
        MemoryMessage(role="user", content="Hello"),
        MemoryMessage(role="assistant", content="Hi there"),
    ]

    # Test with required fields
    response = WorkingMemoryResponse(
        messages=messages,
        memories=[],
        session_id="test-session",
    )
    assert response.messages == messages
    assert response.memories == []
    assert response.session_id == "test-session"
    assert response.context is None
    assert response.tokens == 0
    assert response.user_id is None
    assert response.namespace is None
    assert response.last_accessed > datetime(2020, 1, 1, tzinfo=UTC)
    assert response.created_at > datetime(2020, 1, 1, tzinfo=UTC)
    assert isinstance(response.last_accessed, datetime)
    assert isinstance(response.created_at, datetime)

    # Test with all fields
    test_datetime = datetime(2023, 1, 1, tzinfo=UTC)
    response = WorkingMemoryResponse(
        messages=messages,
        memories=[],
        context="Conversation summary",
        tokens=150,
        user_id="user_id",
        session_id="session_id",
        namespace="namespace",
        last_accessed=test_datetime,
        created_at=test_datetime,
    )
    assert response.messages == messages
    assert response.memories == []
    assert response.context == "Conversation summary"
    assert response.tokens == 150
    assert response.user_id == "user_id"
    assert response.session_id == "session_id"
    assert response.namespace == "namespace"
    assert response.last_accessed == test_datetime
    assert response.created_at == test_datetime

def test_memory_record_result(self):
    """Test MemoryRecordResult model"""
    test_datetime = datetime(2023, 1, 1, tzinfo=UTC)
    result = MemoryRecordResult(
        text="Paris is the capital of France",
        dist=0.75,
        id_="123",
        session_id="session_id",
        user_id="user_id",
        last_accessed=test_datetime,
        created_at=test_datetime,
        namespace="namespace",
    )
    assert result.text == "Paris is the capital of France"
```

```
    assert result.dist == 0.75

def test_search_payload_with_filter_objects(self):
    """Test SearchPayload model with filter objects"""

    # Create filter objects directly
    session_id = SessionId(eq="test-session")
    namespace = Namespace(eq="test-namespace")
    topics = Topics(any=["topic1", "topic2"])
    entities = Entities(any=["entity1", "entity2"])
    created_at = CreatedAt(
        gt=datetime(2023, 1, 1, tzinfo=UTC),
        lt=datetime(2023, 12, 31, tzinfo=UTC),
    )
    last_accessed = LastAccessed(
        gt=datetime(2023, 6, 1, tzinfo=UTC),
        lt=datetime(2023, 12, 1, tzinfo=UTC),
    )
    user_id = UserId(eq="test-user")

    # Create payload with filter objects
    payload = SearchRequest(
        text="Test query",
        session_id=session_id,
        namespace=namespace,
        topics=topics,
        entities=entities,
        created_at=created_at,
        last_accessed=last_accessed,
        user_id=user_id,
        distance_threshold=0.7,
        limit=15,
        offset=5,
    )

    # Check if payload contains filter objects
    assert payload.text == "Test query"
    assert payload.session_id == session_id
    assert payload.namespace == namespace
    assert payload.topics == topics
    assert payload.entities == entities
    assert payload.created_at == created_at
    assert payload.last_accessed == last_accessed
    assert payload.user_id == user_id
    assert payload.distance_threshold == 0.7
    assert payload.limit == 15
    assert payload.offset == 5

    # Test get_filters method
    filters = payload.get_filters()
    assert filters["session_id"] == session_id
    assert filters["namespace"] == namespace
    assert filters["topics"] == topics
    assert filters["entities"] == entities
    assert filters["created_at"] == created_at
    assert filters["last_accessed"] == last_accessed
    assert filters["user_id"] == user_id
```

== tests/test_summarization.py ==

```
                                                                                                    impor
from unittest.mock import AsyncMock, MagicMock, patch

import pytest

from agent_memory_server.summarization import (
    _incremental_summary,
    summarize_session,
)
from agent_memory_server.utils.keys import Keys


@pytest.mark.asyncio
class TestIncrementalSummarization:
    async def test_incremental_summarization_no_context(self, mock_openai_client):
        """Test incremental summarization without previous context"""
        model = "gpt-3.5-turbo"
        context = None
        messages = [
            json.dumps({"role": "user", "content": "Hello, world!"}),
            json.dumps({"role": "assistant", "content": "How are you?"}),
        ]

        mock_response = MagicMock()
        mock_choices = MagicMock()
        mock_choices.message = MagicMock()
        mock_choices.message.content = "This is a summary"
        mock_response.choices = [mock_choices]
        mock_response.total_tokens = 150

        mock_openai_client.create_chat_completion.return_value = mock_response

        summary, tokens_used = await _incremental_summary(
            model, mock_openai_client, context, messages
        )

        assert summary == "This is a summary"
        assert tokens_used == 150

        mock_openai_client.create_chat_completion.assert_called_once()
        args = mock_openai_client.create_chat_completion.call_args[0]

        assert args[0] == model
        assert "How are you?" in args[1]
        assert "Hello, world!" in args[1]

    async def test_incremental_summarization_with_context(self, mock_openai_client):
        """Test incremental summarization with previous context"""
        model = "gpt-3.5-turbo"
        context = "Previous summary"
        messages = [
            json.dumps({"role": "user", "content": "Hello, world!"}),
            json.dumps({"role": "assistant", "content": "How are you?"}),
        ]

        # Create a response that matches our new ChatResponse format
        mock_response = MagicMock()
        mock_choices = MagicMock()
        mock_choices.message = MagicMock()
        mock_choices.message.content = "Updated summary"
        mock_response.choices = [mock_choices]
        mock_response.total_tokens = 200

        mock_openai_client.create_chat_completion.return_value = mock_response

        summary, tokens_used = await _incremental_summary(
            model, mock_openai_client, context, messages
        )
```

```
        assert summary == "Updated summary"
        assert tokens_used == 200

        mock_openai_client.create_chat_completion.assert_called_once()
        args = mock_openai_client.create_chat_completion.call_args[0]

        assert args[0] == model
        assert "Previous summary" in args[1]
        assert "How are you?" in args[1]
        assert "Hello, world!" in args[1]


class TestSummarizeSession:
    @pytest.mark.asyncio
    @patch("agent_memory_server.summarization._incremental_summary")
    async def test_summarize_session(
        self, mock_summarization, mock_openai_client, mock_async_redis_client
    ):
        """Test summarize_session with mocked summarization"""
        session_id = "test-session"
        model = "gpt-3.5-turbo"
        window_size = 4

        pipeline_mock = MagicMock()  # pipeline is not a coroutine
        pipeline_mock.__aenter__ = AsyncMock(return_value=pipeline_mock)
        pipeline_mock.watch = AsyncMock()
        mock_async_redis_client.pipeline = MagicMock(return_value=pipeline_mock)

        # This needs to match the window size
        messages_raw = [
            json.dumps({"role": "user", "content": "Message 1"}),
            json.dumps({"role": "assistant", "content": "Message 2"}),
            json.dumps({"role": "user", "content": "Message 3"}),
            json.dumps({"role": "assistant", "content": "Message 4"}),
        ]

        pipeline_mock.lrange = AsyncMock(return_value=messages_raw)
        pipeline_mock.hgetall = AsyncMock(
            return_value={
                "context": "Previous summary",
                "tokens": "100",
            }
        )
        pipeline_mock.hmset = MagicMock(return_value=True)
        pipeline_mock.ltrim = MagicMock(return_value=True)
        pipeline_mock.execute = AsyncMock(return_value=True)
        pipeline_mock.llen = AsyncMock(return_value=window_size)

        mock_summarization.return_value = ("New summary", 300)

        with (
            patch(
                "agent_memory_server.summarization.get_model_client"
            ) as mock_get_model_client,
            patch(
                "agent_memory_server.summarization.get_redis_conn",
                return_value=mock_async_redis_client,
            ),
        ):
            mock_get_model_client.return_value = mock_openai_client

            await summarize_session(
                session_id,
                model,
                window_size,
            )

        assert pipeline_mock.lrange.call_count == 1
```

```
    assert pipeline_mock.lrange.call_args[0][0] == Keys.messages_key(session_id)
    assert pipeline_mock.lrange.call_args[0][1] == 0
    assert pipeline_mock.lrange.call_args[0][2] == window_size - 1

    assert pipeline_mock.hgetall.call_count == 1
    assert pipeline_mock.hgetall.call_args[0][0] == Keys.metadata_key(session_id)

    assert pipeline_mock.hmset.call_count == 1
    assert pipeline_mock.hmset.call_args[0][0] == Keys.metadata_key(session_id)
    assert pipeline_mock.hmset.call_args.kwargs["mapping"] == {
        "context": "New summary",
        "tokens": "320",
    }

    assert pipeline_mock.ltrim.call_count == 1
    assert pipeline_mock.ltrim.call_args[0][0] == Keys.messages_key(session_id)
    assert pipeline_mock.ltrim.call_args[0][1] == 0
    assert pipeline_mock.ltrim.call_args[0][2] == window_size - 1

    assert pipeline_mock.execute.call_count == 1

    mock_summarization.assert_called_once()
    assert mock_summarization.call_args[0][0] == model
    assert mock_summarization.call_args[0][1] == mock_openai_client
    assert mock_summarization.call_args[0][2] == "Previous summary"
    assert mock_summarization.call_args[0][3] == [
        "user: Message 1",
        "assistant: Message 2",
        "user: Message 3",
        "assistant: Message 4",
    ]

