MemoryFormer: Generative Distributed, Declarative and Audited LLM Memory System

Sasu Tarkoma

Working Draft

July 2024

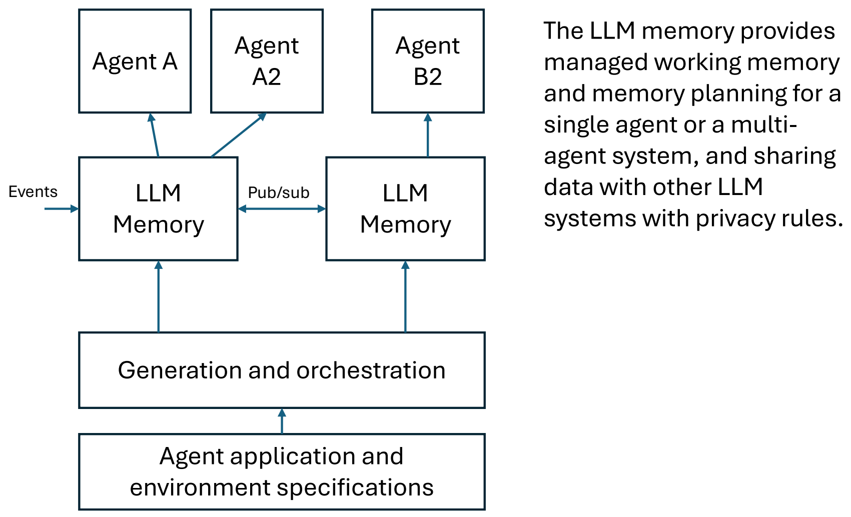
1 Introduction

We address a critical gap in existing solutions by introducing a distributed declarative LLM memory system, the DDMem structure and the MemoryFormer arhictecture for generating, maintaining and auditing agent memories. We propose a hierarchical and dynamic memory solution tailored for Large Language Models (LLMs) within applications requiring flexibility, real-time responsiveness, and decoupling of data sources.

The fundamental concept behind MemoryFormer revolves around transforming application memory into modular and flexible entities, based on the DDMem structure, that leverage rule-based templates combined with asynchronous data subscription mechanisms defined by programming languages or one or more LLMs. The DDMem structure is declarative and distributed supporting different agent configurations.

Figure 1 presents an overview of the MemoryFormer architecture. MemoryFormer generates LLM memories (DDMems) based on the input application specification and description and the system specification. The generation involves typically the use of a coordinating LLM for memory task generation, planning and optimization. The system uses a planner component for partitioning the memory objects over the edge-cloud continuum. The DDMems are then deployed with the chosen environment, such as Python or Javascript, and they are managed by the MemoryFormer orchestration component. The orchestration manages the synchronization and auditing of the memories. Given the managed nature of the architecture, the memory use can include LLM assessment and verification logic. The DMems are reactive and react to various events according to the application logic. The agents can then access the memories and invoke instructions on the memory contents.

Key features include:


Agent A
Agent A2
Agent B2
Events
LLM Memory
Pub/sub
LLM Memory
Generation and orchestration
Agent application and environment specifications

The LLM memory provides managed working memory and memory planning for a single agent or a multi-agent system, and sharing data with other LLM systems with privacy rules.

Figure 1: MemoryFormer system overview

1. Dynamic Configuration: DDMem allows the generation and customization of working memory specifically designed to meet the needs of LLM-based systems that are inherently dynamic.
2. Decoupled Data Sources: The solution facilitates a clear separation between various data inputs, enabling more efficient resource allocation within an application architecture without compromising performance or responsiveness.
3. Memory Planning: MemoryFormer enables planning for memory usage tailored to either single-prompt pipelines or multiple concurrent agents/pipelines.
4. We contribute two optimizations techniques: declarative memory management with privacy and compression rules, call graph optimization for partitioning memory fragments, and a memory fragment syncing technique based on pub/sub. The distributed optimization allows the minimization of expensive LLM invocations by disseminating results within the same application or trust domain.
5. We contribute a system for generating and distributing optimized auditing logic for the memories. The multi-agent auditing solution is based on identifying common memory parts.
6. We contribute a technique for transforming LLM operations into code during the generation of the memory instance and allowing to use both LLM ops and code at run-time. The microbenchmarks indicate significant performance benefit of running code instead of LLM ops.

2 Vision: Toward AI Metamemory

Through MemoryFormer, we envision a generative system for intent-based applications. The application is generated from an intent or related specification and the memory is designed to implement the application with both LLM operations and code. The generation includes determining auditing points for the memory and implementing auditing logic for both LLM operations and code. The memory is then deployed in the distributed environment and orchestrated. We envision a metamemory consisting of all the memory objects in the same trust domain that are optimized together. Figure 2 illustrates this framework.

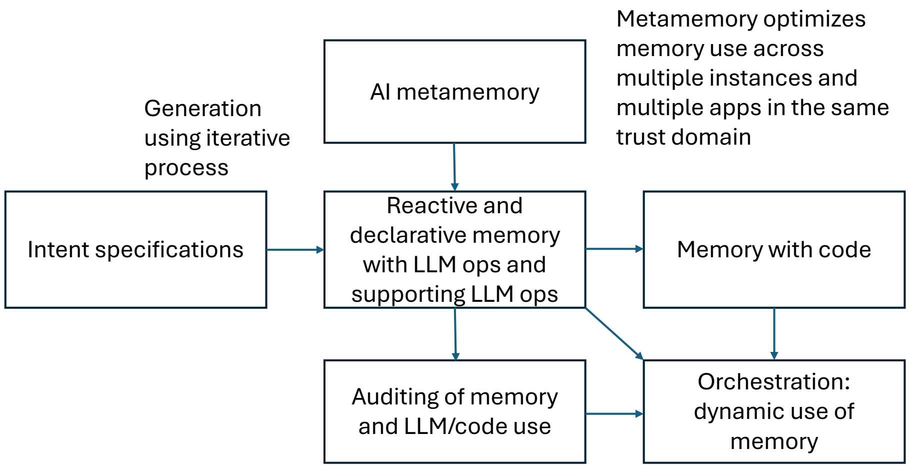

Metamemory optimizes memory use across multiple instances and multiple apps in the same trust domain
AI metamemory
Generation using iterative process
Intent specifications
Reactive and declarative memory with LLM ops and supporting LLM ops
Memory with code
Auditing of memory and LLM/code use
Orchestration: dynamic use of memory

Figure 2: Overall system diagram

3 Related Work

The current GenAI solutions and best practices involve the construction of prompts with context memory that are then transformed into embeddings for input to LLMs. The modern tools include AI pipelines, such as LangChain and Retrieval Augmented Generation (RAG) systems [4], multi-agent systems, and prompt compilers, such as DSPy [5]. Pipeline-based systems, namely LangChain, enabled client-specific prompt and context construction using RAG techniques. Multi-agent systems enabled prompt-based communication over multiple agents (LLMs) [1, 9].

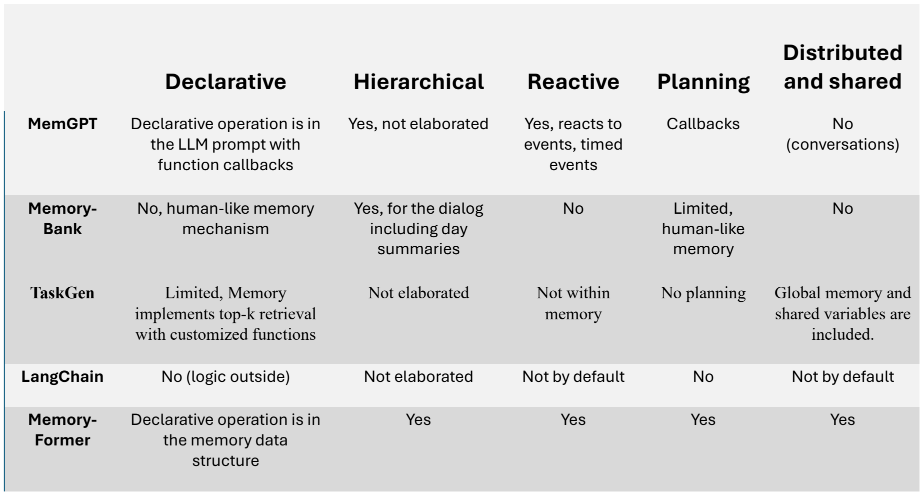
Recent results have identified the need to develop controllable working memories for LLMs [6] and support for long-term memories [10], and human-like episodic memory for LLMs [3].

