```
import json
import logging
import time
import uuid
from datetime import datetime
from typing import Any, Dict, List, Optional

import yaml
from pydantic import BaseModel
from swarm_models.tiktoken_wrapper import TikTokenizer

logger = logging.getLogger(__name__)


class MemoryMetadata(BaseModel):
    """Metadata for memory entries"""

    timestamp: Optional[float] = time.time()
    role: Optional[str] = None
    agent_name: Optional[str] = None
    session_id: Optional[str] = None
    memory_type: Optional[str] = None  # 'short_term' or 'long_term'
    token_count: Optional[int] = None
    message_id: Optional[str] = str(uuid.uuid4())
```

```
class MemoryEntry(BaseModel):
    """Single memory entry with content and metadata"""

    content: Optional[str] = None
    metadata: Optional[MemoryMetadata] = None


class MemoryConfig(BaseModel):
    """Configuration for memory manager"""

    max_short_term_tokens: Optional[int] = 4096
    max_entries: Optional[int] = None
    system_messages_token_buffer: Optional[int] = 1000
    enable_long_term_memory: Optional[bool] = False
    auto_archive: Optional[bool] = True
    archive_threshold: Optional[float] = 0.8  # Archive when 80% full


class MemoryManager:
    """
    Manages both short-term and long-term memory for an agent, handling token limits,
    archival, and context retrieval.

    Args:
        config (MemoryConfig): Configuration for memory management
        tokenizer (Optional[Any]): Tokenizer to use for token counting
        long_term_memory (Optional[Any]): Vector store or database for long-term storage
    """

    def __init__(
        self,
        config: MemoryConfig,
        tokenizer: Optional[Any] = None,
        long_term_memory: Optional[Any] = None,
    ):
        self.config = config
        self.tokenizer = tokenizer or TikTokenizer()
        self.long_term_memory = long_term_memory

        # Initialize memories
        self.short_term_memory: List[MemoryEntry] = []
        self.system_messages: List[MemoryEntry] = []

        # Memory statistics
        self.total_tokens_processed: int = 0
        self.archived_entries_count: int = 0

    def create_memory_entry(
        self,
        content: str,
        role: str,
        agent_name: str,
        session_id: str,
        memory_type: str = "short_term",
    ) -> MemoryEntry:
        """Create a new memory entry with metadata"""
        metadata = MemoryMetadata(
            timestamp=time.time(),
            role=role,
            agent_name=agent_name,
            session_id=session_id,
            memory_type=memory_type,
            token_count=self.tokenizer.count_tokens(content),
        )
        return MemoryEntry(content=content, metadata=metadata)


    def add_memory(
        self,
        content: str,
        role: str,
        agent_name: str,
        session_id: str,
        is_system: bool = False,
    ) -> None:
        """Add a new memory entry to appropriate storage"""
        entry = self.create_memory_entry(
            content=content,
            role=role,
            agent_name=agent_name,
            session_id=session_id,
            memory_type="system" if is_system else "short_term",
        )

        if is_system:
            self.system_messages.append(entry)
        else:
            self.short_term_memory.append(entry)

        # Check if archiving is needed
        if self.should_archive():
            self.archive_old_memories()

        self.total_tokens_processed += entry.metadata.token_count

    def get_current_token_count(self) -> int:
        """Get total tokens in short-term memory"""
        return sum(
            entry.metadata.token_count
            for entry in self.short_term_memory
        )

    def get_system_messages_token_count(self) -> int:
        """Get total tokens in system messages"""
        return sum(
            entry.metadata.token_count
            for entry in self.system_messages
        )

    def should_archive(self) -> bool:
        """Check if archiving is needed based on configuration"""
        if not self.config.auto_archive:
            return False

        current_usage = (
            self.get_current_token_count()
            / self.config.max_short_term_tokens
        )
        return current_usage >= self.config.archive_threshold

    def archive_old_memories(self) -> None:
        """Move older memories to long-term storage"""
        if not self.long_term_memory:
            logger.warning(
                "No long-term memory storage configured for archiving"
            )
            return

        while self.should_archive():
            # Get oldest non-system message
            if not self.short_term_memory:
```

```
            break

        oldest_entry = self.short_term_memory.pop(0)

        # Store in long-term memory
        self.store_in_long_term_memory(oldest_entry)
        self.archived_entries_count += 1

def store_in_long_term_memory(self, entry: MemoryEntry) -> None:
    """Store a memory entry in long-term memory"""
    if self.long_term_memory is None:
        logger.warning(
            "Attempted to store in non-existent long-term memory"
        )
        return

    try:
        self.long_term_memory.add(str(entry.model_dump()))
    except Exception as e:
        logger.error(f"Error storing in long-term memory: {e}")
        # Re-add to short-term if storage fails
        self.short_term_memory.insert(0, entry)

def get_relevant_context(
    self, query: str, max_tokens: Optional[int] = None
) -> str:
```

```
"""
Get relevant context from both memory types

Args:
    query (str): Query to match against memories
    max_tokens (Optional[int]): Maximum tokens to return

Returns:
    str: Combined relevant context
"""
contexts = []

# Add system messages first
for entry in self.system_messages:
    contexts.append(entry.content)

# Add short-term memory
for entry in reversed(self.short_term_memory):
    contexts.append(entry.content)

# Query long-term memory if available
if self.long_term_memory is not None:
    long_term_context = self.long_term_memory.query(query)
    if long_term_context:
        contexts.append(str(long_term_context))
    # Combine and truncate if needed
    combined = "\n".join(contexts)