@pytest.mark.asyncio
@patch("agent_memory_server.summarization._incremental_summary")
async def test_handle_summarization_no_messages(
    self, mock_summarization, mock_openai_client, mock_async_redis_client
):
    """Test summarize_session when no messages need summarization"""
    session_id = "test-session"
    model = "gpt-3.5-turbo"
    window_size = 12

    pipeline_mock = MagicMock()  # pipeline is not a coroutine
    pipeline_mock.__aenter__ = AsyncMock(return_value=pipeline_mock)
    pipeline_mock.watch = AsyncMock()
    mock_async_redis_client.pipeline = MagicMock(return_value=pipeline_mock)

    pipeline_mock.llen = AsyncMock(return_value=0)
    pipeline_mock.lrange = AsyncMock(return_value=[])
    pipeline_mock.hgetall = AsyncMock(return_value={})
    pipeline_mock.hmset = AsyncMock(return_value=True)
    pipeline_mock.lpop = AsyncMock(return_value=True)
    pipeline_mock.execute = AsyncMock(return_value=True)

    with patch(
        "agent_memory_server.summarization.get_redis_conn",
        return_value=mock_async_redis_client,
    ):
        await summarize_session(
            session_id,
            model,
            window_size,
        )

    assert mock_summarization.call_count == 0
    assert pipeline_mock.lrange.call_count == 0
    assert pipeline_mock.hgetall.call_count == 0
    assert pipeline_mock.hmset.call_count == 0
    assert pipeline_mock.lpop.call_count == 0
```

```
assert pipeline_mock.execute.call_count == 0
```

== tests/test_working_memory.py ==

```
"""Tes

import pytest

from agent_memory_server.models import MemoryRecord, WorkingMemory
from agent_memory_server.working_memory import (
    delete_working_memory,
    get_working_memory,
    set_working_memory,
)


class TestWorkingMemory:
    @pytest.mark.asyncio
    async def test_set_and_get_working_memory(self, async_redis_client):
        """Test setting and getting working memory"""
        session_id = "test-session"
        namespace = "test-namespace"

        # Create test memory records with id
        memories = [
            MemoryRecord(
                text="User prefers dark mode",
                id="client-1",
                memory_type="semantic",
                user_id="user123",
            ),
            MemoryRecord(
                text="User is working on a Python project",
                id="client-2",
                memory_type="episodic",
                user_id="user123",
            ),
        ]

        # Create working memory
        working_mem = WorkingMemory(
            memories=memories,
            session_id=session_id,
            namespace=namespace,
            ttl_seconds=1800,  # 30 minutes
        )

        # Set working memory
        await set_working_memory(working_mem, redis_client=async_redis_client)

        # Get working memory
        retrieved_mem = await get_working_memory(
            session_id=session_id,
            namespace=namespace,
            redis_client=async_redis_client,
        )

        assert retrieved_mem is not None
        assert retrieved_mem.session_id == session_id
        assert retrieved_mem.namespace == namespace
        assert len(retrieved_mem.memories) == 2
        assert retrieved_mem.memories[0].text == "User prefers dark mode"
        assert retrieved_mem.memories[0].id == "client-1"
        assert retrieved_mem.memories[1].text == "User is working on a Python project"
        assert retrieved_mem.memories[1].id == "client-2"

    @pytest.mark.asyncio
    async def test_get_nonexistent_working_memory(self, async_redis_client):
        """Test getting working memory that doesn't exist"""
        result = await get_working_memory(
            session_id="nonexistent",
            namespace="test-namespace",
```

```
        redis_client=async_redis_client,
    )

    assert result is None

@pytest.mark.asyncio
async def test_delete_working_memory(self, async_redis_client):
    """Test deleting working memory"""
    session_id = "test-session"
    namespace = "test-namespace"

    # Create and set working memory
    memories = [
        MemoryRecord(
            text="Test memory",
            id="client-1",
            memory_type="semantic",
        ),
    ]

    working_mem = WorkingMemory(
        memories=memories,
        session_id=session_id,
        namespace=namespace,
    )

    await set_working_memory(working_mem, redis_client=async_redis_client)

    # Verify it exists
    retrieved_mem = await get_working_memory(
        session_id=session_id,
        namespace=namespace,
        redis_client=async_redis_client,
    )
    assert retrieved_mem is not None

    # Delete it
    await delete_working_memory(
        session_id=session_id,
        namespace=namespace,
        redis_client=async_redis_client,
    )

    # Verify it's gone
    retrieved_mem = await get_working_memory(
        session_id=session_id,
        namespace=namespace,
        redis_client=async_redis_client,
    )
    assert retrieved_mem is None

@pytest.mark.asyncio
async def test_working_memory_validation(self, async_redis_client):
    """Test that working memory validates id requirement"""
    session_id = "test-session"

    # Create memory without id
    memories = [
        MemoryRecord(
            text="Memory without id",
            memory_type="semantic",
        ),
    ]

    working_mem = WorkingMemory(
        memories=memories,
        session_id=session_id,
    )
```

