Azentiq Memory Manager: User Guide

Table of Contents

1. Introduction

Azentiq Memory Manager is a flexible framework designed for sophisticated contextual memory management in AI applications. This system organizes information into tiered storage with configurable persistence characteristics, enabling AI applications to maintain context effectively across interactions. The library addresses the critical challenge of balancing context retention with resource constraints, particularly token limitations in large language model applications.

```
# Basic initialization example
from core.memory_manager import MemoryManager
memory_manager = MemoryManager(redis_url="redis://localhost:6379/0")
```

2. Installation

The Azentiq Memory Manager can be installed through standard Python package management tools. Choose the installation method that best aligns with your development workflow and requirements.

```
# Install from PyPI
pip install azentiq-memory-manager

# Or clone repository and install
git clone https://github.com/azentiq/memory-manager.git
cd memory-manager
pip install -e .
```

3. Core Features

3.1 Memory Tiers

Memory tiers represent distinct persistence levels for stored information, each optimized for different retention requirements. This tiered architecture ensures efficient resource utilization while maintaining appropriate information availability.

```
from core.memory_manager import MemoryManager, MemoryTier

# Initialize memory manager
memory_manager = MemoryManager(redis_url="redis://localhost:6379/0")

# Available tiers:
# - SHORT_TERM: Temporary storage with TTL (time-to-live)
# - WORKING: Session-persistent storage without expiration
# - LONG_TERM: Permanent storage (planned for future releases)
```

Implementation Recommendation: Select tiers based on information lifecycle requirements—temporary context in SHORT_TERM, session-important information in WORKING, and permanent knowledge (future capability) in LONG_TERM.

3.2 Basic Memory Operations

The core API provides intuitive operations for memory lifecycle management. These operations enable applications to create, retrieve, update, and delete memories with precise control over persistence and importance attributes.

```
# Add a memory
memory_id = memory_manager.add_memory(
    content="This is important information",
    metadata={"type": "fact", "source": "user"},
    tier=MemoryTier.SHORT_TERM,  # Default if not specified
    importance=0.7,  # 0.0 to 1.0, default is 0.0
    session_id="user123"  # Optional grouping
)

# Retrieve a memory
memory = memory_manager.get_memory(memory_id)

# Update a memory
memory_manager.update_memory(
    memory_id=memory_id,
    content="Updated information",
    importance=0.8,
    tier=MemoryTier.WORKING  # Move to a different tier
)

# Delete a memory
memory_manager.delete_memory(memory_id)

# Search for memories
memories = memory_manager.search_by_metadata(
    {"type": "fact"},
    tier=MemoryTier.SHORT_TERM
)
```

Implementation Recommendation: Structure memory operations around well-defined metadata schemas to facilitate efficient searching and organization throughout the application lifecycle.

3.3 Token Budget Management

The token budget management system addresses the critical challenge of operating within context window constraints of language models. These tools provide precise token estimation and strategic memory selection to optimize information retrieval within token limitations.

```
from utils.token_budget.manager import TokenBudgetManager
from utils.token_budget.estimator import TokenEstimator

# Initialize token management
estimator = TokenEstimator()
budget_manager = TokenBudgetManager(
    token_estimator=estimator,
    max_tokens=4000  # Maximum tokens to use
)

# Select memories within token budget
selected_memories = budget_manager.select_memories(
    memories=all_memories,
    recency_weight=0.6,  # Higher values prioritize recent memories
    importance_weight=0.4  # Higher values prioritize important memories
)

# Add a memory while adapting to token budget
budget_manager.add_memory(memory, memories_dict)
```

Implementation Recommendation: Configure token budget parameters based on your specific LLM's context window, with careful attention to the balance between recency and importance for your application's particular context requirements.

4. Memory Management Approaches

4.1 Direct Tier Assignment (Consumer-Level)

The direct assignment approach provides immediate, explicit control over memory tier placement directly within application code. This approach excels in applications with straightforward memory organization requirements where tier assignments follow clear, predictable patterns.

```
# Explicitly assign memories to tiers based on your application logic
memory_manager.add_memory(
    content="Temperature anomaly detected",
    metadata={"type": "anomaly", "device_id": "sensor001"},
    importance=0.9,  # High importance for anomalies
    tier=MemoryTier.WORKING,  # Store in working memory for longer retention
    session_id="iot_session"
)
```

Implementation Recommendation: Consider direct tier assignment for applications with fixed, well-understood memory retention requirements where the classification logic is domain-specific and unlikely to change frequently.