    if max_tokens:
        combined = self.truncate_to_token_limit(
            combined, max_tokens
        )

    return combined

def truncate_to_token_limit(
    self, text: str, max_tokens: int
) -> str:
    """Truncate text to fit within token limit"""
    current_tokens = self.tokenizer.count_tokens(text)

    if current_tokens <= max_tokens:
        return text

    # Truncate by splitting into sentences and rebuilding
    sentences = text.split(". ")
    result = []
    current_count = 0

    for sentence in sentences:
        sentence_tokens = self.tokenizer.count_tokens(sentence)
        if current_count + sentence_tokens <= max_tokens:
            result.append(sentence)
            current_count += sentence_tokens,
        else:
            break

    return ". ".join(result)

def clear_short_term_memory(
    self, preserve_system: bool = True
) -> None:
    """Clear short-term memory with option to preserve system messages"""
    if not preserve_system:
        self.system_messages.clear()
    self.short_term_memory.clear()
    logger.info(
        "Cleared short-term memory"
        + " (preserved system messages)"
        if preserve_system
        else ""
    )

def get_memory_stats(self) -> Dict[str, Any]:
    """Get detailed memory statistics"""
    return {
        "short_term_messages": len(self.short_term_memory),
        "system_messages": len(self.system_messages),
        "current_tokens": self.get_current_token_count(),
        "system_tokens": self.get_system_messages_token_count(),
        "max_tokens": self.config.max_short_term_tokens,
        "token_usage_percent": round(
            (
                self.get_current_token_count()
                / self.config.max_short_term_tokens
            )
            * 100,
            2,
        ),
        "has_long_term_memory": self.long_term_memory is not None,
        "archived_entries": self.archived_entries_count,
        "total_tokens_processed": self.total_tokens_processed,
    }

def save_memory_snapshot(self, file_path: str) -> None:
    """Save current memory state to file"""
    try:
        data = {
            "timestamp": datetime.now().isoformat(),
            "config": self.config.model_dump(),
            "system_messages": [
                entry.model_dump()
                for entry in self.system_messages
            ],
            "short_term_memory": [
                entry.model_dump()
                for entry in self.short_term_memory
            ],
            "stats": self.get_memory_stats(),
        }

        with open(file_path, "w") as f:
            if file_path.endswith(".yaml"):
                yaml.dump(data, f)
            else:
                json.dump(data, f, indent=2)

        logger.info(f"Saved memory snapshot to {file_path}")

    except Exception as e:
        logger.error(f"Error saving memory snapshot: {e}")
        raise

def load_memory_snapshot(self, file_path: str) -> None:
    """Load memory state from file"""
    try:
        with open(file_path, "r") as f:
            if file_path.endswith(".yaml"):
                data = yaml.safe_load(f)
            else:
```

```
                data = json.load(f)

        self.config = MemoryConfig(**data["config"])
        self.system_messages = [
            MemoryEntry(**entry)
            for entry in data["system_messages"]
        ]
        self.short_term_memory = [
            MemoryEntry(**entry)
            for entry in data["short_term_memory"]
        ]

        logger.info(f"Loaded memory snapshot from {file_path}")

    except Exception as e:
        logger.error(f"Error loading memory snapshot: {e}")
        raise

def search_memories(
    self, query: str, memory_type: str = "all"
) -> List[MemoryEntry]:
    """
    Search through memories of specified type

    Args:
        query (str): Search query
        memory_type (str): Type of memories to search ("short_term", "system", "long_term", or "all")

    Returns:
        List[MemoryEntry]: Matching memory entries
    """
    results = []

    if memory_type in ["short_term", "all"]:
        results.extend(
            [
                entry
                for entry in self.short_term_memory
                if query.lower() in entry.content.lower()
            ]
        )

    if memory_type in ["system", "all"]:
        results.extend(
            [
                entry
                for entry in self.system_messages
                if query.lower() in entry.content.lower()
            ]
        )

    if (
        memory_type in ["long_term", "all"]
        and self.long_term_memory is not None
    ):
        long_term_results = self.long_term_memory.query(query)
        if long_term_results:
            # Convert long-term results to MemoryEntry format
            for result in long_term_results:
                content = str(result)
                metadata = MemoryMetadata(
                    timestamp=time.time(),
                    role="long_term",
                    agent_name="system",
                    session_id="long_term",
                    memory_type="long_term",
                    token_count=self.tokenizer.count_tokens(
                        content
                    ),
                )
                results.append(
                    MemoryEntry(
                        content=content, metadata=metadata
                    )
                )

    return results
```

```
def get_memory_by_timeframe(
    self, start_time: float, end_time: float
) -> List[MemoryEntry]:
    """Get memories within a specific timeframe"""
    return [
        entry
        for entry in self.short_term_memory
        if start_time <= entry.metadata.timestamp <= end_time
    ]


def export_memories(
    self, file_path: str, format: str = "json"
) -> None:
    """Export memories to file in specified format"""
    data = {
        "system_messages": [
            entry.model_dump() for entry in self.system_messages
        ],
        "short_term_memory": [
            entry.model_dump() for entry in self.short_term_memory
        ],
        "stats": self.get_memory_stats(),
    }

    with open(file_path, "w") as f:
        if format == "yaml":
```

```
    yaml.dump(data, f)

else:

    json.dump(data, f, indent=2)
```