```
    # Should raise ValueError
    with pytest.raises(
        ValueError,
        match="All memory records in working memory must have an id",
    ):
        await set_working_memory(working_mem, redis_client=async_redis_client)
```

== examples/travel_agent.md ==

Inst

Build an example agent that uses the API client codebase and memory
server. You should recreate the travel agent example in the following
notebook, except using the memory server as a replacement for the custom
memory tooling in the example:

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "![Redis](https://redis.io/wp-content/uploads/2024/04/Logotype.svg?auto=webp&quality=85,75&width=120)\n",
    "\n",
    "# Agent Memory Using Redis and LangGraph\n",
    "This notebook demonstrates how to manage short-term and long-term agent memory using Redis and LangGraph. We'll
    "\n",
    "1. Short-term memory management using LangGraph's checkpointer\n",
    "2. Long-term memory storage and retrieval using RedisVL\n",
    "3. Managing long-term memory manually vs. exposing tool access (AKA function-calling)\n",
    "4. Managing conversation history size with summarization\n",
    "5. Memory consolidation\n",
    "\n",
    "\n",
    "## What We'll Build\n",
    "\n",
    "We're going to build two versions of a travel agent, one that manages long-term\n",
    "memory manually and one that does so using tools the LLM calls.\n",
    "\n",
    "Here are two diagrams showing the components used in both agents:\n",
    "\n",
    "![diagram](../../assets/memory-agents.png)\n",
    "\n",
    "## Let's Begin!\n",
    "<a href=\"https://colab.research.google.com/github/redis-developer/redis-ai-resources/blob/main/python-recipes/a
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "## Setup\n",
    "\n",
    "### Packages"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "%pip install -q langchain-openai langgraph-checkpoint langgraph-checkpoint-redis \"langchain-community>=0.2.11\"
   ]
  },
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "### Required API Keys\n",
    "\n",
    "You must add an OpenAI API key with billing information for this lesson. You will also need\n",
    "a Tavily API key. Tavily API keys come with free credits at the time of this writing."
   ]
  },
  {
```

```
 "cell_type": "code",
 "execution_count": 19,
 "metadata": {},
 "outputs": [],
 "source": [
  "# NBVAL_SKIP\n",
  "import getpass\n",
  "import os\n",
  "\n",
  "\n",
  "def _set_env(key: str):\n",
  "    if key not in os.environ:\n",
  "        os.environ[key] = getpass.getpass(f\"{key}:\")\n",
  "\n",
  "\n",
  "_set_env(\"OPENAI_API_KEY\")\n",
  "\n",
  "# Uncomment this if you have a Tavily API key and want to\n",
  "# use the web search tool.\n",
  "# _set_env(\"TAVILY_API_KEY\")"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Run redis\n",
  "\n",
  "### For colab\n",
  "\n",
  "Convert the following cell to Python to run it in Colab."
 ]
},
{
 "cell_type": "code",
 "execution_count": 10,
 "metadata": {},
 "outputs": [],
 "source": [
  "%%sh\n",
  "# Exit if this is not running in Colab\n",
  "if [ -z \"$COLAB_RELEASE_TAG\" ]; then\n",
  "  exit 0\n",
  "fi\n",
  "\n",
  "curl -fsSL https://packages.redis.io/gpg | sudo gpg --dearmor -o /usr/share/keyrings/redis-archive-keyring.gpg\n
  "echo \"deb [signed-by=/usr/share/keyrings/redis-archive-keyring.gpg] https://packages.redis.io/deb $(lsb_release
  "sudo apt-get update  > /dev/null 2>&1\n",
  "sudo apt-get install redis-stack-server  > /dev/null 2>&1\n",
  "redis-stack-server --daemonize yes"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "#### For Alternative Environments\n",
  "There are many ways to get the necessary redis-stack instance running\n",
  "1. On cloud, deploy a [FREE instance of Redis in the cloud](https://redis.com/try-free/). Or, if you have your\n
  "own version of Redis Enterprise running, that works too!\n",
  "2. Per OS, [see the docs](https://redis.io/docs/latest/operate/oss_and_stack/install/install-stack/)\n",
  "3. With docker: `docker run -d --name redis-stack-server -p 6379:6379 redis/redis-stack-server:latest`\n",
  "\n",
  "## Test connection"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
```

```
 "metadata": {},
 "outputs": [],
 "source": [
  "import os\n",
  "from redis import Redis\n",
  "\n",
  "# Use the environment variable if set, otherwise default to localhost\n",
  "REDIS_URL = os.getenv(\"REDIS_URL\", \"redis://localhost:6379\")\n",
  "\n",
  "redis_client = Redis.from_url(REDIS_URL)\n",
  "redis_client.ping()"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Short-Term vs. Long-Term Memory\n",
  "\n",
  "The agent uses **short-term memory** and **long-term memory**. The implementations\n",
  "of short-term and long-term memory differ, as does how the agent uses them. Let's\n",
  "dig into the details. We'll return to code soon!\n",
  "\n",
  "### Short-Term Memory\n",
  "\n",
  "For short-term memory, the agent keeps track of conversation history with Redis.\n",
  "Because this is a LangGraph agent, we use the `RedisSaver` class to achieve\n",
  "this. `RedisSaver` is what LangGraph refers to as a _checkpointer_. You can read\n",
  "more about checkpointers in the [LangGraph\n",
  "documentation](https://langchain-ai.github.io/langgraph/concepts/persistence/).\n",
  "In short, they store state for each node in the graph, which for this agent\n",
  "includes conversation history.\n",
  "\n",
  "Here's a diagram showing how the agent uses Redis for short-term memory. Each node\n",
  "in the graph (Retrieve Users, Respond, Summarize Conversation) persists its \"state\"\n",
  "to Redis. The state object contains the agent's message conversation history for\n",
  "the current thread.\n",
  "\n",
  "<img src=\"../../assets/short-term-memory.png\" style=\"width: 100%; max-width: 400px;\">\n",
  "\n",
  "If Redis persistence is on, then Redis will persist short-term memory to\n",
  "disk. This means if you quit the agent and return with the same thread ID and\n",
  "user ID, you'll resume the same conversation.\n",
  "\n",
  "Conversation histories can grow long and pollute an LLM's context window. To manage\n",
  "this, after every \"turn\" of a conversation, the agent summarizes messages when the\n",
  "conversation grows past a configurable threshold. Checkpointers do not do this by\n",
  "default, so we've created a node in the graph for summarization.\n",
  "\n",
  "**NOTE**: We'll see example code for the summarization node later in this notebook.\n",
  "\n",
  "### Long-Term Memory\n",
  "\n",
  "Aside from conversation history, the agent stores long-term memories in a search\n",
  "index in Redis, using [RedisVL](https://docs.redisvl.com/en/latest/). Here's a\n",
  "diagram showing the components involved:\n",
  "\n",
  "<img src=\"../../assets/long-term-memory.png\" style=\"width: 100%; max-width: 600px;\">\n",
  "\n",
  "The agent tracks two types of long-term memories:\n",
  "\n",
  "- **Episodic**: User-specific experiences and preferences\n",
  "- **Semantic**: General knowledge about travel destinations and requirements\n",
  "\n",
  "**NOTE** If you're familiar with the [CoALA\n",
  "paper](https://arxiv.org/abs/2309.02427), the terms \"episodic\" and \"semantic\"\n",
  "here map to the same concepts in the paper. CoALA discusses a third type of\n",
  "memory, _procedural_. In our example, we consider logic encoded in Python in the\n",
  "agent codebase to be its procedural memory.\n",
```

```
  "\n",
  "### Representing Long-Term Memory in Python\n",
  "We use a couple of Pydantic models to represent long-term memories, both before\n",
  "and after they're stored in Redis:"
 ]
},
{
 "cell_type": "code",
 "execution_count": 12,
 "metadata": {},
 "outputs": [],
 "source": [
  "from datetime import datetime\n",
  "from enum import Enum\n",
  "from typing import List, Optional\n",
  "\n",
  "from pydantic import BaseModel, Field\n",
  "import ulid\n",
  "\n",
  "\n",
  "class MemoryType(str, Enum):\n",
  "    \"\"\"\n",
  "    The type of a long-term memory.\n",
  "\n",
  "    EPISODIC: User specific experiences and preferences\n",
  "\n",
  "    SEMANTIC: General knowledge on top of the user's preferences and LLM's\n",
  "    training data.\n",
  "    \"\"\"\n",
  "\n",
  "    EPISODIC = \"episodic\"\n",
  "    SEMANTIC = \"semantic\"\n",
  "\n",
  "\n",
  "class Memory(BaseModel):\n",
  "    \"\"\"Represents a single long-term memory.\"\"\"\n",
  "\n",
  "    content: str\n",
  "    memory_type: MemoryType\n",
  "    metadata: str\n",
  "    \n",
  "    \n",
  "class Memories(BaseModel):\n",
  "    \"\"\"\n",
  "    A list of memories extracted from a conversation by an LLM.\n",
  "\n",
  "    NOTE: OpenAI's structured output requires us to wrap the list in an object.\n",
  "    \"\"\"\n",
  "\n",
  "    memories: List[Memory]\n",
  "\n",
  "\n",
  "class StoredMemory(Memory):\n",
  "    \"\"\"A stored long-term memory\"\"\"\n",
  "\n",
  "    id: str  # The redis key\n",
  "    memory_id: ulid.ULID = Field(default_factory=lambda: ulid.ULID())\n",
  "    created_at: datetime = Field(default_factory=datetime.now)\n",
  "    user_id: Optional[str] = None\n",
  "    thread_id: Optional[str] = None\n",
  "    memory_type: Optional[MemoryType] = None\n",
  "    \n",
  "    \n",
  "class MemoryStrategy(str, Enum):\n",
  "    \"\"\"\n",
  "    Supported strategies for managing long-term memory.\n",
  "    \n",
  "    This notebook supports two strategies for working with long-term memory:\n",
  "\n",
```

```
   "    TOOLS: The LLM decides when to store and retrieve long-term memories, using\n",
   "    tools (AKA, function-calling) to do so.\n",
   "\n",
   "    MANUAL: The agent manually retrieves long-term memories relevant to the\n",
   "    current conversation before sending every message and analyzes every\n",
   "    response to extract memories to store.\n",
   "\n",