4.2 Template-Based Progression (Declarative)

The template-based approach introduces declarative memory management through YAML configuration files, separating memory organization policies from application code. This approach enables sophisticated memory lifecycle management with minimal code changes when memory policies evolve.

```
from progression.engine import ProgressionEngine

# Initialize progression engine with a template
engine = ProgressionEngine(
    memory_manager=memory_manager,
    template_name="conversational"  # Built-in template
)

# Or use a custom template
engine = ProgressionEngine(
    memory_manager=memory_manager,
    template_path="/path/to/custom_template.yaml"
)

# The engine will automatically handle memory progression based on rules
```

Example template (YAML):

```
name: "custom_template"
description: "Custom progression rules"
tiers:
  - name: "SHORT_TERM"
    ttl: 3600  # 1 hour in seconds
    types:
      - name: "conversation_turn"
        retention_policy: "time_based"
        max_items: 20
  - name: "WORKING"
    ttl: 86400  # 24 hours in seconds
    types:
      - name: "important_fact"
        retention_policy: "importance_based"
        min_importance: 0.7
rules:
  - name: "promote_high_importance"
    trigger:
      event: "memory_added"
    condition: "memory.importance > 0.8"
    action:
      type: "promote"
      source_tier: "SHORT_TERM"
      target_tier: "WORKING"
```

Implementation Recommendation: Deploy template-based progression for applications with complex memory lifecycle requirements, especially when memory management policies need to evolve independently from application code.

5. Advanced Features

5.1 Memory Adaptation Strategies

Adaptation strategies provide sophisticated mechanisms for managing memory collections when they exceed token budgets or other constraints. These strategies intelligently select which memories to retain, modify, or remove based on configurable criteria.

```
from utils.token_budget.adaptation.reduce import ReduceAdaptationStrategy

# Initialize adaptation strategy
reduce_strategy = ReduceAdaptationStrategy(token_estimator=estimator)

# Adapt memories to fit within token budget
adapted_memories, new_token_count, removed_ids = reduce_strategy.adapt_memories(
    memories=memories_dict,
    used_tokens=current_tokens,
    target_tokens=target_tokens,
    reduction_target=0.2  # Aim to reduce by 20%
)
```

Implementation Recommendation: Implement adaptation strategies for applications with long-running sessions or large memory collections, particularly when working with restrictive token budgets.

5.2 Custom Memory Selectors

Memory selectors enable fine-grained control over which memories are retrieved for specific contexts. By balancing factors such as recency, importance, and relevance, selectors ensure optimal memory utilization.

```
from utils.token_budget.selection.priority import PriorityMemorySelector

# Initialize selector
selector = PriorityMemorySelector()

# Select memories with custom weights
selected = selector.select_memories(
    memories=all_memories,
    max_tokens=3000,
    recency_weight=0.7,
    importance_weight=0.3
)
```

Implementation Recommendation: Fine-tune selection parameters based on your application's specific context requirements—prioritize recency for rapidly changing environments and importance for decision-critical information.

6. Sample Applications

6.1 Conversational AI

This implementation pattern demonstrates memory management for conversational systems where maintaining context across multiple turns is essential. The pattern illustrates proper conversation turn storage and context retrieval techniques.

```
# Initialize with conversational template
engine = ProgressionEngine(
    memory_manager=memory_manager,
    template_name="conversational"
)

# Store conversation turn
memory_manager.add_memory(
    content=user_message,
    metadata={"type": "conversation_turn", "speaker": "user"},
    tier=MemoryTier.SHORT_TERM,
    session_id=conversation_id
)

# Later, retrieve relevant context
context_memories = memory_manager.search_by_metadata(
    {"type": "conversation_turn"},
    tier=MemoryTier.SHORT_TERM,
    session_id=conversation_id
)
```

Implementation Recommendation: Structure conversational memory with clear turn demarcation, and consider implementing summarization strategies for longer conversations to maintain context within token constraints.

6.2 IoT Monitoring (as seen in samples)

This implementation demonstrates memory management for IoT applications where high-volume telemetry data must be balanced with retention of significant anomalies and insights. The pattern illustrates effective tier separation based on data significance.

```
# Store telemetry in SHORT_TERM
memory_manager.add_memory(
    content=json.dumps(telemetry_data),
    metadata={
        "type": "telemetry",
        "device_id": device_id,
        "timestamp": datetime.now().isoformat()
    },
    tier=MemoryTier.SHORT_TERM,
    session_id=session_id
)

# Store anomalies in WORKING memory
memory_manager.add_memory(
    content=anomaly_description,
    metadata={
        "type": "anomaly",
        "device_id": device_id
    },
    importance=0.9,  # High importance
    tier=MemoryTier.WORKING,
    session_id=session_id
)
```

Implementation Recommendation: Implement clear demarcation between routine telemetry and significant events, using tier assignment to establish appropriate retention policies for each data category.