As a key state of the art research example, MemGPT introduces an OS-inspired LLM system designed for virtual context management [7]. This system operates by triggering LLM inference through various events, which are generalized inputs to MemGPT. These events include user messages (in chat applications), system messages (such as main context capacity warnings), user interactions (like alerts for user logins or document uploads), and timed events that run on a regular schedule, enabling MemGPT to operate 'unprompted' without user intervention. The system features a LLM element enhanced with a memory system and functions that enable it to manage its own memory efficiently.

Code generation has been recently proposed as a technique for supporting LLM-enabled applications in the edge-cloud continuum [2]. Code generation is a suitable direction in generating memories for embedded devices. Instead of executing LLM inference, LLM inference generates code for run-time use.

Memory abstraction with a global context and shared variables has been proposed for multi-agent systems [8]. In this model, agents have memory banks that store processing results. The memory bank is a Python dictionary of memory items that are retrieved using top-k retrieval.

Figure 3 presents a summary of the key state of the art. MemoryFormer differs from the state of the art in its pub/sub reactive operation, planning of single and multi-agent memory use, and generating auditing logic. The MemoryFormer design can leverage existing pipeline optimizers and compilers, such as DSPy, in its operation.



	Declarative	**Hierarchical**	**Reactive**	**Planning**	**Distributed and shared**
MemGPT	Declarative operation is in the LLM prompt with function callbacks	Yes, not elaborated	Yes, reacts to events, timed events	Callbacks	No (conversations)
Memory-Bank	No, human-like memory mechanism	Yes, for the dialog including day summaries	No	Limited, human-like memory	No
TaskGen	Limited, Memory implements top-k retrieval with customized functions	Not elaborated	Not within memory	No planning	Global memory and shared variables are included.
LangChain	No (logic outside)	Not elaborated	Not by default	No	Not by default
Memory-Former	Declarative operation is in the memory data structure	Yes	Yes	Yes	Yes

Figure 3: Comparing agent memory solutions

4 Declarative Distributed LLM Memory Model (DDMem)

MemoryFormer is a distributed and declarative memory system designed for agent based applications. It employs a novel approach to memory management by incorporating the Declarative Distributed LLM Memory (DDMem) model, which consists of instructions $\mathcal{I}$ and content $\mathcal{C}$. This system description outlines the main components, algorithms, and features of MemoryFormer.

The DDMem structure includes two primary components: instructions $\mathcal{I}$ and content $\mathcal{C}$, defined as follows:

$$\mathcal{I} = \{\mathbf{c}_1, \mathbf{c}_2, ..., \mathbf{c}_n\} \tag{1}$$

where each command $\mathbf{c}_i$ has attributes: command type (e.g., SUBSCRIBE, LLM_QUERY), anchor point identifier for content, and optional parameters. The instructions are categorized into configuration parameters, initial content population, and reactive behaviors for updating memory contents based on asynchronous or timed events.

The instruction part provides planning capability for the use of memory. The instructions can also utilize LLM function callbacks for expressive operation. The use of function callbacks is managed by the MemoryFormer system.

The content component $\mathcal{C}$ is a graph data structure representing the knowledge base of the LLM agent:

$$\mathcal{C} = (\mathbf{V}, \mathbf{E}) \tag{2}$$

where nodes $v_i \in \mathbf{V}$ have attributes (unique identifier, type, and optional properties) and edges $e_i \in \mathbf{E}$ have attributes (unique identifier, relationship type, source node, and target node). Content is referred to with anchors $\mathcal{A}$, which are labels for identified parts of the content. The DDMem design can support graph based memory content; however, we have adopted to use hierarchical structure for the design to keep the system simple and efficient.

5 MemoryFormer Overview

Figure 4 outlines the key distributed scenarios for MemoryFormer. In a standalone case, an agent uses the DDMem structure locally and invokes LLMs based on the generated configuration, for example local or edge LLMs. In the client-server/cloud case, the DDMem is partitioned or replicated across the environment. In this case, some of the memory instructions are executed outside the client and synchronized using pub/sub on the client device. In the federated case, this partitioning and synchronization happens across the edge-cloud continuum. The MemoryFormer orchestration system is responsible for managing the distributed process. This design allows the optimization of the distributed memory and LLM use. As an example, instead of executing multiple LLM operations on a memory part shared by multiple memory instances, the operation can be done only once for the shared parts and communicated using pub/sub. Similar method is applied for the auditing logic.

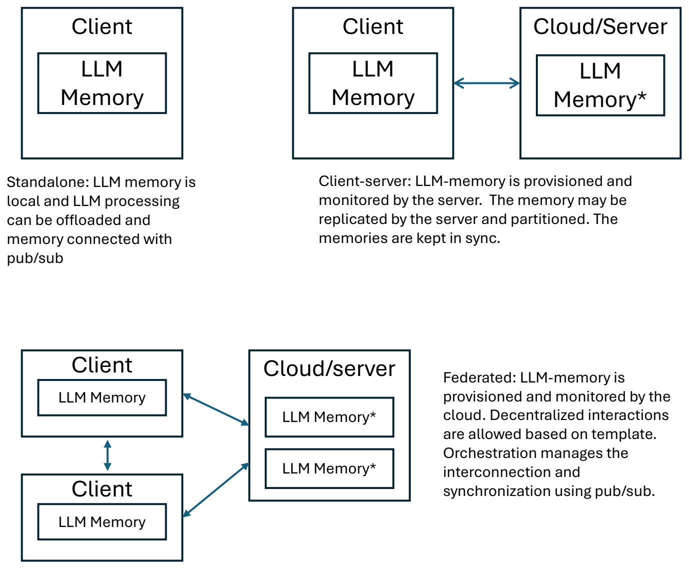

Client
LLM Memory
Standalone: LLM memory is local and LLM processing can be offloaded and memory connected with pub/sub
Client
LLM Memory
Cloud/Server
LLM Memory*
Client-server: LLM-memory is provisioned and monitored by the server. The memory may be replicated by the server and partitioned. The memories are kept in sync.
Client
LLM Memory
Client
LLM Memory
Cloud/server
LLM Memory*
LLM Memory*
Federated: LLM-memory is provisioned and monitored by the cloud. Decentralized interactions are allowed based on template. Orchestration manages the interconnection and synchronization using pub/sub.

Figure 4: Distributed declarative memory for agents

5.1 Instructions

The instruction part of DDMem consists of commands that are categorized into memory setup and initialization, reactive commands, and agent run commands. The design accommodates basic control logic through the commands and it can be extended with agent run commands and LLM callbacks for expressive semantics. The baseline commands are depicted in Table 1.

The following equation defines the read and write operations of the commands,

$$\mathcal{W} = \{(\alpha, \beta) : \alpha \in \mathcal{I}, \beta \in \mathcal{C}\}. \tag{3}$$

Here, $\mathcal{W}$ represents the set of write operations where each operation is a tuple (α, β) consisting of an instruction α from $\mathcal{I}$ and a content update β to a part of the content component $\mathcal{C}$. The reactive and hierarchical structure result in a system, in which a events/instructions trigger writes that can involved multiple read operations. This structure results in DAG call graph for a single memory instance.

The reactive commands having triggers for events and content items are implemented using a breadth-first execution strategy over the call graph created by the read and write operations. The system needs to detect and prevent loops from forming.