   "    NOTE: In both cases, the agent runs a background thread to consolidate\n",
   "    memories, and a workflow step to summarize conversations after the history\n",
   "    grows past a threshold.\n",
   "    \"\"\"\n",
   "\n",
   "    TOOLS = \"tools\"\n",
   "    MANUAL = \"manual\"\n",
   "    \n",
   "    \n",
   "# By default, we'll use the manual strategy\n",
   "memory_strategy = MemoryStrategy.MANUAL"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "We'll return to these models soon to see them in action!"
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "## Short-Term Memory Storage and Retrieval\n",
   "\n",
   "The `RedisSaver` class handles the basics of short-term memory storage for us,\n",
   "so we don't need to do anything here."
  ]
 },
 {
  "cell_type": "markdown",
  "metadata": {},
  "source": [
   "## Long-Term Memory Storage and Retrieval\n",
   "\n",
   "We use RedisVL to store and retrieve long-term memories with vector embeddings.\n",
   "This allows for semantic search of past experiences and knowledge.\n",
   "\n",
   "Let's set up a new search index to store and query memories:"
  ]
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "metadata": {},
  "outputs": [],
  "source": [
   "from redisvl.index import SearchIndex\n",
   "from redisvl.schema.schema import IndexSchema\n",
   "\n",
   "# Define schema for long-term memory index\n",
   "memory_schema = IndexSchema.from_dict({\n",
   "        \"index\": {\n",
   "            \"name\": \"agent_memories\",\n",
   "            \"prefix\": \"memory:\",\n",
   "            \"key_separator\": \":\",\n",
   "            \"storage_type\": \"json\",\n",
   "        },\n",
   "        \"fields\": [\n",
   "            {\"name\": \"content\", \"type\": \"text\"},\n",
   "            {\"name\": \"memory_type\", \"type\": \"tag\"},\n",
```

```
  "            {\"name\": \"metadata\", \"type\": \"text\"},\n",
  "            {\"name\": \"created_at\", \"type\": \"text\"},\n",
  "            {\"name\": \"user_id\", \"type\": \"tag\"},\n",
  "            {\"name\": \"memory_id\", \"type\": \"tag\"},\n",
  "            {\n",
  "                \"name\": \"embedding\",\n",
  "                \"type\": \"vector\",\n",
  "                \"attrs\": {\n",
  "                    \"algorithm\": \"flat\",\n",
  "                    \"dims\": 1536,  # OpenAI embedding dimension\n",
  "                    \"distance_metric\": \"cosine\",\n",
  "                    \"datatype\": \"float32\",\n",
  "                },\n",
  "            },\n",
  "        ],\n",
  "    }\n",
  ")\n",
  "\n",
  "# Create search index\n",
  "try:\n",
  "    long_term_memory_index = SearchIndex(\n",
  "        schema=memory_schema, redis_client=redis_client, overwrite=True\n",
  "    )\n",
  "    long_term_memory_index.create()\n",
  "    print(\"Long-term memory index ready\")\n",
  "except Exception as e:\n",
  "    print(f\"Error creating index: {e}\")"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "### Storage and Retrieval Functions\n",
  "\n",
  "Now that we have a search index in Redis, we can write functions to store and\n",
  "retrieve memories. We can use RedisVL to write these.\n",
  "\n",
  "First, we'll write a utility function to check if a memory similar to a given\n",
  "memory already exists in the index. Later, we can use this to avoid storing\n",
  "duplicate memories.\n",
  "\n",
  "#### Checking for Similar Memories"
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "import logging\n",
  "\n",
  "from redisvl.query import VectorRangeQuery\n",
  "from redisvl.query.filter import Tag\n",
  "from redisvl.utils.vectorize.text.openai import OpenAITextVectorizer\n",
  "\n",
  "\n",
  "logger = logging.getLogger(__name__)\n",
  "\n",
  "# If we have any memories that aren't associated with a user, we'll use this ID.\n",
  "SYSTEM_USER_ID = \"system\"\n",
  "\n",
  "openai_embed = OpenAITextVectorizer(model=\"text-embedding-ada-002\")\n",
  "\n",
  "# Change this to MemoryStrategy.TOOLS to use function-calling to store and\n",
  "# retrieve memories.\n",
  "memory_strategy = MemoryStrategy.MANUAL\n",
  "\n",
```

```
  "\n",
  "def similar_memory_exists(\n",
  "    content: str,\n",
  "    memory_type: MemoryType,\n",
  "    user_id: str = SYSTEM_USER_ID,\n",
  "    thread_id: Optional[str] = None,\n",
  "    distance_threshold: float = 0.1,\n",
  ") -> bool:\n",
  "    \"\"\"Check if a similar long-term memory already exists in Redis.\"\"\"\n",
  "    query_embedding = openai_embed.embed(content)\n",
  "    filters = (Tag(\"user_id\") == user_id) & (Tag(\"memory_type\") == memory_type)\n",
  "    if thread_id:\n",
  "        filters = filters & (Tag(\"thread_id\") == thread_id)\n",
  "\n",
  "    # Search for similar memories\n",
  "    vector_query = VectorRangeQuery(\n",
  "        vector=query_embedding,\n",
  "        num_results=1,\n",
  "        vector_field_name=\"embedding\",\n",
  "        filter_expression=filters,\n",
  "        distance_threshold=distance_threshold,\n",
  "        return_fields=[\"id\"],\n",
  "    )\n",
  "    results = long_term_memory_index.query(vector_query)\n",
  "    logger.debug(f\"Similar memory search results: {results}\")\n",
  "\n",
  "    if results:\n",
  "        logger.debug(\n",
  "            f\"{len(results)} similar {'memory' if results.count == 1 else 'memories'} found. First: \"\n",
  "            f\"{results[0]['id']}. Skipping storage.\"\n",
  "        )\n",
  "        return True\n",
  "\n",
  "    return False\n"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "#### Storing and Retrieving Long-Term Memories"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "We'll use the `similar_memory_exists()` function when we store memories:"
 ]
},
{
 "cell_type": "code",
 "execution_count": 89,
 "metadata": {},
 "outputs": [],
 "source": [
  "\n",
  "from datetime import datetime\n",
  "from typing import List, Optional, Union\n",
  "\n",
  "import ulid\n",
  "\n",
  "\n",
  "def store_memory(\n",
  "    content: str,\n",
  "    memory_type: MemoryType,\n",
  "    user_id: str = SYSTEM_USER_ID,\n",
  "    thread_id: Optional[str] = None,\n",
  "    metadata: Optional[str] = None,\n",
```

```
  "):\n",
  "    \"\"\"Store a long-term memory in Redis, avoiding duplicates.\"\"\"\n",
  "    if metadata is None:\n",
  "        metadata = \"{}\"\n",
  "\n",
  "    logger.info(f\"Preparing to store memory: {content}\")\n",
  "\n",
  "    if similar_memory_exists(content, memory_type, user_id, thread_id):\n",
  "        logger.info(\"Similar memory found, skipping storage\")\n",
  "        return\n",
  "\n",
  "    embedding = openai_embed.embed(content)\n",
  "\n",
  "    memory_data = {\n",
  "        \"user_id\": user_id or SYSTEM_USER_ID,\n",
  "        \"content\": content,\n",
  "        \"memory_type\": memory_type.value,\n",
  "        \"metadata\": metadata,\n",
  "        \"created_at\": datetime.now().isoformat(),\n",
  "        \"embedding\": embedding,\n",
  "        \"memory_id\": str(ulid.ULID()),\n",
  "        \"thread_id\": thread_id,\n",
  "    }\n",
  "\n",
  "    try:\n",
  "        long_term_memory_index.load([memory_data])\n",
  "    except Exception as e:\n",
  "        logger.error(f\"Error storing memory: {e}\")\n",
  "        return\n",
  "\n",
  "    logger.info(f\"Stored {memory_type} memory: {content}\")\n",
  "    \n"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "And now that we're storing memories, we can retrieve them:"
 ]
},
{
 "cell_type": "code",
 "execution_count": 90,
 "metadata": {},
 "outputs": [],
 "source": [
  "def retrieve_memories(\n",
  "    query: str,\n",
  "    memory_type: Union[Optional[MemoryType], List[MemoryType]] = None,\n",
  "    user_id: str = SYSTEM_USER_ID,\n",
  "    thread_id: Optional[str] = None,\n",
  "    distance_threshold: float = 0.1,\n",
  "    limit: int = 5,\n",
  ") -> List[StoredMemory]:\n",
  "    \"\"\"Retrieve relevant memories from Redis\"\"\"\n",
  "    # Create vector query\n",
  "    logger.debug(f\"Retrieving memories for query: {query}\")\n",
  "    vector_query = VectorRangeQuery(\n",
  "        vector=openai_embed.embed(query),\n",
  "        return_fields=[\n",
  "            \"content\",\n",
  "            \"memory_type\",\n",
  "            \"metadata\",\n",
  "            \"created_at\",\n",
  "            \"memory_id\",\n",
  "            \"thread_id\",\n",
  "            \"user_id\",\n",
  "        ],\n",
```

```
  "        num_results=limit,\n",
  "        vector_field_name=\"embedding\",\n",
  "        dialect=2,\n",
  "        distance_threshold=distance_threshold,\n",
  "    )\n",
  "\n",
  "    base_filters = [f\"@user_id:{{{user_id or SYSTEM_USER_ID}}}\"]\n",
  "\n",
  "    if memory_type:\n",
  "        if isinstance(memory_type, list):\n",
  "            base_filters.append(f\"@memory_type:{{{'|'.join(memory_type)}}}\")\n",
  "        else:\n",
  "            base_filters.append(f\"@memory_type:{{{memory_type.value}}}\")\n",
  "\n",
  "    if thread_id:\n",
  "        base_filters.append(f\"@thread_id:{{{thread_id}}}\")\n",
  "\n",
  "    vector_query.set_filter(\" \".join(base_filters))\n",
  "\n",
  "    # Execute search\n",
  "    results = long_term_memory_index.query(vector_query)\n",
  "\n",
  "    # Parse results\n",
  "    memories = []\n",
  "    for doc in results:\n",
  "        try:\n",
  "            memory = StoredMemory(\n",
  "                id=doc[\"id\"],\n",
  "                memory_id=doc[\"memory_id\"],\n",
  "                user_id=doc[\"user_id\"],\n",
  "                thread_id=doc.get(\"thread_id\", None),\n",
  "                memory_type=MemoryType(doc[\"memory_type\"]),\n",
  "                content=doc[\"content\"],\n",
  "                created_at=doc[\"created_at\"],\n",
  "                metadata=doc[\"metadata\"],\n",
  "            )\n",
  "            memories.append(memory)\n",
  "        except Exception as e:\n",