7. Best Practices

Memory Tier Selection

Effective tier selection optimizes resource utilization while ensuring appropriate information persistence. Consider the following guidelines for optimal tier configuration:

- Allocate transient, session-specific information to SHORT_TERM tier
- Reserve WORKING tier for important findings requiring extended persistence
- Plan for LONG_TERM tier usage (future capability) for permanent knowledge storage

Importance Assignment

Strategic importance assignment ensures priority for critical information during selection and adaptation processes:

- Critical information warranting persistent retention: 0.7-1.0
- Contextually useful but non-critical information: 0.3-0.6
- Routine or background information: 0.0-0.2

Token Budget Management

Effective token budget configuration maximizes context utilization within model constraints:

- Align token budgets with your specific LLM's context window capacity
- Configure recency and importance weights based on your application's specific context requirements
- Implement appropriate adaptation strategies for long-running sessions

Architecture Selection

Choose the appropriate memory management architecture based on application complexity and flexibility requirements:

- Template-based approach for complex memory lifecycle requirements
- Direct assignment for straightforward applications with clear memory policies
- Consider hybrid approaches that leverage both paradigms for optimal flexibility

Metadata Structuring

Consistent metadata schemas enable efficient memory organization and retrieval:

- Implement standard type field conventions for memory categorization
- Include temporal metadata for chronological organization
- Incorporate domain-specific identifiers for contextual grouping

Metadata Example and Usage Patterns

Metadata provides powerful capabilities for organizing, retrieving, and relating memories. Below is an example of a comprehensive metadata schema for a research assistant agent:

```
# Adding a memory with structured metadata
memory_manager.add_memory(
    content="The study by Smith et al. (2024) found that quantum computing advances s
    metadata={
        # Core categorization
        "type": "research_finding",          # Primary content type
        "subtype": "empirical_result",       # Content subtype

        # Domain-specific attributes
        "domain": "quantum_computing",       # Subject domain
        "source": "academic_paper",          # Source type
        "citation": "Smith et al. (2024)",   # Reference information
        "confidence": 0.95,                  # Data reliability score

        # Temporal context
        "discovered_at": "2025-06-15T10:30:00Z",  # When this information was found
        "publication_date": "2024-12-01",    # Original publication date

        # Relational connections
        "related_topics": ["error_correction", "quantum_gates", "qubits"],
        "supports": ["quantum_advantage_hypothesis"],  # Links to theories this suppo
        "contradicts": ["classical_simulation_theory"], # Links to theories this chal

        # Usage tracking
        "times_retrieved": 0,               # Track memory usage frequency
        "last_used_for": None,              # Track last usage context
    },
    tier=MemoryTier.WORKING,
    importance=0.8,  # High importance for novel research findings
    session_id="quantum_research_project"
)
```

Leveraging Metadata in Memory Retrieval:

```
# Find all quantum computing research with high confidence
quantum_research = memory_manager.search_by_metadata(
    {"domain": "quantum_computing", "confidence": {"$gt": 0.8}},
    tier=MemoryTier.WORKING
)

# Find contradictory evidence to a theory
contradictions = memory_manager.search_by_metadata(
    {"contradicts": "classical_simulation_theory"},
    tier=MemoryTier.WORKING
)

# Track memory usage
for memory in retrieved_memories:
    memory_manager.update_memory(
        memory_id=memory.memory_id,
        metadata={
            **memory.metadata,  # Preserve existing metadata
            "times_retrieved": memory.metadata.get("times_retrieved", 0) + 1,
            "last_used_for": "quantum_algorithm_development"
        }
    )
```

Implementation Recommendation: Design metadata schemas that anticipate future query patterns. Create consistent naming conventions for key fields across your application, and build hierarchical relationships through field conventions like type/subtype pairs. Update metadata dynamically to track usage patterns and enable adaptive memory management.

8. Agentic AI Integration

The Azentiq Memory Manager provides a specialized framework for autonomous agent development, enabling a critical separation of concerns that maximizes development efficiency and agent capabilities.

8.1 Separation of Concerns

The framework allows agent developers to focus on core reasoning and domain-specific logic, while delegating memory management complexity to a dedicated subsystem:

```
# Initialize memory infrastructure (once)
memory_manager = MemoryManager(redis_url="redis://localhost:6379/0")
engine = ProgressionEngine(memory_manager, template_name="agent")

# In agent implementation, focus purely on agent logic
class AutonomousAgent:
    def __init__(self, memory_manager):
        self.memory_manager = memory_manager

    def plan(self, objective):
        # Get relevant context from memory
        context = self.memory_manager.search_by_metadata(
            {"relevant_to": objective},
            tier=MemoryTier.WORKING
        )

        # Focus on core planning logic, not memory mechanics
        plan = self._generate_plan(objective, context)
        return plan
```

Implementation Recommendation: Design agents with a clear boundary between memory interaction and core reasoning/decision-making components to maximize maintainability.