5.2 Memory Allocation

During memory allocation, MemoryFormer uses the DDMem model to define memory tasks based on the application specification. It creates memory ele-

Table 1: Event/Operation Descriptions

Event/Operation	**Description**
SUBSCRIBE	Subscribe an event to content anchor
PUBLISH	Publish an event given code trigger is met for content anchors
UNSUBSCRIBE	Unsubscribe event
CLASSIFY	Classify input event into one or more anchors given the specification and the input scopes
FETCH	Retrieve URL or file to anchor
STORE	Save anchors to a file or URL
UPDATE/DELETE	Update or delete content anchor
RAG	Reasoning about the graph with respect to the selected content and write to an anchor (command or react to an event)
SYNC	Established a sync with pub/sub for the given content anchors. This used used by the management plane.
LLM	Perform an LLM operation to selected content and write to an anchor (command or react to an event)
CODE	Execute Python code with selected content items. Write to an anchor) (command or react to an event)
COMPRESS	Compress the anchor with the Language Model (command or react to an event)
PRIVACY	Given the input privacy rules process the selected content and write to an anchor (command or react to an event)
CONTEXT	Given the input, reason how to best use the memory and return context.
AUDIT	Internal auditing instruction for the given anchor generated by the system.

ments containing instructions and content, with fine-tuned instruction prompts for LLM agents. The fine-tuning of the instructions can leverage state of the art methods, such as Chain-of-Thought (CoT), Tree-of-Thought (ToT), and compilation (DSPy).

The GenerateInstructionPrompt function takes the memory task and the LLM language model as input, generating a customized instruction prompt for the LLM agent to follow. The MemoryFormer system includes a prompt tuning component to optimize instruction prompts for LLM agents. This component uses techniques such as few-shot learning and RL optimization to fine-tune the generated instruction prompts, ensuring that LLM agents understand and execute their tasks correctly.

Algorithm 1 Memory Allocation Algorithm

```
1: procedure MemoryAllocation($app_spec$, $\mathcal{L}$)    ▷ $\mathcal{L}$ is the LLM language model
2:     $\mathcal{M} \leftarrow \emptyset$    ▷ Initialize memory objects set
3:     $\mathcal{T} \leftarrow$ DetermineMemoryTasks($app_spec$)    ▷ Determine tasks from the app specification
4:     for $\mathbf{t}_i \in \mathcal{T}$ do
5:         $\mathbf{I}, \mathbf{P} \leftarrow$ GenerateInstructionPrompt($\mathbf{t}_i$, $\mathcal{L}$)
6:         $\mathbf{m}_i \leftarrow$ CreateMemoryObject($\mathbf{t}_i, \mathbf{I}$)
7:         $\mathcal{M}.add(\mathbf{m}_i)$
8:     end for
9:     return $\mathcal{M}$
10: end procedure
```

6 From LLM calls to code

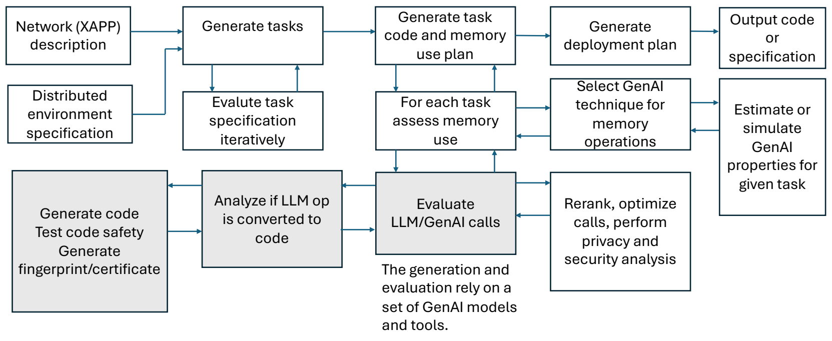
Agent and multi-agent LLM systems face the challenge that LLM operations are costly. This can be alleviated with code generation replacing LLM calls. The code needs to be verified once versus LLMs that are expected to require continuous verification through a sampling policy.

We extend the design with code generation and execution capabilities. State of the art has examples of LLM function calling and LLM generated code being executed by agents. We go beyond state of the art through the managed distributed memory construct that allows declarative and reactive memory with code generation and verification. The paradigm allows to generate, test and verify multiple memories and then deploing those.

The pub/sub nature of the operations enable to update the deployed memory code fragments and runtime while building trust to the updates through the memory verification process.

We have implemented the LLM to code feature using the CODE command. The command CODE that executes the given Python code instruction allowing to use other parts of the memory as inputs. The internal data structures are JSON and the generation process needs to be aware of the structures, for example event structures that are input to the CODE instructions. The result is stored in an anchor as JSON. Only safe Python code is allowed and this is verified during generation and execution. The available Python modules and functions are instrumented and taken into account in the generation process. The verification process is built-in into the memory through internal attestation with the MemoryFormer orchestration.

We can further apply LLM and code reliability analysis and optimization. This is an important feature of the invention. We can run parts/whole command graph and estimate the criticality, reliability, security, resource use, and performance of each step. We can then identify the commands that benefit from code transformation. Human labelling of the results will provide more accuracy to the optimization process.



Network (XAPP) description
Generate tasks
Generate task code and memory use plan
Generate deployment plan
Output code or specification
Distributed environment specification
Evalute task specification iteratively
For each task assess memory use
Select GenAI technique for memory operations
Estimate or simulate GenAI properties for given task
Generate code Test code safety Generate fingerprint/certificate
Analyze if LLM op is converted to code
Evaluate LLM/GenAI calls
Rerank, optimize calls, perform privacy and security analysis
The generation and evaluation rely on a set of GenAI models and tools.

Figure 5: Overview of code generation process

7 Distributed Memory Management and Partitioning

MemoryFormer partitions the memory tasks based on the system model, including agents, LLM availability and placement, cost, and partitionable subgraphs identified through call graph analysis. Subscribe and LLM operations for identified non-overlapping content items are placed close to their sources.

Algorithm 2 Distributed Memory Management and Partitioning Algorithm

```
1: procedure DistributedMemoryManagementAndPartitioning($\mathcal{M}$, $\mathbf{G}$)
2:     $\mathcal{N} \leftarrow$ ListOffloadDestinations()
3:     for $\mathbf{m}_i \in \mathcal{M}$ do
4:         if IsPartitionableSubgraph($\mathbf{m}_i, \mathbf{G}$) then
5:             $\mathbf{n}_j \leftarrow$ FindNearestOffloadWithCapacity($\mathbf{m}_i$)
6:             $\mathbf{n}_j$.AddMemoryObject($\mathbf{m}_i$)
7:         else
8:             $\mathbf{n}_k \leftarrow$ FindNearestOffload($\mathbf{m}_i$)
9:             $\mathbf{n}_k$.AddMemoryObject($\mathbf{m}_i$)
10:        end if
11:    end for
12:    return $\mathcal{N}$
13: end procedure
```

We present a simple cost model to examine the LLM invocation costs of MemoryFormer. MemoryFormer optimizes the cost using two methods: i. memory LLM fine-tuning and model selection, ii. partitioning memory use over the distributed environment and syncing memory fragments across trusted agents. Both optimizations can result in cost savings and improved scalability.

Let N be the number of agents, M the number of content items, C_{LLM} the cost of a single LLM invocation, and f the fraction of content items updated using the pub/sub mechanism. A fraction $1-f$ of the content items are updated independently by each agent.