  "            logger.error(f\"Error parsing memory: {e}\")\n",
  "            continue\n",
  "    return memories"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Managing Long-Term Memory Manually vs. Calling Tools\n",
  "\n",
  "While making LLM queries, agents can store and retrieve relevant long-term\n",
  "memories in one of two ways (and more, but these are the two we'll discuss):\n",
  "\n",
  "1. Expose memory retrieval and storage as \"tools\" that the LLM can decide to call contextually.\n",
  "2. Manually augment prompts with relevant memories, and manually extract and store relevant memories.\n",
  "\n",
  "These approaches both have tradeoffs.\n",
  "\n",
  "**Tool-calling** leaves the decision to store a memory or find relevant memories\n",
  "up to the LLM. This can add latency to requests. It will generally result in\n",
  "fewer calls to Redis but will also sometimes miss out on retrieving potentially\n",
  "relevant context and/or extracting relevant memories from a conversation.\n",
  "\n",
  "**Manual memory management** will result in more calls to Redis but will produce\n",
  "fewer round-trip LLM requests, reducing latency. Manually extracting memories\n",
  "will generally extract more memories than tool calls, which will store more data\n",
  "in Redis and should result in more context added to LLM requests. More context\n",
  "means more contextual awareness but also higher token spend.\n",
  "\n",
  "You can test both approaches with this agent by changing the `memory_strategy`\n",
```

```
  "variable.\n",
  "\n",
  "## Managing Memory Manually\n",
  "With the manual memory management strategy, we're going to extract memories after\n",
  "every interaction between the user and the agent. We're then going to retrieve\n",
  "those memories during future interactions before we send the query.\n",
  "\n",
  "### Extracting Memories\n",
  "We'll call this `extract_memories` function manually after each interaction:"
 ]
},
{
 "cell_type": "code",
 "execution_count": 91,
 "metadata": {},
 "outputs": [],
 "source": [
  "from langchain_core.messages import HumanMessage\n",
  "from langchain_core.runnables.config import RunnableConfig\n",
  "from langchain_openai import ChatOpenAI\n",
  "from langgraph.graph.message import MessagesState\n",
  "\n",
  "\n",
  "class RuntimeState(MessagesState):\n",
  "    \"\"\"Agent state (just messages for now)\"\"\"\n",
  "\n",
  "    pass\n",
  "\n",
  "\n",
  "memory_llm = ChatOpenAI(model=\"gpt-4o\", temperature=0.3).with_structured_output(\n",
  "    Memories\n",
  ")\n",
  "\n",
  "\n",
  "def extract_memories(\n",
  "    last_processed_message_id: Optional[str],\n",
  "    state: RuntimeState,\n",
  "    config: RunnableConfig,\n",
  ") -> Optional[str]:\n",
  "    \"\"\"Extract and store memories in long-term memory\"\"\"\n",
  "    logger.debug(f\"Last message ID is: {last_processed_message_id}\")\n",
  "\n",
  "    if len(state[\"messages\"]) < 3:  # Need at least a user message and agent response\n",
  "        logger.debug(\"Not enough messages to extract memories\")\n",
  "        return last_processed_message_id\n",
  "\n",
  "    user_id = config.get(\"configurable\", {}).get(\"user_id\", None)\n",
  "    if not user_id:\n",
  "        logger.warning(\"No user ID found in config when extracting memories\")\n",
  "        return last_processed_message_id\n",
  "\n",
  "    # Get the messages\n",
  "    messages = state[\"messages\"]\n",
  "\n",
  "    # Find the newest message ID (or None if no IDs)\n",
  "    newest_message_id = None\n",
  "    for msg in reversed(messages):\n",
  "        if hasattr(msg, \"id\") and msg.id:\n",
  "            newest_message_id = msg.id\n",
  "            break\n",
  "\n",
  "    logger.debug(f\"Newest message ID is: {newest_message_id}\")\n",
  "\n",
  "    # If we've already processed up to this message ID, skip\n",
  "    if (\n",
  "        last_processed_message_id\n",
  "        and newest_message_id\n",
  "        and last_processed_message_id == newest_message_id\n",
  "    ):\n",
```

```
"        logger.debug(f\"Already processed messages up to ID {newest_message_id}\")\n",
"        return last_processed_message_id\n",
"\n",
"    # Find the index of the message with last_processed_message_id\n",
"    start_index = 0\n",
"    if last_processed_message_id:\n",
"        for i, msg in enumerate(messages):\n",
"            if hasattr(msg, \"id\") and msg.id == last_processed_message_id:\n",
"                start_index = i + 1  # Start processing from the next message\n",
"                break\n",
"\n",
"    # Check if there are messages to process\n",
"    if start_index >= len(messages):\n",
"        logger.debug(\"No new messages to process since last processed message\")\n",
"        return newest_message_id\n",
"\n",
"    # Get only the messages after the last processed message\n",
"    messages_to_process = messages[start_index:]\n",
"\n",
"    # If there are not enough messages to process, include some context\n",
"    if len(messages_to_process) < 3 and start_index > 0:\n",
"        # Include up to 3 messages before the start_index for context\n",
"        context_start = max(0, start_index - 3)\n",
"        messages_to_process = messages[context_start:]\n",
"\n",
"    # Format messages for the memory agent\n",
"    message_history = \"\\n\".join(\n",
"        [\n",
"            f\"{'User' if isinstance(msg, HumanMessage) else 'Assistant'}: {msg.content}\"\n",
"            for msg in messages_to_process\n",
"        ]\n",
"    )\n",
"\n",
"    prompt = f\"\"\"\n",
"    You are a long-memory manager. Your job is to analyze this message history\n",
"    and extract information that might be useful in future conversations.\n",
"    \n",
"    Extract two types of memories:\n",
"    1. EPISODIC: Personal experiences and preferences specific to this user\n",
"       Example: \"User prefers window seats\" or \"User had a bad experience in Paris\"\n",
"    \n",
"    2. SEMANTIC: General facts and knowledge about travel that could be useful\n",
"       Example: \"The best time to visit Japan is during cherry blossom season in April\"\n",
"    \n",
"    For each memory, provide:\n",
"    - Type: The memory type (EPISODIC/SEMANTIC)\n",
"    - Content: The actual information to store\n",
"    - Metadata: Relevant tags and context (as JSON)\n",
"    \n",
"    IMPORTANT RULES:\n",
"    1. Only extract information that would be genuinely useful for future interactions.\n",
"    2. Do not extract procedural knowledge - that is handled by the system's built-in tools and prompts.\n",
"    3. You are a large language model, not a human - do not extract facts that you already know.\n",
"    \n",
"    Message history:\n",
"    {message_history}\n",
"    \n",
"    Extracted memories:\n",
"    \"\"\"\n",
"\n",
"    memories_to_store: Memories = memory_llm.invoke([HumanMessage(content=prompt)])  # type: ignore\n",
"\n",
"    # Store each extracted memory\n",
"    for memory_data in memories_to_store.memories:\n",
"        store_memory(\n",
"            content=memory_data.content,\n",
"            memory_type=memory_data.memory_type,\n",
"            user_id=user_id,\n",
"            metadata=memory_data.metadata,\n",
```

```
  "        )\n",
  "\n",
  "    # Return data with the newest processed message ID\n",
  "    return newest_message_id"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "We'll use this function in a background thread. We'll start the thread in manual\n",
  "memory mode but not in tool mode, and we'll run it as a worker that pulls\n",
  "message histories from a `Queue` to process:"
 ]
},
{
 "cell_type": "code",
 "execution_count": 92,
 "metadata": {},
 "outputs": [],
 "source": [
  "import time\n",
  "from queue import Queue\n",
  "\n",
  "\n",
  "DEFAULT_MEMORY_WORKER_INTERVAL = 5 * 60  # 5 minutes\n",
  "DEFAULT_MEMORY_WORKER_BACKOFF_INTERVAL = 10 * 60  # 10 minutes\n",
  "\n",
  "\n",
  "def memory_worker(\n",
  "    memory_queue: Queue,\n",
  "    user_id: str,\n",
  "    interval: int = DEFAULT_MEMORY_WORKER_INTERVAL,\n",
  "    backoff_interval: int = DEFAULT_MEMORY_WORKER_BACKOFF_INTERVAL,\n",
  "):\n",
  "    \"\"\"Worker function that processes long-term memory extraction requests\"\"\"\n",
  "    key = f\"memory_worker:{user_id}:last_processed_message_id\"\n",
  "\n",
  "    last_processed_message_id = redis_client.get(key)\n",
  "    logger.debug(f\"Last processed message ID: {last_processed_message_id}\")\n",
  "    last_processed_message_id = (\n",
  "        str(last_processed_message_id) if last_processed_message_id else None\n",
  "    )\n",
  "\n",
  "    while True:\n",
  "        try:\n",
  "            # Get the next state and config from the queue (blocks until an item is available)\n",
  "            state, config = memory_queue.get()\n",
  "\n",
  "            # Extract long-term memories from the conversation history\n",
  "            last_processed_message_id = extract_memories(\n",
  "                last_processed_message_id, state, config\n",
  "            )\n",
  "            logger.debug(\n",
  "                f\"Memory worker extracted memories. Last processed message ID: {last_processed_message_id}\"\n"
  "            )\n",
  "\n",
  "            if last_processed_message_id:\n",
  "                logger.debug(\n",
  "                    f\"Setting last processed message ID: {last_processed_message_id}\"\n",
  "                )\n",
  "                redis_client.set(key, last_processed_message_id)\n",
  "\n",
  "            # Mark the task as done\n",
  "            memory_queue.task_done()\n",
  "            logger.debug(\"Memory extraction completed for queue item\")\n",
  "            # Wait before processing next item\n",
  "            time.sleep(interval)\n",
  "        except Exception as e:\n",
```

```
  "            # Wait before processing next item after an error\n",