8.2 Context Window Optimization

The framework's token budget management system is particularly valuable for agentic AI systems operating with large language models, providing automated handling of context window constraints:

```
from utils.token_budget.manager import TokenBudgetManager

# Initialize token management for agent's LLM context
budget_manager = TokenBudgetManager(token_estimator, max_tokens=8000)

# Agent can retrieve optimized context without handling token management
def get_context_for_task(self, task_description):
    # Query potentially relevant memories
    candidate_memories = self.memory_manager.search_by_similarity(
        task_description,
        limit=100  # Get many candidates
    )

    # Let token budget manager optimize selection
    selected_memories = budget_manager.select_memories(
        candidate_memories,
        recency_weight=0.4,
        importance_weight=0.6
    )

    return selected_memories
```

Implementation Recommendation: Configure token budgets to align with your agent's specific LLM, and adjust priority weights based on your agent's task characteristics.

8.3 Progressive Memory Architecture

The template-based progression system enables sophisticated agent memory architectures inspired by cognitive science models:

```
# agent_memory.yaml
name: "autonomous_agent"
description: "Cognitive architecture for autonomous agents"
tiers:
  - name: "SHORT_TERM"
    ttl: 1800  # 30 minutes
    types:
      - name: "perception"
        retention_policy: "time_based"
        max_items: 50
      - name: "working_memory"
        retention_policy: "count_based"
        max_items: 10
  - name: "WORKING"
    ttl: 86400  # 24 hours
    types:
      - name: "episodic_memory"
        retention_policy: "importance_based"
        min_importance: 0.6
      - name: "learned_fact"
        retention_policy: "importance_based"
        min_importance: 0.7
rules:
  - name: "consolidate_important_perceptions"
    trigger:
      event: "memory_added"
      metadata:
        type: "perception"
    condition: "memory.importance > 0.8"
    action:
      type: "promote"
      source_tier: "SHORT_TERM"
      target_tier: "WORKING"
      target_type: "episodic_memory"
```

Implementation Recommendation: Model your agent's memory architecture after cognitive patterns appropriate to your domain, taking advantage of declarative progression rules.

8.4 Integration with Agent Frameworks

The Azentiq Memory Manager integrates seamlessly with popular agent frameworks:

```
# LangChain integration example
from langchain.agents import AgentExecutor, create_react_agent
from langchain.memory import ConversationBufferMemory

class AzentiqMemoryAdapter(ConversationBufferMemory):
    def __init__(self, memory_manager, session_id):
        self.memory_manager = memory_manager
        self.session_id = session_id

    def save_context(self, inputs, outputs):
        # Store in Azentiq memory system
        self.memory_manager.add_memory(
            content=f"User: {inputs}\nAI: {outputs}",
            metadata={"type": "conversation_turn"},
            tier=MemoryTier.SHORT_TERM,
            session_id=self.session_id
        )

    def load_memory_variables(self, inputs):
        # Retrieve from Azentiq memory system
        memories = self.memory_manager.search_by_metadata(
            {"type": "conversation_turn"},
            tier=MemoryTier.SHORT_TERM,
            session_id=self.session_id
        )
        return {"history": "\n".join([m.content for m in memories])}

# Create agent with Azentiq memory
agent_memory = AzentiqMemoryAdapter(memory_manager, "session123")
agent = create_react_agent(llm, tools, prompt, agent_memory)
agent_executor = AgentExecutor(agent=agent, tools=tools, memory=agent_memory)
```

Implementation Recommendation: Create adapter classes that implement framework-specific memory interfaces while leveraging the full capabilities of the Azentiq Memory Manager underneath.

8.5 Memory-Aware Agent Design Patterns

Several design patterns particularly benefit from the memory framework:

1. **Reflective Agents**: Agents that periodically review and consolidate their experiences
2. **Multi-Task Agents**: Agents that must maintain separate context for different ongoing tasks
3. **Learning Agents**: Agents that progressively build knowledge bases from experiences
4. **Collaborative Agents**: Multi-agent systems sharing memory across specialized roles

Implementation Recommendation: Design your agent memory architecture to match your agent's cognitive requirements, leveraging the framework's flexibility to implement sophisticated memory patterns.