The total number of LLM invocations I_{total} is:

$$I_{\text{total}} = f \cdot M + (1-f) \cdot M \cdot N$$

The overall cost C_{total} for LLM invocations is:

$$C_{\text{total}} = I_{\text{total}} \cdot C_{\text{LLM}} = (f \cdot M + (1-f) \cdot M \cdot N) \cdot C_{\text{LLM}}$$

7.1 Call Graph Creation

During call graph creation, MemoryFormer uses a directed graph to represent the relationships between read and write operations for memory objects. This information is used to identify partitionable subgraphs that can be offloaded to distributed agents.

```
1: procedure CreateCallGraph($\mathcal{O}$)    ▷ $\mathcal{O}$ is a set of memory operations
   with write and read objects
2:     G ← InitializeDirectedGraph()
3:     for $\mathbf{op}_i \in \mathcal{O}$ do
4:         write_object ← $\mathbf{op}_i$.write
5:         read_objects ← $\mathbf{op}_i$.read
6:         if write_object $\notin$ G.Nodes() then
7:             G.AddNode(write_object)
8:         end if
9:         for read_object $\in$ $\mathbf{read_objects}$ do
10:            if read_object $\notin$ G.Nodes() then
11:                G.AddNode(read_object)
12:            end if
13:            G.AddEdge(read_object, write_object)
14:        end for
15:    end for
16:    return G
17: end procedure
```

7.2 Finding Partitionable Subgraphs

MemoryFormer searches for partitionable subgraphs where all operations lead to a single end write object. These subgraphs can be offloaded to distributed agents.

The partitionable subgraphs offloaded to distributed agents in MemoryFormer exhibit a local property, where all read and write operations, along with their dependencies, are contained within the subgraph itself. This means that any conflicts or cyclic dependencies arising from these operations are resolved locally within the subgraph, without affecting other agents or parts of the system. As a result, each agent executing a partitionable subgraph can ensure correct and consistent execution of its assigned operations, independent of other agents and their respective subgraphs. This localized property enables efficient and scalable execution of reactive instruction processing in MemoryFormer's distributed environment.

```
 1: procedure FindPartitionableSubgraphs($\mathbf{G}$, $\mathcal{E}$)   ▷ $\mathbf{G}$ is the call graph and $\mathcal{E}$ is the set of end write objects
 2:     $\mathbf{P} \leftarrow []$
 3:     for end_obj $\in \mathcal{E}$ do
 4:         $\mathbf{S} \leftarrow$ FindSubgraphNodes($\mathbf{G}$, end_obj)
 5:         if $|\mathbf{S}| > 1$ then
 6:             $\mathbf{P}$.append($\mathbf{G}$.Subgraph($\mathbf{S}$))
 7:         end if
 8:     end for
 9:     return $\mathbf{P}$
10: end procedure
```

7.3 Runtime Optimization

During runtime, MemoryFormer optimizes memory usage by analyzing the system model and providing instructions for using memories in a distributed environment.

Algorithm 3 Runtime Optimization Algorithm

```
1: procedure RuntimeOptimization($\mathcal{N}$)
2:     for $\mathbf{n}_i \in \mathcal{N}$ do
3:         $\mathbf{I}, \mathbf{O} \leftarrow$ AnalyzeAgent($\mathbf{n}_i$)
4:         $\mathbf{R} \leftarrow$ OptimizeMemoryUsage($\mathbf{I}, \mathbf{O}$)
5:         $\mathbf{n}_i$.UpdateInstructions($\mathbf{R}$)
6:     end for
7: end procedure
```

8 Example: Call Graph Generation

A call graph can be generated from the instruction text to represent relationships between read and write operations for memory objects. This information is used to identify partitionable subgraphs that can be offloaded to distributed agents. The following figure shows an example of a call graph generated using this example:

This example demonstrates a DDMem structure with instruction text and contents. The instructions include SUBSCRIBE and LLM commands, while the content consists of KPI values and analyses related to packet loss, latency, and throughput.

The resulting graph has an end node at the KPI_Latency_Analysis write. The predecessors of this end node form three non-overlapping partitions that can be considered for offloading.

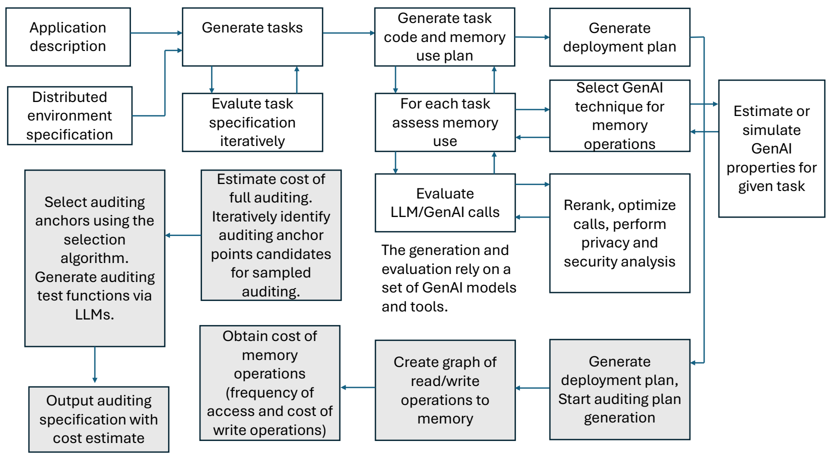
```
operations = [
    {'command': 'subscribe', 'read': 'Latency', 'write:':
    'KPI_Latency'},
    {'command': 'subscribe', 'read': 'Throughput', 'write:':
    'KPI_Throughput'},
    {'command': 'subscribe', 'read': 'Packetloss', 'write:':
    'KPI_Packetloss'},
    {'command': 'LLM', 'read': 'KPI_Latency', 'write':
    'KPI_Latency_Analysis'},
    {'command': 'LLM', 'read': 'KPI_Throughput', 'write':
    'KPI_Throughput_Analysis'},
    {'command': 'LLM', 'read': 'KPI_Packetloss', 'write':
    'KPI_Packetloss_Analysis'},
    {'command': 'LLM', 'read': ['KPI_Latency_Analysis',
    'KPI_Throughput_Analysis','KPI_Packetloss_Analysis'],
    'write': 'KPI_Analysis'}
]
```

9 Auditing LLM Memories

MemoryFormer has a formalism for specifying and verifying memory properties. It provides a declarative way of defining the behavior of a system, making it easier to reason about its properties and verify its correctness. Figure 6 presents an overview of the MemoryFormer architecture. The auditing plan is generated by selecting the memory parts (anchors) that need to be examined after updates, considering the cost of auditing. The system includes various algorithms for LLM memory tasks, anchor point selection, sampling, and auditing instruction generation. Analysis and refinement of the auditing plan ensure the maintenance of an accurate and consistent memory graph.

The auditing process leverages the call graph, which describes the progression of event processing and LLM operations. The auditing process involves traversing the call graph and identifying important anchors in the memory and adding those to the set of elements to be audited. Each element has a corresponding log of operations performed on it according to the call graph. The graph allows different sampling strategies for selecting the auditing candidate elements. We have adopted a straightforward approach by simply extracting the edge nodes (nodes without successors) as candidate elements. These are processing end states and complemented with the log information the system has information to analyze the end result with respect to the processing. This approach may lose some information, which can be mitigated by adding sampled nodes to the candidate set.

The auditing cost can be given if the input event frequencies are known. The graph gives the number of edge nodes, and we can calculate how often they are invoked. For any given edge node, the predecessor subgraph will determine the



Application description
Distributed environment specification
Generate tasks
Evalute task specification iteratively
Generate task code and memory use plan
For each task assess memory use
Evaluate LLM/GenAI calls
The generation and evaluation rely on a set of GenAI models and tools.
Generate deployment plan
Select GenAI technique for memory operations
Rerank, optimize calls, perform privacy and security analysis
Estimate or simulate GenAI properties for given task
Generate deployment plan, Start auditing plan generation
Create graph of read/write operations to memory
Obtain cost of memory operations (frequency of access and cost of write operations)
Estimate cost of full auditing. Iteratively identify auditing anchor points candidates for sampled auditing.
Select auditing anchors using the selection algorithm. Generate auditing test functions via LLMs.
Output auditing specification with cost estimate

Figure 6: Overview of MemoryFormer generation process

update rate. The log sizes affect the memory requirements of auditing. Thus, the memory object with the call frequencies will give us an estimate of the auditing cost. If additional sampling is performed at runtime, this cost needs to be considered as well.