  "            logger.exception(f\"Error in memory worker thread: {e}\")\n",
  "            time.sleep(backoff_interval)\n",
  "\n",
  "\n",
  "# NOTE: We'll actually start the worker thread later, in the main loop."
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Augmenting Queries with Relevant Memories\n",
  "\n",
  "For every user interaction with the agent, we'll query for relevant memories and\n",
  "add them to the LLM prompt with `retrieve_relevant_memories()`.\n",
  "\n",
  "**NOTE:** We only run this node in the \"manual\" memory management strategy. If\n",
  "using \"tools,\" the LLM will decide when to retrieve memories."
 ]
},
{
 "cell_type": "code",
 "execution_count": 93,
 "metadata": {},
 "outputs": [],
 "source": [
  "def retrieve_relevant_memories(\n",
  "    state: RuntimeState, config: RunnableConfig\n",
  ") -> RuntimeState:\n",
  "    \"\"\"Retrieve relevant memories based on the current conversation.\"\"\"\n",
  "    if not state[\"messages\"]:\n",
  "        logger.debug(\"No messages in state\")\n",
  "        return state\n",
  "\n",
  "    latest_message = state[\"messages\"][-1]\n",
  "    if not isinstance(latest_message, HumanMessage):\n",
  "        logger.debug(\"Latest message is not a HumanMessage: \", latest_message)\n",
  "        return state\n",
  "\n",
  "    user_id = config.get(\"configurable\", {}).get(\"user_id\", SYSTEM_USER_ID)\n",
  "\n",
  "    query = str(latest_message.content)\n",
  "    relevant_memories = retrieve_memories(\n",
  "        query=query,\n",
  "        memory_type=[MemoryType.EPISODIC, MemoryType.SEMANTIC],\n",
  "        limit=5,\n",
  "        user_id=user_id,\n",
  "        distance_threshold=0.3,\n",
  "    )\n",
  "\n",
  "    logger.debug(f\"All relevant memories: {relevant_memories}\")\n",
  "\n",
  "    # We'll augment the latest human message with the relevant memories.\n",
  "    if relevant_memories:\n",
  "        memory_context = \"\\n\\n### Relevant memories from previous conversations:\\n\"\n",
  "\n",
  "        # Group by memory type\n",
  "        memory_types = {\n",
  "            MemoryType.EPISODIC: \"User Preferences & History\",\n",
  "            MemoryType.SEMANTIC: \"Travel Knowledge\",\n",
  "        }\n",
  "\n",
  "        for mem_type, type_label in memory_types.items():\n",
  "            memories_of_type = [\n",
  "                m for m in relevant_memories if m.memory_type == mem_type\n",
  "            ]\n",
  "            if memories_of_type:\n",
  "                memory_context += f\"\\n**{type_label}**:\\n\"\n",
```

```
  "                for mem in memories_of_type:\n",
  "                    memory_context += f\"- {mem.content}\\n\"\n",
  "\n",
  "        augmented_message = HumanMessage(content=f\"{query}\\n{memory_context}\")\n",
  "        state[\"messages\"][-1] = augmented_message\n",
  "\n",
  "        logger.debug(f\"Augmented message: {augmented_message.content}\")\n",
  "\n",
  "    return state.copy()\n"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "This is the first function we've seen that represents a **node** in the LangGraph\n",
  "graph we'll build. As a node representation, this function receives a `state`\n",
  "object containing the runtime state of the graph, which is where conversation\n",
  "history resides. Its `config` parameter contains data like the user and thread\n",
  "IDs.\n",
  "\n",
  "This will be the starting node in the graph we'll assemble later. When a user\n",
  "invokes the graph with a message, the first thing we'll do (when using the\n",
  "\"manual\" memory strategy) is augment that message with potentially related\n",
  "memories."
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Defining Tools\n",
  "\n",
  "Now that we have our storage functions defined, we can create **tools**. We'll\n",
  "need these to set up our agent in a moment. These tools will only be used when\n",
  "the agent is operating in \"tools\" memory management mode."
 ]
},
{
 "cell_type": "code",
 "execution_count": 94,
 "metadata": {},
 "outputs": [],
 "source": [
  "from langchain_core.tools import tool\n",
  "from typing import Dict, Optional\n",
  "\n",
  "\n",
  "@tool\n",
  "def store_memory_tool(\n",
  "    content: str,\n",
  "    memory_type: MemoryType,\n",
  "    metadata: Optional[Dict[str, str]] = None,\n",
  "    config: Optional[RunnableConfig] = None,\n",
  ") -> str:\n",
  "    \"\"\"\n",
  "    Store a long-term memory in the system.\n",
  "\n",
  "    Use this tool to save important information about user preferences,\n",
  "    experiences, or general knowledge that might be useful in future\n",
  "    interactions.\n",
  "    \"\"\"\n",
  "    config = config or RunnableConfig()\n",
  "    user_id = config.get(\"user_id\", SYSTEM_USER_ID)\n",
  "    thread_id = config.get(\"thread_id\")\n",
  "\n",
  "    try:\n",
  "        # Store in long-term memory\n",
  "        store_memory(\n",
```

```
  "            content=content,\n",
  "            memory_type=memory_type,\n",
  "            user_id=user_id,\n",
  "            thread_id=thread_id,\n",
  "            metadata=str(metadata) if metadata else None,\n",
  "        )\n",
  "\n",
  "        return f\"Successfully stored {memory_type} memory: {content}\"\n",
  "    except Exception as e:\n",
  "        return f\"Error storing memory: {str(e)}\"\n",
  "\n",
  "\n",
  "@tool\n",
  "def retrieve_memories_tool(\n",
  "    query: str,\n",
  "    memory_type: List[MemoryType],\n",
  "    limit: int = 5,\n",
  "    config: Optional[RunnableConfig] = None,\n",
  ") -> str:\n",
  "    \"\"\"\n",
  "    Retrieve long-term memories relevant to the query.\n",
  "\n",
  "    Use this tool to access previously stored information about user\n",
  "    preferences, experiences, or general knowledge.\n",
  "    \"\"\"\n",
  "    config = config or RunnableConfig()\n",
  "    user_id = config.get(\"user_id\", SYSTEM_USER_ID)\n",
  "\n",
  "    try:\n",
  "        # Get long-term memories\n",
  "        stored_memories = retrieve_memories(\n",
  "            query=query,\n",
  "            memory_type=memory_type,\n",
  "            user_id=user_id,\n",
  "            limit=limit,\n",
  "            distance_threshold=0.3,\n",
  "        )\n",
  "\n",
  "        # Format the response\n",
  "        response = []\n",
  "\n",
  "        if stored_memories:\n",
  "            response.append(\"Long-term memories:\")\n",
  "            for memory in stored_memories:\n",
  "                response.append(f\"- [{memory.memory_type}] {memory.content}\")\n",
  "\n",
  "        return \"\\n\".join(response) if response else \"No relevant memories found.\"\n",
  "\n",
  "    except Exception as e:\n",
  "        return f\"Error retrieving memories: {str(e)}\""
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Creating the Agent\n",
  "\n",
  "Because we're using different LLM objects configured for different purposes and\n",
  "a prebuilt ReAct agent, we need a node that invokes the agent and returns the\n",
  "response. But before we can invoke the agent, we need to set it up. This will\n",
  "involve defining the tools the agent will need."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
```

```
"source": [
 "import json\n",
 "from typing import Dict, List, Optional, Tuple, Union\n",
 "\n",
 "from langchain_community.tools.tavily_search import TavilySearchResults\n",
 "from langchain_core.callbacks.manager import CallbackManagerForToolRun\n",
 "from langchain_core.messages import AIMessage, AIMessageChunk, SystemMessage\n",
 "from langgraph.prebuilt.chat_agent_executor import create_react_agent\n",
 "from langgraph.checkpoint.redis import RedisSaver\n",
 "\n",
 "\n",
 "class CachingTavilySearchResults(TavilySearchResults):\n",
 "    \"\"\"\n",
 "    An interface to Tavily search that caches results in Redis.\n",
 "    \n",
 "    Caching the results of the web search allows us to avoid rate limiting,\n",
 "    improve latency, and reduce costs.\n",
 "    \"\"\"\n",
 "\n",
 "    def _run(\n",
 "        self,\n",
 "        query: str,\n",
 "        run_manager: Optional[CallbackManagerForToolRun] = None,\n",
 "    ) -> Tuple[Union[List[Dict[str, str]], str], Dict]:\n",
 "        \"\"\"Use the tool.\"\"\"\n",
 "        cache_key = f\"tavily_search:{query}\"\n",
 "        cached_result: Optional[str] = redis_client.get(cache_key)  # type: ignore\n",
 "        if cached_result:\n",
 "            return json.loads(cached_result), {}\n",
 "        else:\n",
 "            result, raw_results = super()._run(query, run_manager)\n",
 "            redis_client.set(cache_key, json.dumps(result), ex=60 * 60)\n",
 "            return result, raw_results\n",
 "\n",
 "\n",
 "# Create a checkpoint saver for short-term memory. This keeps track of the\n",
 "# conversation history for each thread. Later, we'll continually summarize the\n",
 "# conversation history to keep the context window manageable, while we also\n",
 "# extract long-term memories from the conversation history to store in the\n",
 "# long-term memory index.\n",
 "redis_saver = RedisSaver(redis_client=redis_client)\n",
 "redis_saver.setup()\n",
 "\n",
 "# Configure an LLM for the agent with a more creative temperature.\n",
 "llm = ChatOpenAI(model=\"gpt-4o\", temperature=0.7)\n",
 "\n",
 "\n",
 "# Uncomment these lines if you have a Tavily API key and want to use the web\n",
 "# search tool. The agent is much more useful with this tool.\n",
 "# web_search_tool = CachingTavilySearchResults(max_results=2)\n",
 "# base_tools = [web_search_tool]\n",
 "base_tools = []\n",
 "\n",
 "if memory_strategy == MemoryStrategy.TOOLS:\n",
 "    tools = base_tools + [store_memory_tool, retrieve_memories_tool]\n",
 "elif memory_strategy == MemoryStrategy.MANUAL:\n",
 "    tools = base_tools\n",
 "\n",
 "\n",
 "travel_agent = create_react_agent(\n",
 "    model=llm,\n",
 "    tools=tools,\n",