10 Optimizing Multi-agent Memories

We present an algorithm outlined in Figure 7 to improve the efficiency of multi-agent memory systems by identifying and sharing common memory parts among different agents. The input to the algorithm consists of a set of directed graphs G_i, where each graph represents the LLM memory tasks for a particular agent i. Each node in the graph represents a content anchor, which is an updated or input event or trigger, while edges represent read operations from predecessor content anchors. First-level nodes have no predecessors, and edge nodes have no successors.

In the first processing step, the algorithm 4 identifies common shared memory parts among the input memories by comparing their nodes and edges. A common memory part is defined as a set of identical instructions, input data, and content anchors that are present in multiple agents' memories. The identified common memory parts are combined into a joint graph structure G', which updates the weights or costs of the combined structure to reflect the activity and cost of the nodes. Next, the algorithm determines the edge nodes E of G', which represent the final analysis states, and treats them as the candidate auditing set. The algorithm then traverses from these final analysis states towards the first layer nodes, performing possible sampling of additional auditing points. The algorithm may also remove any intermediate nodes that are already marked in an auditing set of one of the agent memories to control the number of auditing

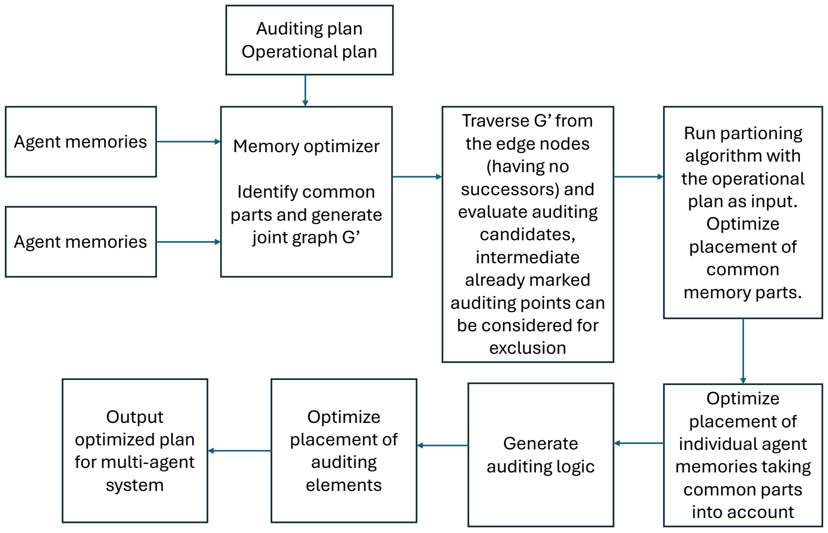

Auditing plan
Operational plan
Agent memories
Agent memories
Memory optimizer
Identify common parts and generate joint graph G'
Traverse G' from the edge nodes (having no successors) and evaluate auditing candidates, intermediate already marked auditing points can be considered for exclusion
Run partioning algorithm with the operational plan as input. Optimize placement of common memory parts.
Optimize placement of individual agent memories taking common parts into account
Generate auditing logic
Optimize placement of auditing elements
Output optimized plan for multi-agent system

Figure 7: Memory auditing for multi-agent systems

points.

In the second processing step, the algorithm runs a partitioning algorithm for G' to place the memory operations in the operating environment, and records the placement of common memory parts in each agent's memory structure and marks them as shared. The algorithm then runs the partitioning algorithm for the individual agent memories to ensure fair distribution of resources. Finally, the algorithm runs the auditing generation process for the common parts and then for the individual memories excluding the common parts. The partitioning algorithm is then run again for the auditing tasks so that they are placed in the operating environment, improving the overall efficiency and effectiveness of the multi-agent memory system with memory sharing.

When working with events and partitions, the system must ensure that there are no loops or harmful concurrent updates to the distributed state. An efficient and easy to way to accomplish this is to partition the subscribe and processing operations and synchronize state via the edge nodes. This combines well with the auditing feature. Thus the partitions are processed at a single node and audited events are generated for subscribers. Additional checks may need to be for additional logic outside the memory; however, the distributed memory itself guarantees the consistency given the specification.

11 Implementation

The MemoryFormer prototype has been implemented with Python using litellm and ollama for LLM capability. The distributed pub/sub is implemented using zqm in dealer/router mode. In this section, we provide two examples of Memo-

Algorithm 4 Optimizing Multi-Agent Memory System with Memory Sharing

```
1: procedure OptimizeMultiAgentMemorySystem($\mathcal{G}$)   ▷ Input: A set of graphs representing agent memories
2:     $\mathcal{C} \leftarrow \emptyset$   ▷ $\mathcal{C}$ is the set of common memory parts
3:     for all $G_i \in \mathcal{G}$ do   ▷ For each graph in $\mathcal{G}$
4:         Identify common memory parts $\mathcal{C}_i$ in $G_i$
5:         $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}_i$
6:     end for
7:     Combine and update weights/costs of nodes in $\mathcal{C}$ to form joint graph structure $G'$
8:     $\mathcal{E} \leftarrow$ Determine edge nodes of $G'$ representing final analysis states
9:     $\mathcal{P} \leftarrow$ Traverse from $\mathcal{E}$ towards first layer nodes, sampling additional auditing points
10:    Remove intermediate nodes already marked in an auditing set of one agent memory (optional)
11:    Place common memory parts in operating environment using a partitioning algorithm and mark as shared
12:    Run partitioning algorithm for individual agent memories to ensure a fair distribution of resources
13:    Generate auditing tasks for common parts $\mathcal{C}$ by sampling nodes from $G'$ based on their weights/costs
14:    Generate additional auditing tasks for individual memories excluding common parts
15:    Place all auditing tasks in operating environment using the same partitioning algorithm to ensure a fair distribution of resources
16: end procedure
```

ryFormer: first an example of a KPI analysis DDMem and then an example of how MemoryFormer uses a DDMem to generate XApps given the instructions.

11.1 KPI Analysis DDMem

```
instruction_text = [
  {"command": "SUBSCRIBE", "event": "Packetloss", "anchor": "KPI_Packetloss"},
  {"command": "SUBSCRIBE", "event": "Latency", "anchor": "KPI_Latency"},
  {"command": "SUBSCRIBE", "event": "Throughput", "anchor": "KPI_Throughput"},
  {"command": "LLM-TRIGGER", "event": "Latency", "datalimit":1, "prompt"
     :"Re-evaluate the latency values and write update latency part for the
     analysis.", "anchor": "KPI_Latency_Analysis", "scope":"KPI_Latency"},
  {"command": "LLM-TRIGGER", "event": "Packetloss", "datalimit":1, "prompt"
     :"Re-evaluate the packet loss values and write update to this part of the
     analysis.", "anchor": "KPI_Packetloss_Analysis", "scope":"KPI_Packetloss"},
  {"command": "LLM-TRIGGER", "event": "Throughput", "datalimit":1, "prompt"
     :"Re-evaluate the throughput values and write update to this part of the
     analysis.", "anchor": "KPI_Throughput_Analysis", "scope":"KPI_Throughput"},
  {"command": "LLM-CONTENT-TRIGGER", "datalimit":1, "prompt" :"Re-evaluate the
     overall network slice status and write detailed analysis.", "anchor":
     "KPI_Analysis", "scope":"KPI_Latency_Analysis KPI_Throughput_Analysis
     KPI_Packetloss_Analysis"},
]

contents_text = {
  "KPI_Packetloss": "",
  "KPI_Latency": "",
  "KPI_Throughput": "",
  "KPI_PacketLoss_Analysis": "",
  "KPI_Latency_Analysis": "",
  "KPI_Throughput_Analysis": "",
  "KPI_Overall_Analysis": "",
}
```

11.2 Generating XApps

```
instruction_text = [
  {"command": "LLM_QUERY", "prompt": "", "anchor": "Tasklist", "scope": "Input
     Init XAPPEvents JSON-Example", "model":"ollama/llama3:70b-instruct"},
  {"command": "LLM_QUERY", "prompt": "", "anchor": "Code", "scope": "Tasks
     Models Tasklist", "model":"ollama/llama3:70b-instruct"},
  {"command": "LLM_QUERY", "prompt": "", "anchor": "Memory-structure", "scope":
     "Memory-prompt Code", "model":"ollama/llama3:70b-instruct"},
  {"command": "LLM_QUERY", "prompt": "", "anchor": "Test", "scope": "Test Code",
     "model":"ollama/llama3:70b-instruct"},
]


contents_text = {
    "XAPPEvents":"Include only XAPP event that are necessary for the tasks. The
     XAPP events are: new_connection_request, incoming_connection_attempt,
     client_connection_request, connection_termination, client_disconnected,
     session_end, data_transfer_start, data_transfer_end, data_transfer_metrics,
     error_occurred, connection_failure, transmission_error, protocol_violation,
     authentication_successful, authentication_failed, configuration_change,
     policy_update, device_setting_change, performance_latency,
     performance_throughput, performance_packet_loss, alert_notification,
     security_alert, threshold_breach, system_notification, resource_allocation,
     resource_deallocation, qos_priority_change, qos_traffic_shaping,
     firmware_update, software_update, user_activity_log,
     unauthorized_access_attempt, ddos_attack, malware_detection,
     topology_change, device_addition, device_removal, bandwidth_usage_report,
     device_health_status, cpu_usage_report, memory_usage_report,
     operational_status_update",
    "Init": (
        "Generate application memory tasks given the following specification in
     simple command format for further "
        "processing. The memory has instructions and content. The tasks use
     commands that operate on named content items. "
        "The JSON structure is a list of commands that have parameters."
        "You can use intermediate content anchors to store and analyze data.
     This will help in partitioning of the application."
        "USE ONLY THESE COMMANDS AND PARAMETERS and OUTPUT JSON. Use step by
     step reasoning."
        "The command parameters are:"
        "event is a label for event type(only one not a list), "
        "prompt is a prompt string, "
        "anchor is a string, always the content destination,"
        "scope is a string of one or more content items for the given command
     separated by whitespaces,"
        "model is the LLM model name to be used, "
        "datalimit is an integer threshold size that must be be exceeded to
     process content,"
        "The commands with their allowed parameters are: "
        "SUBSCRIBE event, anchor;"
        "UNSUBSCRIBE event;"
        "LLM-QUERY prompt, anchor, scope, model, datalimit;"
        "LLM-TRIGGER event, prompt, anchor, scope, model, datalimit;"
        "LLM-CONTENT-TRIGGER prompt, anchor, scope (content changed), model,
     datalimit;"
        "COMPRESS anchor, prompt, datalimit;"
        "FETCH url, anchor."
       "Notes: scope can be used to include prompt content to LLM commands."
    ),
    "Test" : "Output test events for the given reactive JSON program. ",
    "Tasks": """
    Given the input JSON task descriptions with operations examine the prompts
     for clarity and correctness. Output only JSON and ONLY MODIFY PROMPTS. DO
     NOT OUTPUT THOUGHTS.
    """,
    "JSON-Example":"""Here is an example JSON:
    instruction_text_ = [
    {"command": "SUBSCRIBE", "event": "AA", "anchor": "AAA"},
    {"command": "SUBSCRIBE", "event": "BB", "anchor": "BBB"},
    {"command": "LLM-TRIGGER", "event": "AA", "datalimit":100, "prompt"
     :"Re-evaluate the latency values and write update latency part for the
     analysis.", "anchor": "AAAA", "scope":"AAA"},
    {"command": "LLM-TRIGGER", "event": "BB", "datalimit":100, "prompt"
     :"Re-evaluate the packet loss values and write update to this part of the
     analysis.", "anchor": "BBBB", "scope":"BBB"},
    {"command": "LLM-CONTENT-TRIGGER", "datalimit":100, "prompt" :"Re-evaluate
     the overall network slice status and write detailed analysis.", "anchor":
     "CCCC", "scope":"AAAA BBBB"},
  ]""",
    "Memory-prompt:":"Output the memory structure of the input code in JSON. The
     output needs to be in the form of available content items (anchors) and
     their brief description.",
    "Input": (
        "Develop a lightweight Network Event Analyzer XAPP that subscribes to
     network quality events, processes these "
        "events, and performs basic analysis to provide insights. The XAPP
     should subscribe to events such as connection "
        "requests, disconnections, data transfers, and error occurrences."
        "The XAPP will implement analysis functions to "
        "count event types, calculate average data transfer sizes, identify peak
     times for connections and disconnections, "
        "and detect error patterns. It will generate summary reports."
    ),
    "Models": "Use model: ollama/mixtral:instruct",
    "Example":"Here is an example.."
}
```

12 Code Generation Experiment

The code generation has been implemented for the Python MemoryFormer framework. The Python code is executed in a safe environment and the Python modules and features available can be controlled. The concept has been prototyped with memory creation and then instantiating the generated LLM and code memories. The MemoryFormer uses Llama3.1-70b to generate the memory.

Input instruction to MemoryFormer: Objective: Develop a lightweight Network Event Analyzer xApp that subscribes to network quality events, processes these events, and performs basic analysis to provide insights. The xApp should subscribe to data transfer events. The xApp will implement analysis functions to calculate average data transfer sizes and detect data size anomalies. It will generate a summary report in written format. Operate on the 4 latest events that are extracted from the subscription anchor. Events are in an ordered list.

LLM-based generated implementation:

```
[
  {
    "command": "SUBSCRIBE",
    "event": "data_transfer",
    "anchor": "DataTransferEvents"
  },
  {
    "command": "LLM-TRIGGER",
    "event": "data_transfer",
    "prompt": "Extract the 4 latest data transfer events from the
     DataTransferEvents anchor, calculate their average size, and detect any
     anomalies in these events.",
    "anchor": "LatestDataTransferAnalysis",
    "scope": "DataTransferEvents",
  },
  {
    "command": "LLM-CONTENT-TRIGGER",
    "prompt": "Generate a summary report on the average data transfer size and
     any anomalies detected in the latest events. Include recommendations for
     improvement.",
    "anchor": "SummaryReport",
    "scope": "LatestDataTransferAnalysis DataTransferEvents"
  }
]
```

Code-based generated implementation:

```
[
  {
    "command": "SUBSCRIBE",
    "event": "data_transfer",
    "anchor": "DataTransferEvents"
  },
  {
    "command": "CODE-TRIGGER",
    "event": "data_transfer",
    "code": (
      "def main_function(DataTransferEvents):\n"
      "    # Extract the 4 latest data transfer events, calculate their average
     size, and detect any anomalies in these events.\n"
      "    import statistics\n"
      "    latest_events = sorted(DataTransferEvents, key=lambda x:
     x['timestamp'], reverse=True)[:4]\n"
      "    sizes = [event['size'] for event in latest_events if 'size' in
     event]\n"
      "    avg_size = sum(sizes) / len(sizes)\n"
      "    anomalies = [event for event in latest_events if event['size'] > 1.5
     * avg_size or event['size'] < 0.5 * avg_size]\n"
      "    return {'average_size': avg_size, 'anomalies': anomalies}\n"
    ),
    "scope": "DataTransferEvents",
    "anchor": "LatestDataTransferAnalysis"
  },
  {
    "command": "CODE-CONTENT-TRIGGER",
    "code": (
      "def main_function(LatestDataTransferAnalysis, DataTransferEvents):\n"
      "    # Generate a summary report on the average data transfer size and any
     anomalies detected in the latest events.\n"
      "    import statistics\n"
      "    avg_size = LatestDataTransferAnalysis['average_size']\n"
      "    anomalies = LatestDataTransferAnalysis['anomalies']\n"
      "    recommendations = 'No recommendations'\n"
      "    if len(anomalies) > 0:\n"
      "        recommendations = 'Check network configuration for unusual
     traffic patterns'\n"
      "    return {'summary_report': f'Average data transfer size: {avg_size}
     bytes', 'recommendations': recommendations}\n"
    ),
    "scope": "LatestDataTransferAnalysis DataTransferEvents",
    "anchor": "SummaryReport"
  }
]
```

Benchmarking with 20 randomly generated events processed in 10 separate benchmark runs on MacBookPro M2 Max, 64 GB RAM, with ollama/llama3:instruct.

Timings for part 1 (LLM): [563712.8408203125, 595450.8620605469, 565318.1921386719, 620468.8579101562, 547847.5842285156, 696699.212890625, 576333.537109375, 562984.8093261719, 561458.0251464844, 580247.8249511719]

Standard deviation for part 1: 41423.740780853645 ms

Timings for part 2 (code): [22.06005859375, 14.67822265625, 13.4931640625, 12.7197265625, 11.537109375, 10.910888671875, 10.558349609375, 10.445068359375, 9.968017578125, 9.90673828125] Standard deviation for part 2: 3.4870043439955656 ms

The code version is significantly faster than the LLM version indicating that code-only or hybrid managed programs are desirable for cost and resource conscious operation.

13 MemoryFormer template for code generation

```
instruction_text = [
  {"command": "LLM_QUERY", "prompt": "", "anchor": "Tasklist", "scope": "Input Init XAPPEvents JSON-Example Example-events", "model":"ollama/llama3.1:70b"},
  {"command": "LLM_QUERY", "prompt": "", "anchor": "Code", "scope": "Tasks Models Tasklist Input", "model":"ollama/llama3.1:70b"},
  {"command": "LLM_QUERY", "prompt": "", "anchor": "Code2", "scope": "Code-opt Code Python Example-events", "model":"ollama/llama3.1:70b"},
]

contents_text = {
    "AppEvents":"Include only xApp events that are necessary for the tasks. The xApp events are: new_connection_request, incoming_connection_attempt, client_connection_request, connection_termination, client_disconnected, session_end, data_transfer_start, data_transfer_end, data_transfer_metrics, error_occurred, connection_failure, transmission_error, protocol_violation, authentication_successful, authentication_failed, configuration_change, policy_update, device_setting_change, performance_latency, performance_throughput, performance_packet_loss, alert_notification, security_alert, threshold_breach, system_notification, resource_allocation, resource_deallocation, qos_priority_change, qos_traffic_shaping, firmware_update, software_update, user_activity_log, unauthorized_access_attempt, ddos_attack, malware_detection, topology_change, device_addition, device_removal, bandwidth_usage_report, device_health_status, cpu_usage_report, memory_usage_report, operational_status_update",
    "Init": (
        "Generate application memory tasks given the following specification in simple command format for further "
        "processing. The memory has instructions and content. The tasks use commands that operate on named content items. "
         "The prompts will have access to the identified data items (called scopes). These are appended to the prompt as context."
        "The JSON structure is a list of commands that have parameters."
        "You can use intermediate content anchors to store and analyze data. For
     example selecting a subset of events. Do not mix content objects in JSON
     anchors. This will help in partitioning of the application."
        "You can also use trigger conditions for triggering further actions,
     such as complex LLM processing. This will help in later optimization."
        "USE ONLY THESE COMMANDS AND PARAMETERS and OUTPUT JSON. Use step by
     step reasoning."
        "Important: SUBSCRIBE all events you use in the commands."
        "The command parameters are:"
        "event is a label for event type (not a list), "
        "prompt is a prompt string, "
        "anchor is a string, always the content destination,"
        "scope is a string of one or more content items for the given command
     separated by whitespaces,"
        "model is the LLM model name to be used. Only use the provided LLM model
     or omit this. "
        "The commands with their allowed parameters are: "
        "SUBSCRIBE event, anchor;"
        "UNSUBSCRIBE event;"
        "LLM-QUERY (only used in init stage) prompt, anchor, scope, model,
     datalimit;"
        "LLM-TRIGGER event, prompt, anchor, scope, model, datalimit;"
        "LLM-CONTENT-TRIGGER prompt, anchor, scope (content changed), model,
     datalimit;"
       "Notes: scope can be used to include prompt content to LLM commands."
       "Scopes are useful in reading information, but they are also costly and
     change in any of the scopes will trigger an update."
       "Anchor write will replace the content. Do not implement multiple writes
     to the same anchor without updating the whole anchor."
    ),
# ---------------
    "Test" : "Output test events for the given reactive JSON program. The events
     are used to test the program.",
    "Tasks": """
    Given the input JSON task descriptions with operations examine the prompts
     for clarity and correctness.
    The prompts are provided within the JSON commands. Expand prompts  to make
     them more instructive and accurate for the requested functionality.
    The prompts will have access to the identified data items (called scopes).
     These are appended to the prompt as context.
    Provide sufficient instruction for the LLM to process the command and write
     the content without additional remarks or thoughts.
    The LLM should behave like an information processor implementing the planned
     specification.
    Output only continuous JSON and ONLY MODIFY PROMPTS. DO NOT OUTPUT THOUGHTS."
    Start output from '[',
    """,
# ---------------
    "JSON-Example":"""Here is an example JSON:
    instruction_text_ = [
    {"command": "SUBSCRIBE", "event": "AA", "anchor": "AAA"},
    {"command": "SUBSCRIBE", "event": "BB", "anchor": "BBB"},
    {"command": "LLM-TRIGGER", "event": "AA", "prompt" :"Re-evaluate the latency
     values and write update latency part for the analysis.", "anchor": "AAAA",
     "scope":"AAA"},
    {"command": "LLM-TRIGGER", "event": "BB", "prompt" :"Re-evaluate the packet
     loss values and write update to this part of the analysis.", "anchor":
     "BBBB", "scope":"BBB"},
    {"command": "LLM-CONTENT-TRIGGER", "prompt" :"Re-evaluate the overall network
     slice status and write detailed analysis.", "anchor": "CCCC", "scope":"AAAA
     BBBB"},
  ]""",

# ---------------

    "Code-opt-final":"Consider the CODE elements of the JSON and carefully
     ensure that the code is executable."
    "Do not modify anything except the Python code. Do not modify JSON encoding.
     The input parameters are given in a list of dictionaries in the order of
     the scope entries. Scope field identifies the parameters."
    "Output only continuous JSON and pay particular attention to multiline
     strings."
    "Ensure that the final output can be parsed as JSON and the Python code in
     JSON is correctly formatted.",

# ---------------
    "Code-opt":"Consider each LLM invocation prompt provided after these
     instructions (LLM-QUERY, LLM-TRIGGER, LLM-CONTENT-TRIGGER) and if the
     prompt is implementable in safe basic Python, implement the LLM operation. "
    "If so, change the command to one of the code commands. Include all
     commands, do not change other commands than requested."
    "The commands are:"
    "scope denotes the params in all code fields (param1 param2...). Only these
     are available."
    "CODE code scope anchor, where scope denotes the params (param1 param2...).
     Not used for reactive events, only used in init. "
    "CODE-TRIGGER event code scope anchor, where event is the trigger."
    "CODE-CONTENT-TRIGGER code scope anchor, where scope denotes the parameters
     and content triggers"
    "The parameter code denotes executable safe Python code. Output only
     continuous JSON and pay particular attention to multiline strings. "
    "Strip Leading and Trailing Whitespace: Ensures no extraneous whitespace at
     the beginning or end of the string."
double quotes and removes them if present."
    "Replace Multiple Spaces: Replaces sequences of two or more spaces with a
     single space."
    "Remove Extraneous Whitespace Around Delimiters: Removes spaces around JSON
     delimiters like braces ({}), brackets ([]), colons (:), and commas (,)."
    "Remove Control Characters: Removes any remaining control characters that
     may cause issues."
    "Use regular string delimiters with proper escaping for the code part.
     Output full code for the functionality."
    "Start output from '['"
    "You have to have the def main_function(param1, param2): in the code, the
     params are the scopes. The input parameters are given in a list of
     dictionaries. "
    "Ensure that the anchor writes and reads are aligned in terms of the JSON
     data structure. If you write a list of dicts to an anchor, you need to then
     use the same format when reading."
    "Do not use import. Here is an example of Python code:"
"""
[
    {
        "command": "CODE-TRIGGER",
        "event": "data_transfer",
        "code": (
            "def main_function(latest_events):\n"
            "    # Extract the 5 most recent data transfer events, including
     their timestamps and sizes.\n"
            "    data_transfers = [event for event in latest_events if
     event['type'] == 'data_transfer']\n"
            "    return {'events': data_transfers[:5]}"
        ),
        "scope": "latest_events",
        "anchor": "processed_events"
    }
]
Use dictionaries to store results.
You must use exactly the same term as parameter in the main_function as
     specified in the scope.
""",
"\nOutput the full JSON app without thoughts and carefully check JSON notation."
"Do not escape. Start output from '['",
# ---------------
    "Memory-prompt:":"Instructions: Output the memory structure of the input code
     in continuous JSON. The output needs to be in the form of available content
     items (anchors) and their brief description.",
    "Input": (
        "Objective: Develop a lightweight Network Event Analyzer xApp that
     subscribes to network quality events, processes these "
        "events, and performs basic analysis to provide insights."
        "The xApp should subscribe to data transfer events. "
        "The xApp will implement analysis functions to "
        "calculate average data transfer sizes and detect data size anomalies. "
        "It will generate a summary report in written format. Operate on the 4
     latest events that are extracted from the subscription anchor. Events are in
     an ordered list. "
    ),
    "Models": "Use LLM model: ollama/llama3.1:70b",
    "Example":"Here is an example.",
    "Python":"""
    # ONLY USE THESE Python features. DO NOT import modules.
    SAFE_BUILTINS = {
    'abs': abs,
    'divmod': divmod,
    'pow': pow,
    'round': round,
    'len': len,
    'str': str,
    'format': format,
    'append_to_list': lambda lst, item: lst.append(item),  # Add append function
    'sum': sum  # Add sum function
}

SAFE_MATH = {
    'pi': math.pi,
    'e': math.e,
    'sqrt': math.sqrt,
    'sin': math.sin,
    'cos': math.cos,
    'tan': math.tan,
    'log': math.log,
    'log10': math.log10,
    'exp': math.exp,
    'factorial': math.factorial
}

SAFE_STATISTICS = {
    'mean': statistics.mean,
    'median': statistics.median,
    'mode': statistics.mode,
    'stdev': statistics.stdev
}

SAFE_STRING_METHODS = {
    'lower': str.lower,
    'upper': str.upper,
    'capitalize': str.capitalize,
    'title': str.title,
    'split': str.split,
    'join': str.join,
    'replace': str.replace
}

SAFE_RE = {
    'compile': re.compile,
    'match': re.match,
    'search': re.search,
    'findall': re.findall,
    'sub': re.sub,
    'split': re.split
}

SAFE_COLLECTIONS = {
    'defaultdict': defaultdict
}

SAFE_PANDAS = {
    'DataFrame': pd.DataFrame,
    'Series': pd.Series
}

SAFE_GLOBALS = {
    'sorted': sorted,
    '__builtins__': SAFE_BUILTINS,
    'math': SAFE_MATH,
    'statistics': SAFE_STATISTICS,
    'str_methods': SAFE_STRING_METHODS,
    're': SAFE_RE,
    'collections': SAFE_COLLECTIONS,
    'pandas': SAFE_PANDAS
}

Note: print is not available.
""",
    # ---------------

    "Example-events" : """Here are example events.
    [
  {
    "type": "connection",
    "timestamp": "2024-07-29T12:34:56Z",
    "source_ip": "192.168.1.1",
    "destination_ip": "192.168.1.100",
    "protocol": "TCP",
    "port": 8080
  },
  {
    "type": "disconnection",
    "timestamp": "2024-07-29T12:45:00Z",
    "source_ip": "192.168.1.1",
    "destination_ip": "192.168.1.100",
    "protocol": "TCP",
    "port": 8080,
    "duration": 600
  },
  {
    "type": "data_transfer",
    "timestamp": "2024-07-29T13:00:00Z",
    "source_ip": "192.168.1.2",
    "destination_ip": "192.168.1.101",
    "protocol": "UDP",
    "port": 9000,
    "size": 1500
  },
  {
    "type": "error",
    "timestamp": "2024-07-29T13:05:30Z",
    "source_ip": "192.168.1.3",
    "destination_ip": "192.168.1.102",
    "protocol": "TCP",
    "port": 443,
    "error_code": "CONNECTION_TIMEOUT",
    "error_message": "Connection timed out after 30 seconds"
  }
]
"""
}
```

14 Sub2Code, Pub2Code and Intent2Code

Next, we employ the MemoryFormer to implement a complex event processing system. The MemoryFormer is a natural building block for complex event processing since it can accommodate natural language input, event schemas and then transform these into executable LLM operations and code. The distributed and managed nature helps in the deployeemnt of the event detection capabilities.

We outline three key components:

- Sub2Code: generates the structure for subscribing the requested complex event based on the gven input. MemoryFormer generates the specification for subcribing the elementary components and the detection logic.

- Pub2Code: generates the elementary publications given a description of the publication.
- Intent2Code: generates a specification for realizing the given intent that may consist of a set of subscribe and publish operations.

The system can be further enhanced by a schema knowledgebase for the events and creating meta-filters from the generated filters.

15 Conclusion

MemoryFormer provides an innovative solution for managing memories in large-scale distributed systems. By incorporating the Declarative LLM Memory model, call graph creation, finding partitionable subgraphs, and offloading memory tasks to distributed agents, this system effectively allocates and manages memories while optimizing performance and resource utilization. The presented algorithms demonstrate MemoryFormer's main components and features. MemoryFormer includes the automatic generation of auditing plans for individual agent memories and multi-agent systems.

References

[1] Weize Chen, Ziming You, Ran Li, Yitong Guan, Chen Qian, Chenyang Zhao, Cheng Yang, Ruobing Xie, Zhiyuan Liu, and Maosong Sun. Internet of agents: Weaving a web of heterogeneous agents for collaborative intelligence, 2024.

[2] Qifei Dong, Xiangliang Chen, and Mahadev Satyanarayanan. Creating edge AI from cloud-based llms. In Nigel Davies and Chenren Xu, editors, *Proceedings of the 25th International Workshop on Mobile Computing Systems and Applications, HOTMOBILE 2024, San Diego, CA, USA, February 28-29, 2024*, pages 8–13. ACM, 2024.

[3] Zafeirios Fountas, Martin A Benfeghoul, Adnan Oomerjee, Fenia Christopoulou, Gerasimos Lampouras, Haitham Bou-Ammar, and Jun Wang. Human-like episodic memory for infinite context llms, 2024.

[4] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[5] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and

Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023.

[6] Daliang Li, Ankit Singh Rawat, Manzil Zaheer, Xin Wang, Michal Lukasik, Andreas Veit, Felix Yu, and Sanjiv Kumar. Large language models with controllable working memory. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1774–1793, Toronto, Canada, July 2023. Association for Computational Linguistics.

[7] Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *CoRR*, abs/2310.08560, 2023.

[8] John Chong Min Tan, Prince Saroj, Bharat Runwal, Hardik Maheshwari, Brian Lim Yi Sheng, Richard Cottrill, Alankrit Chona, Ambuj Kumar, and Mehul Motani. Taskgen: A task-based, memory-infused agentic framework using strictjson, 2024.

[9] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[10] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19724–19731, Mar. 2024.