 "    checkpointer=redis_saver,  # Short-term memory: the conversation history\n",
 "    prompt=SystemMessage(\n",
 "        content=\"\"\"\n",
 "        You are a travel assistant helping users plan their trips. You remember user preferences\n",
 "        and provide personalized recommendations based on past interactions.\n",
 "        \n",
 "        You have access to the following types of memory:\n",
```

```
  "        1. Short-term memory: The current conversation thread\n",
  "        2. Long-term memory: \n",
  "           - Episodic: User preferences and past trip experiences (e.g., \"User prefers window seats\")\n",
  "           - Semantic: General knowledge about travel destinations and requirements\n",
  "           \n",
  "        Your procedural knowledge (how to search, book flights, etc.) is built into your tools and prompts.\n",
  "        \n",
  "        Always be helpful, personal, and context-aware in your responses.\n",
  "        \"\"\"\n",
  "    ),\n",
  ")"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Responding to the User\n",
  "\n",
  "Now we can write our node that invokes the agent and responds to the user:"
 ]
},
{
 "cell_type": "code",
 "execution_count": 96,
 "metadata": {},
 "outputs": [],
 "source": [
  "def respond_to_user(state: RuntimeState, config: RunnableConfig) -> RuntimeState:\n",
  "    \"\"\"Invoke the travel agent to generate a response.\"\"\"\n",
  "    human_messages = [m for m in state[\"messages\"] if isinstance(m, HumanMessage)]\n",
  "    if not human_messages:\n",
  "        logger.warning(\"No HumanMessage found in state\")\n",
  "        return state\n",
  "\n",
  "    try:\n",
  "        for result in travel_agent.stream(\n",
  "            {\"messages\": state[\"messages\"]}, config=config, stream_mode=\"messages\"\n",
  "        ):\n",
  "            result_messages = result.get(\"messages\", [])\n",
  "\n",
  "            ai_messages = [\n",
  "                m\n",
  "                for m in result_messages\n",
  "                if isinstance(m, AIMessage) or isinstance(m, AIMessageChunk)\n",
  "            ]\n",
  "            if ai_messages:\n",
  "                agent_response = ai_messages[-1]\n",
  "                # Append only the agent's response to the original state\n",
  "                state[\"messages\"].append(agent_response)\n",
  "\n",
  "    except Exception as e:\n",
  "        logger.error(f\"Error invoking travel agent: {e}\")\n",
  "        agent_response = AIMessage(\n",
  "            content=\"I'm sorry, I encountered an error processing your request.\"\n",
  "        )\n",
  "    return state"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Summarizing Conversation History\n",
  "\n",
  "We've been focusing on long-term memory, but let's bounce back to short-term\n",
  "memory for a moment. With `RedisSaver`, LangGraph will manage our message\n",
  "history automatically. Still, the message history will continue to grow\n",
  "indefinitely, until it overwhelms the LLM's token context window.\n",
```

```
  "\n",
  "To solve this problem, we'll add a node to the graph that summarizes the\n",
  "conversation if it's grown past a threshold."
 ]
},
{
 "cell_type": "code",
 "execution_count": 97,
 "metadata": {},
 "outputs": [],
 "source": [
  "from langchain_core.messages import RemoveMessage\n",
  "\n",
  "# An LLM configured for summarization.\n",
  "summarizer = ChatOpenAI(model=\"gpt-4o\", temperature=0.3)\n",
  "\n",
  "# The number of messages after which we'll summarize the conversation.\n",
  "MESSAGE_SUMMARIZATION_THRESHOLD = 10\n",
  "\n",
  "\n",
  "def summarize_conversation(\n",
  "    state: RuntimeState, config: RunnableConfig\n",
  ") -> Optional[RuntimeState]:\n",
  "    \"\"\"\n",
  "    Summarize a list of messages into a concise summary to reduce context length\n",
  "    while preserving important information.\n",
  "    \"\"\"\n",
  "    messages = state[\"messages\"]\n",
  "    current_message_count = len(messages)\n",
  "    if current_message_count < MESSAGE_SUMMARIZATION_THRESHOLD:\n",
  "        logger.debug(f\"Not summarizing conversation: {current_message_count}\")\n",
  "        return state\n",
  "\n",
  "    system_prompt = \"\"\"\n",
  "    You are a conversation summarizer. Create a concise summary of the previous\n",
  "    conversation between a user and a travel assistant.\n",
  "    \n",
  "    The summary should:\n",
  "    1. Highlight key topics, preferences, and decisions\n",
  "    2. Include any specific trip details (destinations, dates, preferences)\n",
  "    3. Note any outstanding questions or topics that need follow-up\n",
  "    4. Be concise but informative\n",
  "    \n",
  "    Format your summary as a brief narrative paragraph.\n",
  "    \"\"\"\n",
  "\n",
  "    message_content = \"\\n\".join(\n",
  "        [\n",
  "            f\"{'User' if isinstance(msg, HumanMessage) else 'Assistant'}: {msg.content}\"\n",
  "            for msg in messages\n",
  "        ]\n",
  "    )\n",
  "\n",
  "    # Invoke the summarizer\n",
  "    summary_messages = [\n",
  "        SystemMessage(content=system_prompt),\n",
  "        HumanMessage(\n",
  "            content=f\"Please summarize this conversation:\\n\\n{message_content}\"\n",
  "        ),\n",
  "    ]\n",
  "\n",
  "    summary_response = summarizer.invoke(summary_messages)\n",
  "\n",
  "    logger.info(f\"Summarized {len(messages)} messages into a conversation summary\")\n",
  "\n",
  "    summary_message = SystemMessage(\n",
  "        content=f\"\"\"\n",
  "        Summary of the conversation so far:\n",
  "        \n",
```

```
  "        {summary_response.content}\n",
  "        \n",
  "        Please continue the conversation based on this summary and the recent messages.\n",
  "        \"\"\"\n",
  "    )\n",
  "    remove_messages = [\n",
  "        RemoveMessage(id=msg.id) for msg in messages if msg.id is not None\n",
  "    ]\n",
  "\n",
  "    state[\"messages\"] = [  # type: ignore\n",
  "        *remove_messages,\n",
  "        summary_message,\n",
  "        state[\"messages\"][-1],\n",
  "    ]\n",
  "\n",
  "    return state.copy()"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Assembling the Graph\n",
  "\n",
  "It's time to assemble our graph!"
 ]
},
{
 "cell_type": "code",
 "execution_count": 98,
 "metadata": {},
 "outputs": [],
 "source": [
  "from langgraph.graph import StateGraph, END, START\n",
  "\n",
  "\n",
  "workflow = StateGraph(RuntimeState)\n",
  "\n",
  "workflow.add_node(\"respond\", respond_to_user)\n",
  "workflow.add_node(\"summarize_conversation\", summarize_conversation)\n",
  "\n",
  "if memory_strategy == MemoryStrategy.MANUAL:\n",
  "    # In manual memory mode, we'll retrieve relevant memories before\n",
  "    # responding to the user, and then augment the user's message with the\n",
  "    # relevant memories.\n",
  "    workflow.add_node(\"retrieve_memories\", retrieve_relevant_memories)\n",
  "    workflow.add_edge(START, \"retrieve_memories\")\n",
  "    workflow.add_edge(\"retrieve_memories\", \"respond\")\n",
  "else:\n",
  "    # In tool-calling mode, we'll respond to the user and let the LLM\n",
  "    # decide when to retrieve and store memories, using tool calls.\n",
  "    workflow.add_edge(START, \"respond\")\n",
  "\n",
  "# Regardless of memory strategy, we'll summarize the conversation after\n",
  "# responding to the user, to keep the context window manageable.\n",
  "workflow.add_edge(\"respond\", \"summarize_conversation\")\n",
  "workflow.add_edge(\"summarize_conversation\", END)\n",
  "\n",
  "# Finally, compile the graph.\n",
  "graph = workflow.compile(checkpointer=redis_saver)"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## Consolidating Memories in a Background Thread\n",
  "\n",
  "We're almost ready to create the main loop that runs our graph. First, though,\n",
```

```
  "let's create a worker that consolidates similar memories on a regular schedule,\n",
  "using semantic search. We'll run the worker in a background thread later, in the\n",
  "main loop."
 ]
},
{
 "cell_type": "code",
 "execution_count": 99,
 "metadata": {},
 "outputs": [],
 "source": [
  "from redisvl.query import FilterQuery\n",
  "\n",
  "\n",
  "def consolidate_memories(user_id: str, batch_size: int = 10):\n",
  "    \"\"\"\n",
  "    Periodically merge similar long-term memories for a user.\n",
  "    \"\"\"\n",
  "    logger.info(f\"Starting memory consolidation for user {user_id}\")\n",
  "    \n",
  "    # For each memory type, consolidate separately\n",
  "\n",
  "    for memory_type in MemoryType:\n",
  "        all_memories = []\n",
  "\n",
  "        # Get all memories of this type for the user\n",
  "        of_type_for_user = (Tag(\"user_id\") == user_id) & (\n",
  "            Tag(\"memory_type\") == memory_type\n",
  "        )\n",
  "        filter_query = FilterQuery(filter_expression=of_type_for_user)\n",
  "        \n",
  "        for batch in long_term_memory_index.paginate(filter_query, page_size=batch_size):\n",
  "            all_memories.extend(batch)\n",
  "            \n",
  "        all_memories = long_term_memory_index.query(filter_query)\n",
  "        if not all_memories:\n",
  "            continue\n",
  "\n",
  "        # Group similar memories\n",
  "        processed_ids = set()\n",
  "        for memory in all_memories:\n",
  "            if memory[\"id\"] in processed_ids:\n",
  "                continue\n",
  "\n",
  "            memory_embedding = memory[\"embedding\"]\n",
  "            vector_query = VectorRangeQuery(\n",
  "                vector=memory_embedding,\n",
  "                num_results=10,\n",
  "                vector_field_name=\"embedding\",\n",
  "                filter_expression=of_type_for_user\n",
  "                & (Tag(\"memory_id\") != memory[\"memory_id\"]),\n",
  "                distance_threshold=0.1,\n",
  "                return_fields=[\n",
  "                    \"content\",\n",
  "                    \"metadata\",\n",
  "                ],\n",
  "            )\n",
  "            similar_memories = long_term_memory_index.query(vector_query)\n",
  "\n",
  "            # If we found similar memories, consolidate them\n",
  "            if similar_memories:\n",
  "                combined_content = memory[\"content\"]\n",
  "                combined_metadata = memory[\"metadata\"]\n",
  "\n",
  "                if combined_metadata:\n",
  "                    try:\n",
  "                        combined_metadata = json.loads(combined_metadata)\n",
  "                    except Exception as e:\n",
  "                        logger.error(f\"Error parsing metadata: {e}\")\n",
```

```
"                        combined_metadata = {}\n",
"\n",
"                for similar in similar_memories:\n",
"                    # Merge the content of similar memories\n",
"                    combined_content += f\" {similar['content']}\"\n",
"\n",
"                    if similar[\"metadata\"]:\n",
"                        try:\n",
"                            similar_metadata = json.loads(similar[\"metadata\"])\n",
"                        except Exception as e:\n",
"                            logger.error(f\"Error parsing metadata: {e}\")\n",
"                        similar_metadata = {}\n",
"\n",
"                    combined_metadata = {**combined_metadata, **similar_metadata}\n",
"\n",
"                # Create a consolidated memory\n",
"                new_metadata = {\n",
"                    \"consolidated\": True,\n",
"                    \"source_count\": len(similar_memories) + 1,\n",
"                    **combined_metadata,\n",
"                }\n",
"                consolidated_memory = {\n",
"                    \"content\": summarize_memories(combined_content, memory_type),\n",
"                    \"memory_type\": memory_type.value,\n",
"                    \"metadata\": json.dumps(new_metadata),\n",
"                    \"user_id\": user_id,\n",
"                }\n",
"\n",
"                # Delete the old memories\n",
"                delete_memory(memory[\"id\"])\n",
"                for similar in similar_memories:\n",
"                    delete_memory(similar[\"id\"])\n",
"\n",
"                # Store the new consolidated memory\n",
"                store_memory(\n",
"                    content=consolidated_memory[\"content\"],\n",
"                    memory_type=memory_type,\n",
"                    user_id=user_id,\n",
"                    metadata=consolidated_memory[\"metadata\"],\n",
"                )\n",
"\n",
"                logger.info(\n",
"                    f\"Consolidated {len(similar_memories) + 1} memories into one\"\n",
"                )\n",
"\n",
"\n",
"def delete_memory(memory_id: str):\n",
"    \"\"\"Delete a memory from Redis\"\"\"\n",
"    try:\n",
"        result = long_term_memory_index.drop_keys([memory_id])\n",
"    except Exception as e:\n",
"        logger.error(f\"Deleting memory {memory_id} failed: {e}\")\n",
"    if result == 0:\n",
"        logger.debug(f\"Deleting memory {memory_id} failed: memory not found\")\n",
"    else:\n",
"        logger.info(f\"Deleted memory {memory_id}\")\n",
"\n",
"\n",
"def summarize_memories(combined_content: str, memory_type: MemoryType) -> str:\n",
"    \"\"\"Use the LLM to create a concise summary of similar memories\"\"\"\n",
"    try:\n",
"        system_prompt = f\"\"\"\n",
"        You are a memory consolidation assistant. Your task is to create a single, \n",
"        concise memory from these similar memory fragments. The new memory should\n",
"        be a {memory_type.value} memory.\n",
"        \n",
"        Combine the information without repetition while preserving all important details.\n",
"        \"\"\"\n",
"\n",
```

```
  "        messages = [\n",
  "            SystemMessage(content=system_prompt),\n",
  "            HumanMessage(\n",
  "                content=f\"Consolidate these similar memories into one:\\n\\n{combined_content}\"\n",
  "            ),\n",
  "        ]\n",
  "\n",
  "        response = summarizer.invoke(messages)\n",
  "        return str(response.content)\n",
  "    except Exception as e:\n",
  "        logger.error(f\"Error summarizing memories: {e}\")\n",
  "        # Fall back to just using the combined content\n",
  "        return combined_content\n",
  "\n",
  "\n",
  "def memory_consolidation_worker(user_id: str):\n",
  "    \"\"\"\n",
  "    Worker that periodically consolidates memories for the active user.\n",
  "\n",
  "    NOTE: In production, this would probably use a background task framework, such\n",
  "          as rq or Celery, and run on a schedule.\n",
  "    \"\"\"\n",
  "    while True:\n",
  "        try:\n",
  "            consolidate_memories(user_id)\n",
  "            # Run every 10 minutes\n",
  "            time.sleep(10 * 60)\n",
  "        except Exception as e:\n",
  "            logger.exception(f\"Error in memory consolidation worker: {e}\")\n",
  "            # If there's an error, wait an hour and try again\n",
  "            time.sleep(60 * 60)"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
 "source": [
  "## The Main Loop\n",
  "\n",
  "Now we can put everything together and run the main loop.\n",
  "\n",
  "Running this cell should ask for your OpenAI and Tavily keys, then a username\n",
  "and thread ID. You'll enter a loop in which you can enter queries and see\n",
  "responses from the agent printed below the following cell."
 ]
},
{
 "cell_type": "code",
 "execution_count": null,
 "metadata": {},
 "outputs": [],
 "source": [
  "import threading\n",
  "\n",
  "\n",
  "def main(thread_id: str = \"book_flight\", user_id: str = \"demo_user\"):\n",
  "    \"\"\"Main interaction loop for the travel agent\"\"\"\n",
  "    print(\"Welcome to the Travel Assistant! (Type 'exit' to quit)\")\n",
  "\n",
  "    config = RunnableConfig(configurable={\"thread_id\": thread_id, \"user_id\": user_id})\n",
  "    state = RuntimeState(messages=[])\n",
  "\n",
  "    # If we're using the manual memory strategy, we need to create a queue for\n",
  "    # memory processing and start a worker thread. After every 'round' of a\n",
  "    # conversation, the main loop will add the current state and config to the\n",
  "    # queue for memory processing.\n",
  "    if memory_strategy == MemoryStrategy.MANUAL:\n",
  "        # Create a queue for memory processing\n",
  "        memory_queue = Queue()\n",
```

```
  "\n",
  "        # Start a worker thread that will process memory extraction tasks\n",
  "        memory_thread = threading.Thread(\n",
  "            target=memory_worker, args=(memory_queue, user_id), daemon=True\n",
  "        )\n",
  "        memory_thread.start()\n",
  "\n",
  "    # We always run consolidation in the background, regardless of memory strategy.\n",
  "    consolidation_thread = threading.Thread(\n",
  "        target=memory_consolidation_worker, args=(user_id,), daemon=True\n",
  "    )\n",
  "    consolidation_thread.start()\n",
  "\n",
  "    while True:\n",
  "        user_input = input(\"\\nYou (type 'quit' to quit): \")\n",
  "\n",
  "        if not user_input:\n",
  "            continue\n",
  "\n",
  "        if user_input.lower() in [\"exit\", \"quit\"]:\n",
  "            print(\"Thank you for using the Travel Assistant. Goodbye!\")\n",
  "            break\n",
  "\n",
  "        state[\"messages\"].append(HumanMessage(content=user_input))\n",
  "\n",
  "        try:\n",
  "            # Process user input through the graph\n",
  "            for result in graph.stream(state, config=config, stream_mode=\"values\"):\n",
  "                state = RuntimeState(**result)\n",
  "\n",
  "            logger.debug(f\"# of messages after run: {len(state['messages'])}\")\n",
  "\n",
  "            # Find the most recent AI message, so we can print the response\n",
  "            ai_messages = [m for m in state[\"messages\"] if isinstance(m, AIMessage)]\n",
  "            if ai_messages:\n",
  "                message = ai_messages[-1].content\n",
  "            else:\n",
  "                logger.error(\"No AI messages after run\")\n",
  "                message = \"I'm sorry, I couldn't process your request properly.\"\n",
  "                # Add the error message to the state\n",
  "                state[\"messages\"].append(AIMessage(content=message))\n",
  "\n",
  "            print(f\"\\nAssistant: {message}\")\n",
  "\n",
  "            # Add the current state to the memory processing queue\n",
  "            if memory_strategy == MemoryStrategy.MANUAL:\n",
  "                memory_queue.put((state.copy(), config))\n",
  "\n",
  "        except Exception as e:\n",
  "            logger.exception(f\"Error processing request: {e}\")\n",
  "            error_message = \"I'm sorry, I encountered an error processing your request.\"\n",
  "            print(f\"\\nAssistant: {error_message}\")\n",
  "            # Add the error message to the state\n",
  "            state[\"messages\"].append(AIMessage(content=error_message))\n",
  "\n",
  "\n",
  "try:\n",
  "    user_id = input(\"Enter a user ID: \") or \"demo_user\"\n",
  "    thread_id = input(\"Enter a thread ID: \") or \"demo_thread\"\n",
  "except Exception:\n",
  "    # If we're running in CI, we don't have a terminal to input from, so just exit\n",
  "    exit()\n",
  "else:\n",
  "    main(thread_id, user_id)\n"
 ]
},
{
 "cell_type": "markdown",
 "metadata": {},
```

```
   "source": [
    "## That's a Wrap!\n",
    "\n",
    "Want to make your own agent? Try the [LangGraph Quickstart](https://langchain-ai.github.io/langgraph/tutorials/i
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "env",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.11.11"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 2
}
```

Implementation plan

Your plan:

Progress

Your progress: