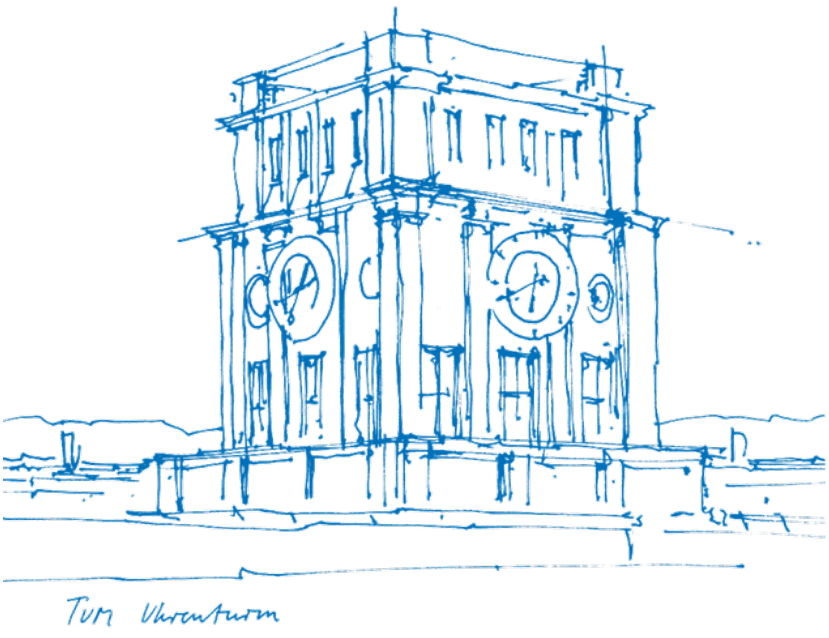
Research Group Social Computing
Department of Computer Science
TUM School of Computation, Information and Technology
Technical University of Munich

Master's Thesis in Information Systems

Alexander Blatzheim

Cognitive Theory for Deliberate Reasoning: Synthesizing Working Memory Architectures for Large Language Models



Research Group Social Computing
Department of Computer Science
TUM School of Computation, Information and Technology
Technical University of Munich

Master's Thesis in Information Systems

Alexander Blatzheim

Cognitive Theory for Deliberate Reasoning: Synthesizing Working Memory Architectures for Large Language Models

Kognitive Theorie für zielgerichtetes Denken:
Synthese von Arbeitsgedächtnisarchitekturen für große Sprachmodelle

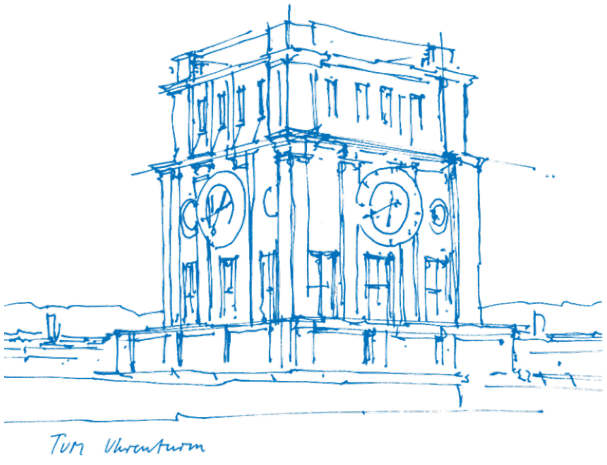
Thesis for the Attainment of the Degree
Master of Science

at the TUM School of Computation, Information and Technology,
Department of Computer Science,
Research Group Social Computing

Examiner
Prof. Dr. Georg Groh

Supervised by
Tobias Eder

Submitted on
29.07.2025



Abstract

Contemporary Large Language Models (LLMs) arguably lack mechanisms for dynamically regulating their expanding reasoning context – the essential prerequisite for supervisory "System 2" cognition. Graph-of-Thought (GoT), a recent paradigm that simulates reasoning via static prompting graphs, suffers from the identical limitation, in reverse: Thoughts are conditioned solely on immediate predecessors. Consequently, insights from distant or parallel paths remain inaccessible, leaving its stated motivation unfulfilled.
This recalls parallels to working memory (WM), in a second disconnect: While agentic systems invoke the term "working memory", the dominant cognitive model, the Multi-Component Model (MCM), appears absent from LLM research, despite offering a rich and modular design space for managing limited, task-relevant context – directly aligned with LLM reasoning challenges.

Our work bridges both gaps: First, we develop a software-architectural interpretation of the MCM to ground further WM design. Next, we synthesize an extendable WM core architecture for LLMs, agnostic to task and integration specifics. We then implement four task-agnostic WM variants, along two axes: Memory structure (flat vs. tree) and retrieval backend (LLM-only vs. embedding-hybrid), each accumulating reflections on GoT thoughts as primary memory content. These are integrated into GoT and evaluated for their ability to distill reasoning context across expanding graphs.

Regrettably, GoT benchmark tasks proved unstable, likely due to insufficient natural language complexity. Thus, our findings remain inconclusive. However, we observe qualitative indications: LLM-only variants are more flexible in structuring content, while responsive to reasoning context shifts. Here, the tree-based variant appears most capable of organizing content, overall. Token usage analysis further highlights a trade-off: LLM-based variants incur the highest token overhead, while embedding-hybrids exhibit more rigid retrieval dynamics. These findings suggest a hybrid strategy, carefully delegating high-flexibility operations to LLMs and routine to embedding methods.

Overall, we offer a first grounded MCM-WM design for LLMs, charting a path for extension into a full MCM analogue. Long-term, we envision WM as a catalyst for System 2-style supervision in next-generation LLMs. In support, we outline two frontiers: Storage modalities and integration environments, on the token-level and in agentic systems already invoking WM ad hoc.

Contents

List of Tables

List of Figures

Chapter 1

Prelude: Fundamental Concepts

This Prelude chapter briefly outlines fundamental concepts in natural language processing (NLP) relevant to the following work. For a comprehensive introduction, we point readers to standard overviews such as Jurafsky and Martin (2025). If you are familiar with the state of the art in NLP around 2025, feel free to skip this chapter.

All models discussed here are based on the Transformer architecture (Vaswani et al., 2023), which introduced attention-based mechanisms to model context across entire input sequences in parallel. We do not describe the architecture further, but instead refer to Jurafsky and Martin (2025, Chapter 9).

1.1 Embedding Models

A core idea in NLP is that of representing words, sentences, or entire documents as dense vectors in a high-dimensional space, known as *embeddings* (Jurafsky & Martin, 2025, Chapter 6). These allow language to be processed geometrically.

A prominent example is *BERT* (Devlin et al., 2019), which learns contextual embeddings of tokens by training on large corpora using masked language modeling (see Jurafsky and Martin (2025, Chapter 11)). Given a sentence, BERT returns a fixed-size vector representation that captures syntactic and semantic information.

The similarity between two embeddings is often quantified using *cosine similarity*, which measures the angle between two vectors. It ranges from –1 (opposite) to 1 (identical), and is invariant to vector length. Specifically, cosine similarity between vectors $\vec{a}$ and $\vec{b}$ is defined as:

$$\text{cos\_sim}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \, \|\vec{b}\|}$$

This metric is widely used because semantic meaning in embeddings is often encoded in the **direction** of the vector, while the **magnitude** may reflect factors such as intensity or frequency. In other words, cosine similarity captures *what* is being said, not *how strongly*. That said, the choice of similarity metric is task-dependent: for instance, in some cases one might treat diametrically opposed vectors (i.e., 180° apart) as semantically equivalent if negation is considered irrelevant (Jurafsky & Martin, 2025, Chapter 6.4).

Modern embedding models are frequently trained using *contrastive learning*, a method that "pulls" semantically similar items closer in the embedding space, while pushing dissimilar ones apart. For overviews see T. Gao et al. (2022) and; with application in Schopf et al. (2024).

1.2 Large Language Models (LLMs)

Large Language Models (Jurafsky & Martin, 2025, Chapter 10) operate on *tokens*: Discrete units derived from text via a so-called tokenizer, in essence, corresponding to subword representation (e.g., "un", "believ", "able").

Large Language Models are generally trained to perform *next-token prediction*: Given a sequence of tokens (e.g., "The", "sky", "is"), the model outputs a probability distribution over all possible tokens, indicating the most likely continuation. This is then fed back into the model to generate the next token, and so on. This process is called *autoregressive* generation.

When the next token is generated, the *temperature* parameter controls randomness during decoding: low temperatures yield more deterministic outputs, while higher values (e.g., 1.0) introduce variability and creativity.

LLMs are typically pre-trained on vast text corpora and optionally fine-tuned on curated datasets or specific tasks (e.g., instruction following, coding, summarization).

1.3 Reasoning LLMs

Reasoning-capable LLMs extend the generation process of their predecessors by first outputting a *Chain-of-Thought* (CoT; Wei et al. (2023) see Section 3.2.1.2). The idea being that, since a LLM always selects the most likely continuation given the preceding context, explicitly generating intermediate reasoning steps can steer the model toward better final answers. A more deliberate and logically structured context makes the correct answer more likely to emerge as the most probable continuation.

In practice, CoT transforms one-step responses into multi-step derivations. For example, given the question: *Q: Alice has 5 apples. She buys 3 more and then gives 2 to Bob. How many does she have left?* a standard LLM might respond in a single step: *A: 6*. In contrast, a CoT-enabled model would first articulate the intermediate reasoning: *A: She starts with 5 apples. After buying 3 more, she has 8. Giving 2 to Bob leaves her with 6.*

By externalizing this reasoning trace, the model creates a scaffold that helps reduce error and encourages coherence. This shift from direct answer generation to structured reasoning has been

shown to improve accuracy significantly, particularly in models with sufficient scale. A prominent example is DeepSeek-AI et al. (2025).

Their training can broadly be summarized in three main stages (cf. (DeepSeek-AI et al., 2025)):

1. **Foundation Pre-Training**: Unsupervised training on raw internet-scale data.
2. **Instruction Finetuning**: Supervised fine-tuning maps reasoning prompts directly to correct final answers.
3. **CoT Finetuning**: A second fine-tune with annotated step-by-step CoT's trains the model to generate explicit reasoning traces.

However reasoning LLMs still face limitations in advanced reasoning tasks, including "overthinking and the risk of propagating errors", as noted by Ferrag et al. (2025), who also argue that that "exposing language models to the full spectrum of search behaviors – including errors – enables them to learn more robustly and self-correct" during training (cf. Gandhi et al. (2024)).

1.4 Agentic Systems

Agentic systems wrap LLMs in an execution environment where they can plan, act, and reflect over time. While LLMs operate in a single forward pass, agentic setups typically incorporate *memory* or *tool use*, and are generally characterized by autonomous *goal orientation*, making them more capable of handling extended tasks (cf. Section3.3.2.1).

Although the focus of this work lies on memory, agentic systems have appeared as the predominant application paradigm for LLMs by the time of writing. To give a snapshot of contemporary agentic services:

- **Coding Agents:** Cursor (cursor.sh).
- **Desktop/Browser Agents:** OpenAI Operator (openai.com), Manus (manus.ai), Rewind.ai (rewind.ai)
- **Multi-Agent Coordination:** CrewAI (crewai.com), AutoGen (Yang et al., 2024)

Chapter 2

Introduction

2.1 Motivation

2.1.1 Point-of-View: The Dual Process Theory of Thought

We invite readers to engage with this short demonstration of the *Stroop Effect* (Stroop, 1935):

⚠ **Warning**: Your brain is about to betray you.

You have 10 seconds.

Read these words as fast as you can. Don't think – just go!

blue	***orange***	***green***	***blue***
orange	***black***	***purple***	***orange***
pink	***green***	***red***	***black***
green	***blue***	***red***	***pink***

Now again – but this time, don't read the words.

Say only the *color* they are displayed in.

Ready? Go!

blue	***orange***	***green***	***blue***
orange	***black***	***purple***	***orange***
pink	***green***	***red***	***black***
pink	***purple***	***purple***	***red***

If you hesitated, stumbled, or felt your mind momentarily jam – good.

The second attempt should have been significantly harder indeed.

What you may have experienced was not a matter of personal failure. It was a predictable and well studied cognitive conflict between two distinct modes of mental operation (Scarpina & Tagini, 2017). Reading familiar words is an *intuitive process*: It happens rapidly, effortlessly, and without conscious deliberation. Naming the ink color, however, requires *attentional control*.

This interference exemplifies a broader principle in the so-called *dual process theory*: That (human) thought operates via two fundamentally different systems (cf. Kahneman (2012)).[1]

System 1: Intuition System 1 is the floodgate operator that funnels raw streams of thought material into your mind, to provide the basis material for all thinking. It is:

- *Running on autopilot* – extremely hard to shut down (this is called meditation). If System 1 is not running in the background, we can consider thinking to be impossible.
- A retrieval agent driven by *external stimuli* – in the example above, the visual cues of written words – these can be seen as query signals for "thought material".
- Operating on *heuristic similarity* between these stimuli, and the material stored in memory.

Specifically, in the first Stroop box, it *automatically* processed a visual stimulus such as ***blue***, and associated it *intuitively* with the sound of "blue", stored somewhere in your brain.

Consider this second example:

Read the following words quickly:

`fork knife plate dinner napkin eat`

Now, what is the first word that comes to mind when you see this?

`S _ _ P`

Did you say **SOUP**?

[1] Unless stated otherwise, all statements regarding System 1 and System 2 in this section follow the framework introduced by Kahneman (2012), which serves as our continuous point of reference throughout.

If so, you're not alone.

Most people do – when they've just been exposed to food-related words.

But if a different group sees this sequence:

`shower  towel  clean  brush  bathe  wash`

they tend to complete `S _ _ P` as **SOAP**.

This is *priming* in action: We presented other *external stimuli* to your System 1, which *remain in context*, an dilute the subsequently presented `S _ _ P` stimuli. You didn't consciously decide to associate "S_ _ P" with soup or soap – *System 1* made the call for you.

The Intuition Machine In this regard, large language models (LLMs) may be seen as a remarkably close computational analogue of System 1. Both operate as *intuition machines*: Associative engines that respond to a window of external stimuli to associatively drawn from a vast internal memory. Like System 1, LLMS:

- Operate on the basis of externally provided stimuli – the input tokens in their context window.
- They "retrieve" output tokens through a forward pass that heuristically retrieves the next most likely token, based on "prior experience".
- They are subject to priming[2]: Contextual cues in the input can systematically bias their predictions, much like associative primes bias human intuition.[3]

Critically, both LLMs and System 1 can be *trained* via gradual restructuring of intuitive associations; which for LLMs, corresponds to weight updates via gradient descent. However, this first requires deliberate *attentional control* – enter System 2.

[2] In the wider conceptual sense, but also specifically in regards to small input changes. Cf. (Salinas & Morstatter, 2024).

[3] Trivially, the "autopilot" aspect of System 1 can be omitted, since if this does not happen; the LLM is not operating – though, supposedly, it is not meditating.)

System 2: Navigating Failure Recall our initial demonstration. When you encountered stimuli like ***red*** or ***green***, your System 1 began to fail – it could not process the stimuli reliably and produced hesitation, interference, or outright error. At that moment, System 2 was recruited:

- It inhibits the automatic execution of intuitive responses, when System 1 starts to jam.
- Portions the stimuli that System 1 could not process, so that it may.
- It resumes the execution of automatic intuition, until another conflict arises.

Given a word in the second Stroop box, System 1 automatically processed both the ink color and the word itself. Because reading is the dominant learned response, System 1 reflexively surfaced the word's meaning – prompting it to scream: "This is ***blue***!" Meanwhile, the less-practiced task of color naming conflicted with this impulse. System 2 had to intervene, suppress the automatic reading response, and redirect attention to the ink color – revealing what you were actually meant to say: ***orange***.

This led to a constant friction: While you were trying to complete the task quickly, System 2 was forced to apply the brakes – slowing your responses just enough for System 1 to process the conflicting signals without defaulting to the wrong one.

The important point to make here, is that System 2 is the supervisor of System 1, not its replacement. Effectively, System 2 helps us to navigate the real-world space, *to regulate our intuition*, to avoid mistakes, and not to repeat them again. It *guides intuitive response generation* by programming the context window – we call this "reasoning".

The Guided Intuition Machine In line with our earlier comparison, we ask: *Where is System 2 in LLMs?* We find no satisfying answer to this question within our conceptualization. However, considerable research has been invested to emulate the reasoning-like behavior of System 2 in LLMs. This suggests that either, our conceptual mapping of "System 1 & 2" thinking to LLMs is mistaken, or the idea that current LLMs are reasoning is flawed. This prompts further investigation – leading us to the next Section (2.1.2).

2.1.2 The Problem: Attempts to Model Reasoning in LLMs

The dual-process theory of thought (cf. Kahneman (2012)) has inspired a wave of work in LLM research aiming to emulate the reasoning-like behavior of *System 2* – the active supervisor that guides and corrects the intuitive reflexes generated by *System 1* (cf. Besta et al. (2024b), DeepSeek-AI et al. (2025), and Yao et al. (2023b)).

Guided by Kahneman (2012), we conceptualize System 1 as equivalent to the task of next-token prediction itself: A process driven by a contextual window of stimuli that primes the next likely response, based on learned heuristics. System 2 on the other hand, may be seen as the regulator of this context window. We introduced this notion in Section 2.1.1, and ask:

Where is System 2 in these models?

We highlight two parallel strands of development that emulate human reasoning with LLMs – both originating at the idea of a Chain-of-Thought (CoT; Wei et al. (2023), see Section 3.2.1.2):

1. **Reasoning LLMs** are trained on CoT examples, to further improve their ability to generate them. They remain standalone next-token predictors. At inference, they effectivity produce their own *external stimuli* (i.e. reasoning context) for guidance – simulating a System 2 that can plan dynamically.
2. **Reasoning Paradigms** impose rigid prompting structures that predefine graph-like paths of thought: A reasoning step must be generated, branched, revisited, and so on (outlined in Section 3.2.1). While these paradigms helped seed the development of reasoning LLMs (CoT being the foundational case) they limit planning flexibility by hardcoding the structure of reasoning in advance. Nonetheless, by externalizing planning and forcing models to focus on a single reasoning step at a time, these paradigms resemble a form of explicit, System 2 style control.

Are these emulations representative of "true" System 2 thinking?

For the former, we cannot say. It is well possible that the attention mechanism (Vaswani et al., 2023) in reasoning models already exhibits sufficient inhibitory control to render additional inhibition (i.e.

via filtering the context window) pointless. However, we remain inherently skeptical that this is the case, given that LLMs generally exhibit the so-called *lost in the middle* effect (Liu et al., 2023), wherein models increasingly fail to recall important stimuli located away from the edges of the context window as its length grows. In our theory, this may result in the accumulation of distraction in context, and therefore at some point, risk the outcome – a typical System 1 trait.[4]. However such investigation lies beyond our current resources. Regardless, we remain skeptical. For the purpose of this motivational section, we therefore adopt the view that reasoning LLMs are merely simulating a System 2, but lack its definitive trait of "active" monitoring and inhibition.[5] We will return to this intuition in our final discussion, in Section 7.1.2.

Our Focus: Reasoning Paradigms This shifts the latter, reasoning paradigms, into our primary focus. As these are incapable of dynamic planning, they cannot be considered "true" System 2 thinkers in the sense of Kahneman (2012), albeit arguably offering more direct control over context then reasoning LLMs. In this sense, we view them not as scalable solutions, but as useful conceptual lenses – controlled testbeds for reasoning behavior. As such, there is reason to hope that progress made in the setting of reasoning paradigms may again transfer back to more flexible models. This motivated our investigation – and led to a critical finding that challenges the foundations of the latest reasoning paradigm, Graph-of-Thought (GoT; Besta et al. (2024b), Section 3.2.1.4).
Despite their promise of well-regulated and yet structured graph-based reasoning, we find that models operating within GoT remain limited to accessing only *the immediate predecessors* of each thought (i.e. reasoning step). Distant or parallel branches, though present in the graph, are effectively ignored. As a result, insights from distant reasoning cannot be reused or integrated – contradicting the very premise of graph-structured deliberation as motivated by Besta et al. (2024b) themselves.

On a broader level, this failure to retrieve across chains – despite the graph's formal structure allowing it – connects back to our suspicion towards reasoning LLMs: The inability to selectively surface relevant context under cognitive load. We hypothesize that this limitation is not incidental, but fundamental. It may obstruct the emergence of "true" System 2 style control, which crucially depends on deliberate and selective retrieval from prior context.

[4] Additional evidence may already point to this idea: There is evidence that longer reasoning chains may be associative of degradable performance (Ballon et al., 2025) – an indication supporting our intuition, but also leaving room for the fact that longer traces may simply represent fruitless searches. Furthermore, reasoning LLMs often "overthink" by producing redundant reasoning steps that can harm performance (Sui et al., 2025), or exhibit error propagation (Gan et al., 2025) by failing to suppress faulty earlier outputs (cf. Ferrag et al. (2025)).

[5] And we are not alone to be skeptical; cf. Shojaee*† et al. (2025)

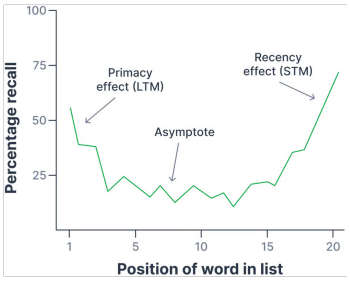
In response, we turn to the controlled setting of GoT as a conceptual testbed to investigate a solution. This brings us to the next Section, 2.1.3.

2.1.3 The Solution: Working Memory

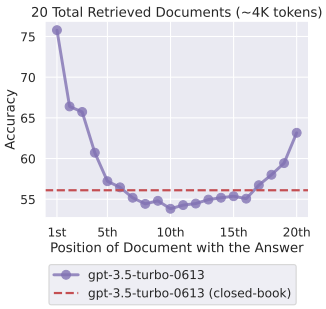
We introduced (in Section 2.1.2) how modern reasoning LLMs appear to face a shared bottleneck with reasoning paradigms, and System 1 style inference: An inability to maintain access to relevant information under cognitive load. Yet while the analogy of System 1 & 2 offers a useful framing, we find its abstraction too coarse to guide concrete improvements. Rather than remain in the realm of psychology, we turn to cognitive science for a more computationally grounded framework.

A striking empirical parallel motivates this shift: The *lost-in-the-middle* (Liu et al., 2023) phenomenon observed in LLMs mirrors the classic *primacy–recency* effect (Deese & Kaufman, 1957; Murdock, 1962) in humans. In both cases, recall accuracy follows a U-shaped curve, with early and recent items retained more reliably than those in the middle (as shown in Figure 1).

Figure 1
Two parallel U-shaped recall patterns in human and artificial cognition. ***Left:*** *Human recall performance exhibits a classic primacy–recency curve (Murdock, 1962).* ***Right:*** *Language models show analogous degradation for mid-context information (Liu et al., 2023).*



(a) *Primacy–recency effect in humans. Recall follows a U-curve, with higher accuracy for early and late items Murdock (1962) (LTM: Long term memory; STM: Short term memory, alias Working Memory)*
Graphic adapted from:
https://www.pipedrive.com/en/blog/quotes-for-email-signature.



(b) *Lost-in-the-middle effect in LLMs. Performance drops for information placed in the middle of the input context. Adapted from Liu et al. (2023).*

Cognitive science connects this U-shaped pattern to the properties of *working memory* (WM). The primacy effect reflects items consolidated from WM, while the recency effect captures what remains active in WM's *limited-capacity* store (cf. Tarnow (2016)).

This limitation – classically framed as the "magical number seven" (Miller, 1956) – is not a defect, but a feature consistently acknowledged across the landscape of cognitive science (Hitch et al., 2025). WM retains only a small, dynamic subset of context: A cognitive distillate that remains accessible under overload. WM is not passive. It features a control system that steers attentional focus – effectively embodying *System 2* by enabling deliberate, selective access to a constrained context – a mechanism that may precisely counteract the lost-in-the-middle effect in LLMs.

Our Focus: The MCM Over the past fifty years, theoretical frameworks of WM have been developed and refined – most dominantly, the Multi-Component Model (MCM; Section 3.3.1.2), which frames WM as a structured system with separable but interacting subsystems.
And yet, despite this rich and deeply studied design space, the MCM remains, to the best of our knowledge, *entirely* absent from LLM research. Existing work either leans on loosely defined psychological notions or, in the case of agentic frameworks that already invoke a "working memory" (Section 3.3.2.1), makes no reference to the apparently dominant model in cognitive science.

We regard this as a missed opportunity. Taken together with the reasoning limitations outlined in the prior Section 2.1.2, this reveals a dual gap – one we consolidate in the following Section 2.2.

2.2 Research Niche

We position this work at the intersection of a dual gap – one that spans both the limitations of reasoning in LLMs and the absence of theoretically grounded memory mechanisms to support it.

Gap I: LLM-Based Reasoning Without Dynamic Context Control As introduced in Section 2.1.2, we suspect that contemporary reasoning LLMs merely simulate structured reasoning due to lacking mechanisms actively regulating expanding context – limiting the emergence of true "System 2" reasoning. This motivates a turn to the formally expressive *Graph-of-Thought* paradigm (GoT; Besta et al. (2024b)), since it directly suffers from the identical limitation, expressed in reverse: Only immediate predecessors in the reasoning graph are attended to, with no mechanism to selectively integrate insights from distant or parallel branches. Thereby rendering the motivation articulated by Besta et al. (2024b) unfulfilled, GoT risks remaining a formalism lacking practical reasoning depth. This also limits its downstream applicability in self-improvement techniques, such as Self-Distillation (J. Huang et al., 2022), whose gains hinge on higher-quality reasoning as a training signal.[6].

Gap II: Absence of Working Memory as Theoretical Scaffold Reminiscent of working memory (WM) as a solution to this problem, we find, that the dominant cognitive model of WM – the Multi-Component Model (MCM; Section 3.3.1.2) – remains entirely absent from current LLM research. Agentic frameworks occasionally refer to "working memory" in name (Section 3.3.2.1), but do so without reference to the MCM, whose subsystem-oriented perspective has been developed and refined over decades in cognitive science. This disconnect, we argue, represents a missed opportunity: The MCM offers a rich, modular design space for managing a set of *limited and dynamically relevant context* – a control structure precisely suited to the bottleneck observed in GoT, and suspected in reasoning LLMs.

Our Research Niche We take GoT as a controlled testbed for reasoning, and position our research within the intersection of this dual gap, between: (1) the context limitations of GoT and (2) the

[6]Self-Distillation has not yet been introduced in this chapter. Our original investigation began there, but soon shifted toward improving the reasoning paradigms it implicitly relies on. We formally introduce Self-Distillation in Section 3.1.1, and show in our initial experiment (Section 5.2) that its effectiveness hinges on the quality of reasoning – thereby providing additional motivation for the presented main line of inquiry.

broader disconnection from the MCM. We investigate a dynamic WM mechanism for LLMs that is both: (1) Capable of distilling relevant material from an arbitrarily large reasoning graph to support the generation of new GoT-thoughts, and (2) rigorously grounded in the cognitive theory of the MCM.

2.3 Contributions

Building on the research niche outlined above, we pursue a memory-centric approach to LLM reasoning. Using Graph-of-Thought (GoT; Besta et al. (2024b)) as our application environment, we draw on the concept of working memory (WM) as defined by the Multi-Component Model (MCM; Baddeley (2000) and Hitch et al. (2025)). Our contributions are summarized as follows:

- We provide a software-engineering centric interpretation of the MCM, as foundation for architectural synthesis (Section 3.3.1.3).
- We synthesize a modular, abstract, and extensible core architecture for exploring LLM-targeted counterparts to the MCM (Section 4.3).
- We implement four experimental and task-agnostic WM variants of our architecture (Section 4.4).
- We apply these WM variants within the GoT framework to address its context limitation (Section 5.3).

A more detailed account of these contributions is provided in our Conclusion (Chapter 8).

2.4 Structural Overview

This work is structured as follows:

- **Chapter 3 – Background**:

  We outlines our research trajectory and presents the relevant theoretical foundations. Beginning with the question of how LLMs might improve themselves through Self-Distillation (SD; J. Huang et al. (2022)), we trace the evolution and limitations of reasoning paradigms, and identify our research niche in the context of working memory (WM) as a potential solution. This leads us further, to explore two strands of reasoning-related memory: The mechanisms used in current LLMs, and the structured models of cognitive theory – in particular, the Multi-Component Model (MCM; Baddeley (2000) and Hitch et al. (2025)).
- **Chapter 4 – Approach**:

  We presents our memory-centric solution to support reasoning in LLMs. We define core design goals, establish theoretical grounding in the MCM, and synthesize a modular core architecture for WM in LLMs including considerations on its extensibility. Four WM variants are introduced, designed for integration into the Graph-of-Thought (GoT; Besta et al. (2024b)) framework.
- **Chapter 5 – Experiments**:

  We present our experiments. First validating our exploratory hypothesis that SD benefits from improved reasoning paradigms, we follow in evaluating our WM variants within the GoT setting to address its context limitations.
- **Chapter 6 – Limitations**:

  We reflect on practical limitations encountered throughout architecture design, implementation, and evaluation, and discuss targeted next steps within our defined scope.
- **Chapter 7 – Discussion**:

  We reconnect to our suspicion that modern LLMs lack deliberate reasoning capabilities, and expand on broader conceptual implications. Thereby, we discuss future directions for working memory in LLMs, advocating for deeper cognitive grounding, and providing structural outlining of research frontiers beyond the scope of this thesis.
- **Chapter 8 – Conclusion**:

  We conclude by summarizing our work, reconnecting it to the research niche, and providing a concise account of our contributions and key findings.

Chapter 3

Background

In this chapter, we present the necessary background by outlining our research trajectory: We establish our niche, highlight important theories, related work, and inspiration that lead to our approach (Chapter 4).

Our inquiry originated from sheer curiosity on how large language models (LLMs) could learn to improve on their own. In Section 3.1, we introduce one answer to this question, Self-Distillation, and outline that a major bottleneck, in turn, lies in an idea used within – the ability to reason.

This brings us into Section 3.2, where we begin by illustrating the intricacies of prompting strategies aimed at modeling increasingly complex reasoning structures; which we simply refer to as "reasoning paradigms". Following, we identify a limitation in the latest paradigm: The inability to retrieve arbitrarily prior reasoning material flexibly, to learn from distant mistakes (Subsection 3.2.3).

Enter Section 3.3, where we narrow our interest further: We identify our niche within the intersection of mentioned limitation and established frameworks of working memory from cognitive science, and formulate our central research question.

To address our goal of designing a working memory for LLMs, we outline core principles and inspiration from established frameworks of working memory in cognitive science, and relevant LLM-related approaches alike (Subsection 3.3.1 and 3.3.2 respectively).

This invites a synthesis of both perspectives within the context of the reasoning paradigm, which we undertake in Chapter 4, where we lay the theoretical foundation for our approach.

3.1 Self-Distillation and the Reasoning Bottleneck

We begin by introducng the idea of Self-Distillation, a variant of self-improvement (Subsection 3.1.1). This brings us to the idea, that Self-Distillation might be fundamentally constrained by the strength of applied reasoning tactic(s) (Subsection 3.1.2). Finally, this leads us further to the next Section 3.2, in which we dwell deeper into such strategies and their limitations.

3.1.1 Self Distillation

The idea of enabling machines to improve themselves dates back to the earliest days of artificial intelligence, as famously exemplified by Samuel (1959).
Central to any such self-improvement method, including Self-Distillation, is the ability to acquire new learning signals, typically in the form of labeled data, which must then be meaningfully incorporated.
Consequently, we briefly introduce the broader context of data annotation methods (Subsection 3.1.1.1), then succinctly narrow our focus to provide a conceptual overview of the family of self-distillation methods (Subsection 3.1.1.2), which prepares the groundwork for a detailed introduction to the original Self-Distillation variant proposed by J. Huang et al. (2022) (Subsection 3.1.1.3)

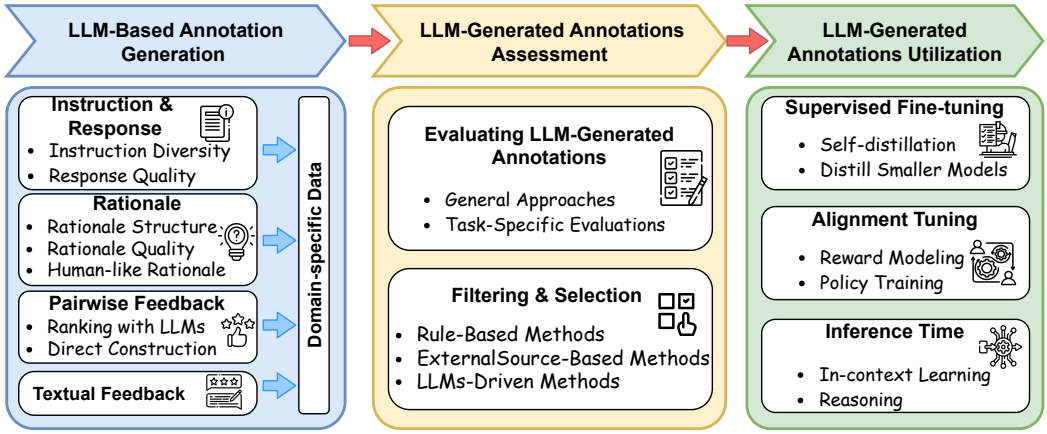
3.1.1.1 Data Annotation with Large Language Models

In machine learning, and specifically, in natural language processing (NLP), the process of data annotation involves several steps. Tan et al. (2024) propose a taxonomy on the existing research for data annotation with LLMS, with three core stages: (1) Annotation Generation, (2) Annotation Assessment, (3) Annotation Utilization (see Figure 2).

Not only are labels assigned to *existing* data (Stage 1), but, in order to improve the quality of labels, the synthesis of various forms of auxiliary information is required (Stage 2). As a result, the task remains complex, leading to a wide range of methods, that ultimately apply generated labels in various contexts (Stage 3).

One of such areas of utilization is supervised finetuning, where a model is continued to be trained with the newly annotated data. This leads us to the family of self-distillation methods.

Figure 2
A taxonomy of existing research on data annotation with LLMs. Adapted from Tan et al. (2024).



3.1.1.2 Conceptual Overview of Self-Distillation

Utilizing generated annotations for self-improvement, one paradigm is Self-Distillation. First introduced by J. Huang et al. (2022), several variants have emerged (Tan et al., 2024). These share a common conceptual foundation. [7]

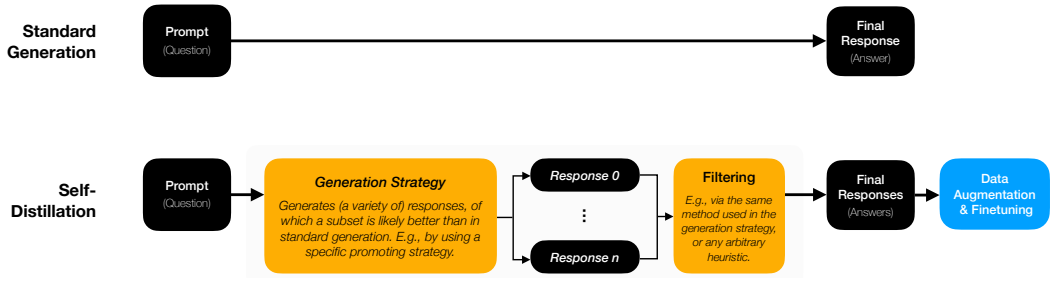
Unlike traditional distillation, where a teacher model provides training targets for a separate, typically smaller student model (thereby "distilling" its knowledge), *Self*-Distillation employs a single model in the teacher and the student role (Tan et al., 2024).

At first glance, however, the idea of improving a model by training it on its own predictions may seem counterintuitive. How can a model improve on labels it generated itself? Intuitively, one might expect such a process to reinforce existing biases, ultimately leading to degeneration – as if one were to confine a model to its own private echo chamber, where it amplifies existing beliefs without ever being challenged or corrected. And, rightfully so, the risk of model collapse is significant (in fact, for any data-annotation method; cf. Tan et al. (2024)).

Consequentially, one would be correct to presume that some additional training signal is required for the method to succeed (analogous to the second stage in Tan et al. (2024) taxonomy). For Self-Distillation, we can conceptualize this signal as the combination of (A) additional computational work, and (B) the use of a filtration heuristic:

[7] Please note that, while related reinforcement learning–based approaches are discussed in Tan et al. (2024), including methods such as Chen et al. (2024), we do not consider these variants here as they fall outside the scope of our research.

Figure 3
Conceptual Point-Of-View of considered Self-Distillation techniques.



Instead of generating a single response directly to a given question, we invest effort either, for example, in producing a wider array of responses or in dedicating more computation to refining individual responses (or both). Then, we employ a heuristic to filter out at least some of the weaker ones (see Figure 3).

This can introduce enough external structure to prevent the model from spiraling in its own mental feedback loop. Intuitively, what begins as an "echo chamber of unfiltered self-talk" can become something closer to a contemplative retreat: isolated, yes, but no longer directionless.

3.1.1.3 The Original Self-Distillation Method

For our work, we focus on the original variant of Self-Distillation, introduced by J. Huang et al. (2022), as we find it offers the most practical setup for further experimentation.

The generation strategy employed within their approach is known as *Self-Consistency* (X. Wang et al., 2023); an ensemble variant of *Chain-of-Thought* prompting (Wei et al., 2023), which we will introduce in more detail in the following Section 3.2.1.2. Self-Consistency embodies both, additional responses, and a filtration heuristic: Given a question, the model samples n answers, including the lines of reasoning leading up to it. A majority vote is then performed over the final answers, filtering out those that do not support the most frequent outcome. The idea behind this strategy, is that correct answers are more likely to appear repeatedly then incorrect ones. Or put differently: "the intuition [is] that a complex reasoning problem typically admits multiple different ways of thinking leading to its unique correct answer" (X. Wang et al., 2023). Finally, the reasoning paths resulting in the majority answer are collected, each forming a training sample that is further augmented into four variants, to prevent overfitting. The model is then trained on all augmented

(question, reasoning, answer) triplets. At test time, the majority voting procedure is repeated, to produce a final answer.

A crucial detail that J. Huang et al. (2022) show here, is the importance of including the obtained reasoning paths, and not just the final answer, in the training data (amounting to a different of 10.25 percentage points on average). Perhaps most remarkably, however, is how J. Huang et al. (2022) demonstrate that in doing so, the fine-tuned model alone, without using Self-Consistency at test time, can in some cases outperform the combined strategy that produced its training data (precisely: on 3/6 datasets using standard prompting; with comparable performance on 2 more, when using CoT instead). In other words, while the annotation strategy may identify coarse patterns that enable the filtration of the worst reasoning errors, the finetuning process appears to facilitate additional reasoning capabilities that the heuristic itself does not explicitly capture.

3.1.2 The Reasoning Bottleneck

Intriguingly, J. Huang et al. (2022) note that, beyond Self-Consistency, more advanced strategies "could readily [be] incorporate[d]" into their method to further refine and filter generated reasoning paths. They also provide a list of related works suitable for such integration, including Jung et al. (2022), Li et al. (2023), Ye and Durrett (2022), and Zelikman et al. (2022).

While diverse in their specific approaches, these methods can also be viewed through another conceptual lens: rather than treating textual paths of reasoning as isolated outputs to be filtered post hoc, we can see these techniques as operating within a reasoning thought graph; a structure in which chains of thought may interact.[8] Within such a graph, a model begins with a single line of reasoning but may generate additional branches, evaluate them, revisit earlier ideas, discard unpromising paths, or integrate competing lines of thought into a coherent whole.
Seen through this framework: Ye and Durrett (2022) propose a method for abandoning thoughts based on reliability assessments; Jung et al. (2022) explore recursive expansion of reasoning chains; Li et al. (2023), like X. Wang et al. (2023), focus on weighing different reasoning paths to guide answer selection.

[8] We exclude Zelikman et al. (2022) from this comparison, as their method assumes answer labels to be available during training.

This "thought-centric" perspective, naturally leads to the next Section, where we examine the structure and function of related reasoning paradigms in greater detail. At the core of our argument lies the assumption that the effectiveness of Self-Distillation is constrained by the reasoning strategy it employs.[9].

This brings us to the guiding question of the next Section: How might we increase the efficacy of such reasoning paradigms?

[9] We show this empirically, on our first larger experiment; see Section 5.2

3.2 Evolution and Limitations of Reasoning Paradigms

In the prior Section 3.1, we argue that the efficacy of reasoning paradigms for large language models (LLMs) can be seen as the primary bottleneck to advancing self-learning methods such as Self-Distillation.

In this Section, we begin by illustrating the evolution of such reasoning strategies (Subsection 3.2.1). Our analysis culminates in the identification of key limitations (Subsection 3.2.2) that lead us to narrow our research interest (Subsection 3.2.3).

We then continue to the next Section 3.3 to narrow our focus further, establish our research niche, and outline its exploration.

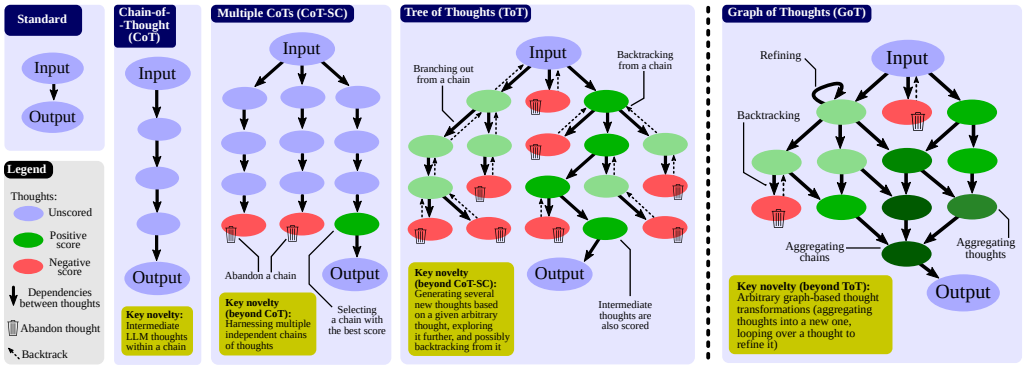
3.2.1 The Evolution of Reasoning Paradigms

In this Section we introduce the evolution relevant reasoning paradigms [10] for large language models (LLMs).

While reasoning from one “thought” to the next is an idea that may feel intuitive – in practice, the notion of a “thought” is highly task-dependent. One working definition describes a thought as “a coherent language sequence that serves as an intermediate step toward problem solving” (Yao et al., 2023b). More generally, a thought may represent “an initial, intermediate, or a final” solution, with its concrete form varying by use case – for instance, it might be a paragraph in a writing task or a sequence of numbers in a sorting task (Besta et al., 2024b). As a heuristic, a thought should be “small enough so that LLMs can generate promising and diverse samples (e.g., generating a whole book is usually too ‘big’ to be coherent), yet big enough so that LLMs can evaluate its prospect toward problem solving” (Yao et al., 2023b).
Despite these nuances, the assumption under which all here presented methods operate, is that thoughts are arranged in structured, often hierarchical, relationships – implicitly or explicitly forming a graph, containing "reasoning paths". Henceforth, the precise level of granularity at which a node in such a graph is defined, is flexible and context-dependent.

[10] To be precise, we refer here to prompting strategies designed to model increasingly complex reasoning structures. These strategies can be considered *static* with respect to their operational structure – a concept we anticipate in this footnote, Please see Subsection 3.2.2.1 for further details.

Figure 4
Evolution of prompting strategies designed to model increasingly complex reasoning structures (in short: "reasoning paradigms"). Note that the ensemble variant, Self-Consistency ("Multiple CoTs (CoT-SC)"), introduced in Subsection 3.1.1, with majority-voting, is not further illustrated here.
Adapted from Besta et al. (2024b).



This distinction becomes especially salient when contrasting the here illustrated techniques, in which the underlying graph structure is usually explicitly predefined and operated on, with more unconstrained methods that allow reasoning to unfold freely as natural language. For instance, contemporary LLMs that pre-generate longer reasoning text (e.g. DeepSeek-AI et al. (2025)), do so without an explicitly modeled graph. Yet fundamentally, whether or not the structure is made explicit, any coherent sequence of "thoughts" can be viewed as a walk through an underlying reasoning graph. Once expressed in text, this graph is necessarily linearized into a sequential form.

With this conceptual backdrop, we now turn to tracing the evolution of reasoning paradigms that progressively place the idea of an explicit thought structure at their center. We examine how each successive method seeks to overcome the limitations of its predecessor, and place greater emphasis on more recent developments, which lay the groundwork for the challenges and opportunities explored in this thesis. A high level overview is provided in Figure 4.

3.2.1.1 Standard Prompting and In-Context-Learning

We refer to the general practice of prompting a model without explicitly encouraging structured reasoning, as *standard prompting*. This category includes a variety of techniques aimed at improving output quality through contextual guidance alone, without requiring the model to engage in intermediate reasoning steps.

A particularly influential form of standard prompting, dating back to the early days of the LLM boom, is *In-Context Learning* (ICL[11] Brown et al. (2020)). Formally, ICL refers to the ability of a language model to perform a task more effectively when provided with a sequence of input-output examples directly within the prompt.[12] The key insight behind its effectiveness lies in the autoregressive nature of language models: Since a LLM predicts each next token based on its preceding context (see prelude Section 1.2), the likelihood of producing the desired output should naturally increase when similar patterns are already embedded in the prompt. Crucially, this process involves no parameter updates. Instead, the prompt appears to help the model activate relevant knowledge within its pretrained representations, helping to localize task-relevant behavior purely through contextual conditioning.

While the effectiveness of ICL tends to improve with the number of examples (or "shots"), it most notably scales with model size. Larger models exhibit stronger performance even without any task-specific finetuning (Brown et al., 2020). However, gains from scaling model size have been show to be least effective for tasks requiring logical and mathematical reasoning (Rae et al., 2022). This illustrates that, although ICL examples may incidentally include traces of reasoning, they are not inherently designed to elicit or structure a reasoning process. Their primary function is pragmatic: to help the model infer which task, out of all it has seen in training, should be performed. In that sense, ICL is best understood not as a reasoning strategy, but as a general-purpose mechanism for standard-prompting.

3.2.1.2 Chain-of-Thought

Addressing the aforementioned limitations of ICL on tasks that require structured reasoning (section 3.2.1.1), a notable advancement in eliciting such behavior comes from the work of Wei et al. (2023), who introduce the idea of *Chain-of-Thought (CoT)* prompting: Rather than prompting a language model to directly generate an answer (even when examples are provided) CoT introduces natural language descriptions of intermediate reasoning steps into the ICL examples. Each example is thus reformulated into a triplet of the form: *(input, reasoning trace, answer)*.[13] The term *Chain-of-*

[11] Though historically, this technique was initially referred to as *few-shot learning*.

[12] For example: `"Paris is the capital of France. Berlin is the capital of Germany. Madrid is the capital of"`, prompting the model to complete with `"Spain"`.

[13] For example, in standard ICL, a prompt might include: `"Q: What is 12 + 8 + 3? A: 23. Q: What is 5 + 9 + 1? A: 15. Q: What is 8 + 2 + 6? A:"`. In the CoT variant, each example includes an intermediate reasoning step: `"Q: What is 12 + 8 + 3? Let's think step by step. 12 + 8 is 20, and 20 + 3 is 23. A: 23."`

Thought refers to this intermediate reasoning sequence. Empirically, the authors show that CoT has improved performance significantly, particularly in large-scale models.

CoT is particularly suited for questions whose answers are not trivial to infer in a single step. Conceptually, one might frame CoT as a form of divide-and-conquer: The model is not only shown what the final solution looks like, but also how to decompose the problem, what kind of sub-steps are required, and how to combine them into a final conclusion.
In doing so, the reasoning trace scaffolds the final answer through a step-by-step progression. Each intermediate step is typically simpler than the full task, making it easier for the model to complete the chain incrementally. For instance, when solving a multi-step arithmetic question, the intermediate steps often follow predictable patterns (once seen), even if the overall problem appears complex. Once these intermediate results are derived, the final answer becomes a relatively straightforward consequence of the reasoning context; again, aligning well with the autoregressive nature of LLMs.

In short, CoT can be understood as the first major step toward integrating explicit reasoning processes with LLMs, moving beyond sole task recognition. However, it also exposes a new limitation: the lack of control over the reasoning process itself. What if we always fail at a specific step? What if the problem calls for exploring multiple lines of reasoning, not just one? This leads us to the next variant: Tree-of-Thought.

3.2.1.3 Tree-of-Thought

In the previous Section 3.2.1.2, our comparison between standard prompting (where a model is expected to generate an answer in a single step) and Chain-of-Thought (CoT) prompting (where a model can reason linearly before generating the final answer) is linked to a well-known dual-process theory of cognition, which distinguishes between a fast, automatic, heuristic-based "System 1" and a slower, more reflective "System 2" mode of thinking (Sloman, 1996). While System 1 relies on surface-level intuition, System 2 actively *plans, evaluates, and reformulates* a given problem [14].

This already speaks to the core limitation of CoT: Its linearity and irreversibility. Though the problem of some reasoning traces being more effective than others may be (partially) solvable via ensembling (see Self-Consistency, Section 3.1.1), once a reasoning path is generated, CoT prompting

[14] A well-known illustration is the bat-and-ball problem: *A bat and a ball cost $1.10 in total. The bat costs $1 more than the ball. Quick! How much does the ball cost?* A System 1 response yields $0.10; System 2 is needed to recognize the correct answer is $0.05.

offers no mechanism to explore alternatives, revise earlier steps, or recover from local errors. To overcome this, Yao et al. (2023b) propose *Tree-of-Thought (ToT)* prompting. Their method poses a superior generalization of CoT and Self-Consistency, that frames reasoning as a structured search over multiple interconnected solution paths.

In ToT, each *thought* is defined as a semantically coherent chunk of language that serves as an intermediate reasoning step. A reasoning trajectory is represented as a sequence of such thoughts, and the overall problem-solving process becomes a traversal over a tree of thought-states $s = [x, z_1, \ldots, z_i]$, where x is the task input and z_j is the j-th thought generated "so far" [15]. Such a tree can then be explored using search algorithms such as breadth-first or depth-first search (BFS/DFS), enabling the model to backtrack to previous thoughts and continue reasoning in a different direction.

To support this, a new capability is required: The ability to evaluate partial reasoning progress and *decide* whether to continue or abandon a current path. In ToT, this is implemented via (thresholding over) a heuristic that assigns a numerical score to each state, estimating how close it is to solving the task. Ultimately, the final answer is selected from the highest-scoring state, while the decision to terminate the search depends on task-specific criteria (such as reaching a verifiable solution, exhausting a step budget, or observing stagnating state score).
While, as the authors note, such heuristics are traditionally either learned or predefined, a key development within ToT is to let the language model itself assume this role; thus incorporating *reflection* (elaborated in Section 3.3.2.2) directly into the reasoning process.
The authors further explore both *independent* and *comparative* evaluation strategies, where candidate paths are scored either in isolation or against one another. Similarly, new thoughts may be generated either through independent sampling, or, by explicitly prompting the model to generate a set of alternatives together. Yao et al. (2023b) report that the former performs better “when the thought space is rich (e.g., each thought is a paragraph), and i.i.d. samples lead to diversity,” whereas the latter is favored “when the thought space is more constrained (e.g., each thought is just a word or a line)”.

Despite offering a substantial improvement over CoT, ToT still inherits important constraints:
First, for each new task, a reasoning tree must be designed manually. Specifically, one has to define

[15] Note that the authors do not explicitly specify whether “thoughts so far” refers to the path from the root to the current node or to the global temporal sequence of all explored thoughts. However, both the search algorithm and provided code (Yao et al., 2023a) strongly indicate the former. This distinction will become crucial in a following Section 3.2.2.2

what a partial solution should look like at each level of the tree, and how the model should generate and evaluate continuations.[16] In other words, the "reasoning-plan" in ToT is static.
Second, and more fundamentally, each reasoning path remains isolated from its neighbors. But what if a promising idea from one path could inform the development of another? What if the goal is not just to select among competing thoughts, but to integrate them? These questions ultimate led to the final variant we introduce in the following: Graph-Of-Thought.

3.2.1.4 Graph-of-Thought

While ToT (Section 3.2.1.3) allows models to explore multiple reasoning paths in parallel, it still imposes a strict tree-like structure on the process. Since integration across branches is not supported, each reasoning branch remains isolated. Besta et al. (2024b) propose *Graph-of-Thought (GoT)* to address this limitation. Their method expands the ToT framework, by facilitating an arbitrary thought graph, in which thoughts can also have multiple incoming edges. This formulation allows for the expression of arbitrary graph topologies: a thought may be refined (via self-loops), expanded into multiple continuations (multiple successors; as in ToT), or aggregated from several sources (multiple incoming edges).

In GoT, thoughts are represented as nodes in a directed graph $G = (V, E)$, where an edge $(t_1, t_2) \in E$ indicates that thought t_2 was generated using t_1 as part of its input context. To advance the reasoning process, GoT introduces *thought transformations* (from hereon also simply referred to as "operations", following the terminology used in the GoT implementation (Besta et al., 2024a). Each such operation is formalized as a function of the form $\mathcal{T} : G \rightarrow G'$, which takes the current thought graph as input and produces an updated graph by generating new thoughts, or abandoning (akin to removing) others. Such operations include for example "Generate", "Score", "Improve", "Validate", "Aggregate", "KeepBestN.[17]. Finally, this leads to the abstraction of a so-called *Graph-Of-Operations (GoO)*, that defines dependencies between *instance* of such operations. [18] In practice, an instance of a GoO subsumes a tree of operation instances, where edges in the tree determine

[16] As an example, consider the creative writing task from the original ToT paper, where the model is prompted to write a short passage ending in four target sentences. The ToT setup fixes the tree to depth 2: the model first generates multiple candidate writing plans, votes on the best one, then generates candidate passages based on that plan and votes again. This entire structure – including the two-step plan-then-write sequence, the number of candidates per step, and the evaluation prompt – must be manually defined in advance.

[17] Notably, GoT also expands the incorporation of Self-Reflection, as a standalone operation. We turn to Self-Reflection in Section 3.3.2.2. Further note that, for clarity, we omit the additional parameter p_0 used by the authors to represent the language model that (can) facilitate an operation.

[18] Consider this pseudo-example: `Generate(branch=0) -> Generate(branch=4) -> Score() -> KeepBestN(n=2) -> Aggregate()`, which would first generate an initial thought, then 4 parallel continuations, score each, keep only the best 2 thoughts, and aggregate them into one.

which operation must be executed before another. The GoO is static and task-specific, and makes the "reasoning plan", that we previously articulated for ToT, explicit.

In summary, we extend the original formulation of the GoT reasoning process as a tuple that includes the planning component (the GoO) explicitly:

$$(G, \mathcal{T}, \mathcal{E}, \mathcal{R}, GoO)$$

where:

- $G = (V, E)$ is the current graph of thoughts, with vertices V and edges $E \subseteq V \times V$,
- $\mathcal{E} : (v, G) \to \mathbb{R}$ is a scoring function that assesses the quality or utility of a thought v in the context of G,
- $\mathcal{R} : (G, k) \to \{v_1, \ldots, v_k\}$ is a ranking function that selects the top k thoughts for further use,
- $GoO = (O, D, S)$ is a directed acyclic graph [19] over operation instances, where:
  - $O = \{o_1, o_2, \ldots, o_n\}$ is a finite set of schedulable operation instances,
  - $D \subseteq O \times O$ defines execution dependencies: $(o_i, o_j) \in D$ implies that o_i must be executed before o_j,
  - $S \subseteq O$ tracks the set of operations that have already been executed.
- $\mathcal{T}(O, D, S) = \{o_j \in O \mid \forall (o_i, o_j) \in D : \ o_i \in S\}$ is the function that returns the frontier of operations whose predecessors have been executed. Execution proceeds by repeatedly executing all operations in $\mathcal{T}$ until no further operations can be scheduled.

Empirically, GoT outperforms ToT across a range of tasks that profit from a divide and conquer approach. For instance, in sorting tasks, GoT achieves a 62% reduction in error compared to ToT, while also lowering cost by more than 31%. These improvements are attributed not to scale, but to GoT's ability to actually facilitate divide and conquer in the reasoning graph, by decomposing tasks into smaller, tractable subtasks, solve them independently, and then re-integrate them.

[19] Ideally, the GoO is always a flow network with exactly one sink, if one wants to leverage the aggregation feature of GoT so that at the end, all thoughts that have not been abandoned can contribute to the final solution.

In doing so, GoT provides a strong foundation for prompting strategies that mirror the networked, recursive nature of human thought. However, as we will see in the next Section 3.2.2, though the theoretical foundation of GoT is rich, its limitation lies not only in the inherited inability of dynamic planning (the staticity of the GoO), but also in an ambiguity between their practical implementation and what is claimed in theory.

3.2.2 The Limitations of Reasoning Paradigms

Following the previous Section 3.2.1 in the introduction of prompting strategies designed to model increasingly complex reasoning structures ("reasoning paradigms"; from no reasoning with ICL, to evermore increasing complexity of reasoning from CoT to ToT to GoT), we illustrated how each successive variants sought to alleviate a limitation of the prior.
In this Section, we want to highlight key limitations observed in the latest variant, GoT, that motivate our initial research niche; the exploration of which, leads us to Section 3.3, where we investigate existing memory architectures for reasoning processes.

3.2.2.1 The Role of Staticity

A fundamental limitation shared by Chain-of-Thought (CoT), Tree-of-Thought (ToT), and Graph-of-Though (GoT) is their reliance on statically defined reasoning structures, that have to be adapted to a given task. These increase in complexity, and are thus harder to automate, the more advanced the method becomes. For a new task: in CoT, one has to manually craft examples [20]; in ToT, one has to set up a task specific tree template; in GoT, instantiate a concrete Graph-of-Operations (GoO), where once defined, dependencies between operations cannot be adjusted during inference.

Still, this limitation can play a constructive role, as static reasoning frameworks may provide a controlled and interpretable environment for investigating reasoning strategies. In that sense they can be seen as proof-of-concepts, rather then as holistic solutions to reasoning.
Indeed, such have already catalyzed the development of more flexible approaches. CoT, despite, or precisely because of its simplicity, has informed the design of modern reasoning models that internalize many of the behaviors originally embedded in such fixed templates, by flexibly generating a more complex chain of thought (see prelude, Section 1.3).

[20] This is arguably easy to automate with modern reasoning models; arguably a direct result of recognizing this limitation.

This points to an interesting research direction: Moving beyond manually defined reasoning templates toward models that can flexibly construct and revise their own structure during inference – not only over chains, but possibly over trees and graphs alike [21]

3.2.2.2 The Locality Constraint

Following the progression from in-context learning (ICL) to Chain-of-Thought (CoT) prompting, to Tree-of-Thought (ToT), and finally to Graph-of-Thought (GoT), each successive framework has introduced mechanisms to advance reasoning limitations of its predecessor (see Section 3.2.1). GoT, in particular, promises a generalization beyond tree-based search by allowing arbitrary graph topologies (Section 3.2.1.4).

However, a closer analysis reveals a critical gap between GoT's formal expressive capacity and its practical implementation: As previously introduced, formally, a thought transformation (aka "operation") in GoT is permitted to operate over the *entire thought graph* G, making it theoretically capable of arbitrarily complex reasoning, including distant and parallel thoughts as input.[22] Yet in practice, GoT imposes a much narrower constraint, where each thought is actually conditioned solely on its immediate parent thoughts (Besta et al., 2024a) [23] , meaning, that operations are limited to local, direct inputs. More precisely, in practice, an instance of an operation simply receives a set of thoughts from its immediate predecessor operations in the GoO, and in turn decides which thoughts to make available to its successors; wether that is via generating new ones or simply filtering the input set. This results in a functional behavior closer to $\mathcal{T} : T_{\text{in}} \rightarrow T_{\text{out}}$, where T_{in} is the union of all output thoughts provided by predecessor operations, and T_{out} is the set of output thoughts. The full graph G – while present as a conceptual abstraction – is not modeled explicitly in the system. [24] Notably, this represents not an advancement over, but rather a regression from the predecessor (ToT) implementation [25].

[21] As discussed at the beginning of this chapter, any structured reasoning process – including those defined over trees or graphs – must ultimately be serialized into a linear thought sequence. While explicitly modeling such structures may offer benefits, particularly for guiding or constraining reasoning behavior, we do not explore this modeling direction further and make no assumptions about its necessity for effective model training, e.g. in connection to reasoning LLMs.

[22] To recall the (simplified) definition by Besta et al. (2024b): Each such operation is a function of the form $\mathcal{T} : G \rightarrow G'$, which takes the current thought graph as input and produces an updated graph by generating new thoughts, or abandoning (akin to removing) others. See Section 3.2.1.4.

[23] As becomes apparent, when inspecting how any operation gathers its input thoughts via the `get_previous_thoughts` function, which "Iterates over all [direct] predecessors [operations] and aggregates their thoughts".

[24] Constructing the thought graph explicitly required extending the original framework. While conceptually straightforward, the process proved labor-intensive, and made clear that the original implementation had not been designed to treat G as a first-class structure.

[25] ToT not only formally permits each thought to be generated in the context of the entire reasoning trajectory from the root to the current node, but also enforces this behavior in its implementation (Yao et al., 2023a), by explicitly concatenating

This practical limitation is consequential, as it undermines a central motivation articulated by the GoT authors themselves. In their introduction, they argue that graph-structured reasoning better reflects human cognition, where one should be able to "explore a certain chain of reasoning, backtrack and start a new one, then realize that a certain idea from the previous chain could be combined with the currently explored one" (Besta et al., 2024b). Such scenarios require the ability to refer to distant, parallel, or even long-abandoned thoughts, and not just direct predecessors.

While one could argue that this divergence between implementation and theory is limited fundamentally by the staticity of the Graph-of-Operations (GoO; see Subsection 3.2.2.1), we want to stress the point that addressing this limitation is separate from dynamically growing such GoO. Specifically, even if the order of operations remains fixed, the system should still be able to ask: *"Can we identify a better set of input thoughts by considering the entire reasoning history, rather than being limited to the immediate outputs of our predecessor operation?"*. However, there is no such mechanism for referencing or retrieving more distant thoughts elsewhere in the graph, even if they are topologically reachable. Nor do Besta et al. (2024b) offer any further guidance on how to encode or traverse the full graph efficiently.

In other words, while GoT's formalism gestures toward a holistic reasoning paradigm, its design falls short of addressing a critical question: How should one determine which parts of an ever-growing thought graph are relevant when generating a new thought? For instance, what if we have already made a specific mistake and recognized this by assigning a low score? Shouldn't the system ensure that this avenue of reasoning is not revisited, and that such an error is not repeated in subsequent steps?

each new thought to the accumulated chain of prior thoughts (see `get_samples` and `get_proposals`).In contrast, GoT ignored this aspect, leavening it up for the task implementation to manually encode payloads into individual thoughts, without standardizing any such behavior.

3.2.3 The Gap in Reasoning Paradigms

In this Subsection, we consolidate the previously discussed limitations to narrow down the scope of our research interest.

To reiterate, while Graph-of-Thought (GoT) expands the structural expressivity of prior methods, we identify two fundamental shortcomings:

1. It retains a rigid planning paradigm (the Graph of Operations remains static; Section 3.2.2.1).
2. It fails to operationalize its own theoretical flexibility, lacking a principled mechanism for identifying or retrieving relevant reasoning material beyond immediately preceding thoughts (Section 3.2.2.2).

Given the complexity of the problem space, we choose to focus on the retrieval constraint. We offer three reasons to support our preference:

1. **Retrieval Precedes Planning**: We suppose that dynamic planning over a reasoning graph presupposes access to the right context. That is, in the context of GoT, dynamically growing the GoO (choosing which type of operation to instantiate, where, with which hyperparameters) would also depend on the optimal set of input thoughts per operation type – but not the other way around: Even if an operation is already chosen, we may still identify a better set of input thoughts by considering the entire reasoning history.
2. **Retrieval is Non-Trivial**: Simply keeping all prior thoughts in the context-window does not scale. As the reasoning graph grows, retrieval failures would become more pronounced. Prior work has shown that LLMs suffer from primacy–recency bias in long contexts, often neglecting middle-range information (Liu et al., 2023). A growing reasoning context full of distractions and local errors would hence impair their ability to reuse intermediate reasoning steps and revise prior assumptions (cf. also Ferrag et al. (2025)). Attempting to attend globally to all prior thoughts is therefore both computationally expensive and cognitively implausible.
3. **Contemporary Models Already Internalize Planning Flexibility**: As introduced in Section 3.2.2.1, inspired by static frameworks like CoT, recent LLM-based reasoning models have started to generating evermore complex reasoning (see prelude, Section 1.3), and thus bypass rigid planning schemes entirely. A focus on retrieval may complement this trend by enabling

such models to better condition their generation on selectively reused content (also, in regards to the previous point).

Taken together, these points motivate a shift in focus: To further the original motivation from Besta et al. (2024b), dynamic access across the entirety of past reasoning is required. In particular, we see a need for a system that can flexibly identify and reuse relevant reasoning insights, selectively incorporating them into new thought generations without getting distracted.[26]

While one could imagine various solutions to this problem, such as including task-specific pipelines or increasingly complex heuristics, our intuition suggests that the need for selective access to an ever-growing body of reasoning material naturally points toward a memory system [27].
This led us to investigate existing work on memory mechanisms for reasoning. An exploration we undertake in the following Section 3.3.

[26] In hindsight, after having researched the space on working memory, we emphasize that with this formulation, we expand the problem space given by GoT slightly: We see not only the need for dynamic retrieval of prior thoughts, but also the need for active *curation* of the retrieved material, so it may become maximally useful for the thought to be generated – akin to filtering our distracting erroneous steps and only preserving the resulting insights.

[27] Specifically, the distinction between an arbitrary retrieval function and a dedicated memory mechanism appears to blur as the information space expands. However, we do not claim this is the only possible path forward.

3.3 Perspectives on Memory for Reasoning in Cognitive Science and LLMs

In this Subsection, we continue to narrow our focus, by defining our research niche and explore it conceptually to motivate ideas for our approach (Chapter 4).[28]

To reiterate: In the initial Subsection 3.1.1, we argued that the effectiveness of self-improvement techniques (specifically Self-Distillation) is constrained by the employed reasoning paradigm; here, specifically, prompting strategies designed to model increasingly complex reasoning structures. We traced the evolution of such strategies in Subsection 3.2.1. This prompted us to identify a key limitation of a recent approach, Graph-of-Thought (GoT): The lack of a principled mechanism for selectively incorporating relevant material from an ever-growing reasoning graph (Section 3.2.3).

While, as mentioned, one could imagine various solutions to this limitation, our intuition suggests that the need for selective access to an arbitrarily large reasoning body naturally points toward memory as a promising direction. Inspired by the conceptual link that GoT's predecessor (Tree-of-Thought) draws to the dual-system theory of cognition (see Section 3.2.1.3), we were reminded us of a foundational analogy from cognitive science: Miller's classic notion of "the magical number seven" (Miller, 1956). It suggests that human *working memory* maintains a compact, dynamic set of active elements, typically 4–7 chunks of information, that guide deliberate reasoning. In light of the uncovered limitation, we followed this analogy, to consider whether a similar form of working memory might serve as a control mechanism in LLM based reasoning.

Yet, as we surveyed the literature, we were surprised to find no work on LLMs referencing the established models of working memory from cognitive science. That is, while – as we will continue to see – there are anecdotal references to general principles from cognitive science, to the best of our knowledge, no comprehensive working memory framework from the field has thus far left a trace in LLM-related research.

This absence helped crystallize our research niche: We aim to explore how insights from human working memory might inspire a new class of memory mechanisms for reasoning in LLMs. To structure this exploration, we formulate our **research question** as follows:

[28] Please note that we do not conduct a comprehensive literature review, but instead refer selectively to works that inform our perspective on the limitations discussed in the prior Subsection 3.2.3.

How might a working memory mechanism for LLMs be designed to selectively obtain relevant material from an arbitrarily large reasoning graph, when generating a new thought? [29]

To approach this, we decompose our search for inspiration within this subsection into two parts:

1. What are the core principles of established human working memory models in **cognitive science**? (Section 3.3.1)
2. Which existing **LLM-related approaches** might best support memory-augmented reasoning in LLMs? (Section 3.3.2)

This, in turn, invites a synthesis of both perspectives, in the context of GoT, with which we lay the theoretical foundation for our approach, in Chapter 4.

3.3.1 Cognitive Science: Models of Working Memory

In this subsection, we outline the core principles of the human working memory (WM) models as established in contemporary cognitive science. Our primary focus is on the Multi-Component Model (MCM; Section 3.3.1.2), which we adopt as our main reference due to its modular structure and widespread recognition (Miyake & Shah, 1999).

We begin by briefly outlining the landscape of working memory models in cognitive science (Section 3.3.1.1), then highlight the core principles shared across models (Section 3.3.1.1), and proceed to a detailed description of the components of the Multi-Component Model (MCM; Section 3.3.1.2). Acknowledging the inherent ambiguity in the field, we conclude with our own interpretation and technical formalization (Section 3.3.1.3), which serves as a foundation for a compact list of key principles that we attribute to the MCM (Section 3.3.1.4).

[29] Please note that we explicit relax our question, by considering more then a retrieval function for previous thoughts, and thinking about any kind of reasoning material. At the same time, we reassert that, in the context of GoT (Subsection 3.2.1.4) our question does not regard changes to the underlying "operation plan". Alltough, as we will see, this is of course a possible future improvement to consider.

Please note that the peripheral concepts presented in relation to the MCM are not intended as a comprehensive literature review. They reflect a selective and necessarily limited perspective, as might be expected from a computer science background.

3.3.1.1 The Core Principles Across the Landscape

Despite structural differences, WM models converge on several shared principles (Hitch et al., 2025):

- WM has limited storage capacity.
- It combines processing and storage functions, temporarily holding and actively manipulating recent information to support goal-directed behavior.
- Attention (i.e. conscious awareness) is essential for maintaining and manipulating (complex) information.
- Information not actively attended decays rapidly.
- WM interacts with long-term memory (LTM). Though the nature of this interaction remains unclear, it "may well influence performance at every stage" (Baddeley, 2010).

These are further underpinned by several general concepts:

Chunking The concept of a "chunk" refers to a coherent unit of representation in WM that can encapsulate multiple individual elements under a single representation, and can be abstract in nature (Baddeley, 2010; Miller, 1956). Importantly, while chunks are believed to leverage hierarchical structures, their exact nature remains not fully understood (Burgess & Hitch, 2005).
Similarly, the process of “chunk*ing*” can be seen as forming such groups or hierarchies; we interpret this as analogous to “indexing” multiple elements into a higher-level abstraction. Additionally, some theories suggest chunking depends on integrating prior knowledge from long-term memory (LTM) to organize information meaningfully (Burgess & Hitch, 2005; Hitch et al., 2025).[30]
Henceforth, practicing chunking techniques can improve the transfer and retrieval of information, but importantly, it does not increase storage capacity (Ericsson & Kintsch, 1995). Additionally, it is suspected that new kinds of abstractions may form precisely when sufficient contextual variations of individual items have been encountered (Burgess & Hitch, 2005).

[30] For example, meaningful sentences allow to "compress" larger spans of words: Subjects generally fail to recall a series of more then 5-6 words when they are unrelated, compared to 16+ words if they form meaningful sentences (Hitch et al., 2025).

Binding Binding is the process by which separate features (e.g., the color, shape, and location of an item) are integrated into coherent, retrievable representations (Roskies, 1999) (e.g., "red + bird + tree = a red bird in a tree").

Binding is distinct from chunking: While binding combines features within a single item, chunking "compresses" or "indexes" multiple items into higher-order representations (Hitch et al., 2025).

Capacity Constraints A core characteristic of WM is its limited storage capacity. Miller's classic formulation originally suggested that a person can hold up to seven chunks of information in WM (Miller, 1956). While the original estimate assumed a fixed chunk size, later research by Cowan (2001) showed that chunks can take up variable capacity, depending on their category (e.g. numbers take up more capacity then words) and features (e.g. longer words take up more capacity then shorter ones). The precise nature of the capacity limit in working memory remains an open question, as no single hypothesis accounts for all empirically observed phenomena (Baddeley, 2010; Ma et al., 2014; Oberauer & Kliegl, 2006). Several hypotheses offering partial explanations. We briefly summarize these approaches here, drawing on popular sources [31] where we found a useful high-level categorization scheme:

- *Decay Theories*: Assume that information fades over time unless actively rehearsed. Since rehearsal time is limited, so is the number of items that can be maintained (Baddeley, 2000; Barrouillet et al., 2004). This is the central theory embedded in the MCM.
- *Resource Theories*: View WM capacity as a shared resource distributed across items, where recall precision declines as more items are stored (Ma et al., 2014).
- *Interference Theories*: Propose that stored items directly interfere with each other. Notably, items that are more similar interfere with each other more strongly (Baddeley, 2000; Maehara & Saito, 2007). Several sub-variants can be distinguished:
  - *Retrieval Competition*: Items stored close together may share retrieval cues and compete for recall (e.g., recalling the second item of a list instead of the first; Oberauer et al. (2012)).

[31] See *Working Memory – Capacity*, Wikipedia, https://en.wikipedia.org/wiki/Working_memory#Capacity, accessed June 2, 2025.

  - *Superposition*: Interference arises from *additive noise*, where multiple items blend into a shared, noisy trace (e.g., recalling one word may unintentionally bring in elements from others; Oberauer et al. (2012)).
  - *Feature Overwriting*: Interference involves *subtractive loss*, where similar items compete for shared features (e.g., when storing two similar faces in memory, like those of siblings, shared features such as eye shape or hair color may interfere, making it harder to recall which features belong to which person; Oberauer and Kliegl (2006)).

3.3.1.2 The Multi-Component Model of Working Memory

In this Subsection, we present the *Multi-Component Model* (MCM; Baddeley (2000, 2010), Baddeley and Hitch (1974), and Hitch et al. (2025)), which appears as the most widely cited (Miyake & Shah, 1999) framework for conceptualizing working memory (WM) in cognitive science.

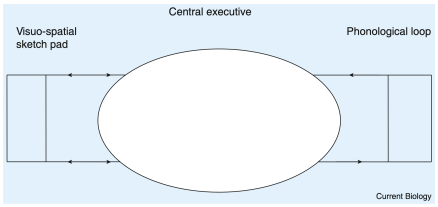
As illustrated in Section 3.3.1.1, holistic frameworks like the MCM are conceptual tools rather than precise formalizations; they deliberately leave room for interpretation, making ambiguity inevitable. To account for this, we aim to outline the authors' perspectives as faithfully as possible, structuring this subsection by increasing degrees of ambiguity: After giving a short overview, we begin by detailing the components of the MCM individually (Section 3.3.1.2), proceed to discuss suspected interactions between these components and other subsystems – where most ambiguity arises (Section 3.3.1.2); concluding with our own interpretation and formalization (Section 3.3.1.3).

Overview The MCM has evolved over 50 years, with successive additions introduced to account for increasingly diverse experimental findings (see Figure 5). In its latest form (Hitch et al., 2025) the MCM conceptualizes WM as a modular system of temporary storage buffers, each handling specific types of information, optionally coordinated by a stateless process manager known as the *Central Executive*. It distinguishes between two classes of buffers:

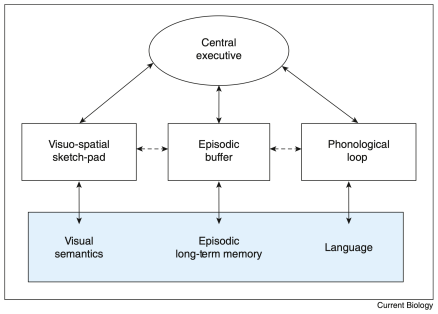
- **Slave Buffers**: Specialized stores for domain-specific information, that stand in direct contact to the perceptual-motor system. There are two such buffers: the
  - **Phonological Loop** (for auditory information), and the
  - **Visuospatial Sketchpad** (for visual and spacial information).

- **Episodic Buffer**: A *passive* multimodal store that integrates information across modalities. It "is episodic in the sense that it holds episodes whereby information is integrated across space and potentially extended across time", and can be thought of the memory interface to the central executive; encapsulating all information accessible to "conscious awareness", that are "assumed to be retrievable consciously" (Baddeley, 2000).

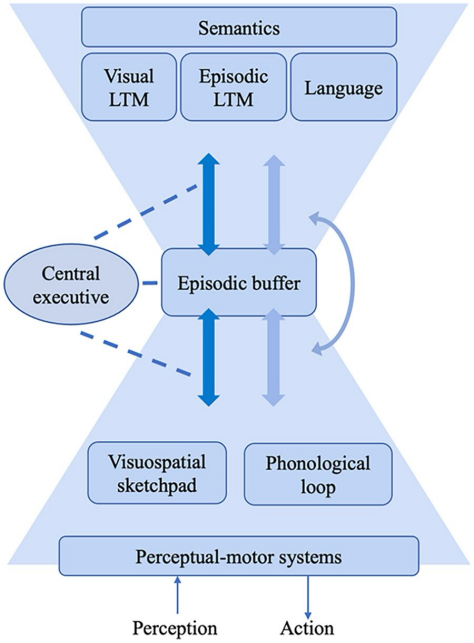
Figure 5
Evolution of the Multi-Component Model (MCM) of Working Memory.



(a) *MCM (1974). Adapted from Baddeley and Hitch (1974): "The earlier concept of the short-term memory has been elaborated to include an attentional controller and two modality based temporary stores. The components are assumed to interact, and to be linked to both perception and long-term memory."*



(b) *MCM (2000). Adapted from (Baddeley, 2000): "Includes links to long-term memory and a fourth component, the episodic buffer that is accessible to conscious awareness."*



(c) *MCM (2025). Adapted from (Hitch et al., 2025): "A multicomponent view of the cognitive system with working memory located between semantic long-term memory and perceptuo-motor subsystems with the episodic buffer as the central hub. The central executive is shown as a separate resource that can be deployed in a variety of ways to interact with many parts of the system. Light and dark arrows indicate implicit and explicit processes, respectively."*

Individual Components of the MCM

We continue by outlining all components of the Multi-Component Model (MCM) in isolation, as is feasible: The Central Executive, Phonological Loop, Visuospatial Sketchpad, and Episodic Buffer.

The Central Executive The latest version of the MCM (Hitch et al., 2025) distinguishes between "implicit" and "explicit" processes. Implicit processes operate automatically, while explicit pro-

cesses require conscious awareness (akin to the previously mentioned “System 1” and “System 2” in dual-process theory (Sloman, 1996), respectively). To execute explicit processes, the system relies on deliberate attentional control. This is the role of the *Central Executive*, which Hitch et al. (2025) liken to “top-level management” in an organization:

> "So far, we have found a potentially useful analogy [for the central executive] with real-world systems for the management of complex, multifaceted organizations. In such systems, **routine operations are delegated** to free up central management capacity but, importantly, they **can be elevated to central management** when circumstances require, such as an unexpected event [...] but [this] has finite resources for central management operations. [...] A further key feature in our current thinking is the close link [32] between the central executive and the focus of attention in the episodic buffer. In terms of the present analogy, we would see this [(the episodic buffer)] as **corresponding to a display of the current situation** with which central management is concerned."

The Central Executive is a stateless, capacity-limited resource for *controlling and allocating attention to cognitive processes* (Hitch et al., 2025). It has no memory of its own and does not manipulate information directly; rather, it monitors the progress of ongoing operations (like a manager overseeing a dashboard) and intervenes when necessary to reallocate resources. According to Baddeley (2010) and Hitch et al. (2025), this includes:

- refreshing representations in the episodic buffer ("administering the management dashboard"),
- retrieving information from long-term memory ("requesting a management report"),
- and integrating multimodal information (binding and chunking; "restructuring items in the dashboard").

The Central Executive aligns with the *Supervisory Attentional System (SAS)* described by Norman and Shallice (1986), ensuring goal-directed behavior when automatic processes are insufficient. Its effectiveness is limited by:

- attentional capacity ("the size of the management team"),
- current memory load ("how bloated the dashboard is"),

[32] Which made it impossible for us to isolate the central executive completely from other components. Note that all described interactions stem directly from the authors.

- and the availability of suitable strategies ("the experience of the management").

Critics have argued that the Central Executive resembles a “homunculus” – a placeholder for an undefined process that simply shifts the problem of control to a higher level (Logie, 2016). In response, Hitch et al. (2025) acknowledge the challenge but argue that retiring the concept in favor of a “set of distributed processors” would be premature. They state that:

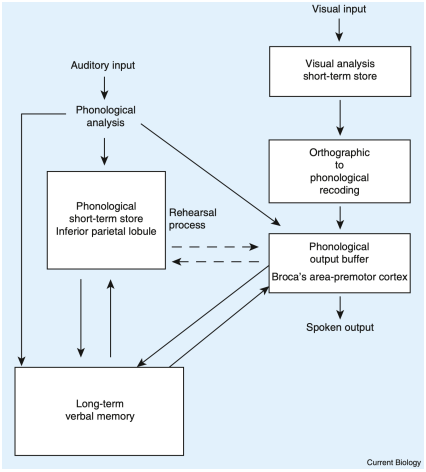
> “[...] it is far from clear how such a system would fulfill the job specification of the central executive. We see this as embracing aspects of perception [(e.g., alerting, focusing, zooming attention)], cognition [(e.g., planning, switching, refreshing, monitoring)], emotion [(e.g., activating, inhibiting)], and action [control (e.g., setting goals, initiating, coordinating, inhibiting)].” noting that "few of them have yet been adequately researched, and we do not claim that they are definitive".

The Phonological Loop The *Phonological Loop* (Baddeley, 2010; Baddeley & Hitch, 1974; Hitch et al., 2025) is a slave buffer that is dedicated for storing phonological information for short-term manipulation and is essential for tasks such as rehearsing word sequences. According to Baddeley (2010), it represents the only slave buffer suited for recalling sequences, and consists of further subsystems (see figure 6). Most notably:

- The **Phonological Short-Term Store** (aka "the inner ear"): A *passive* store, which holds incoming auditory information in temporal order but is subject to rapid decay (within approximately two seconds) unless refreshed.
- The **Phonological Output Buffer** (aka "the inner voice"): An *active* articulatory rehearsal component, which can refresh memory traces by sub-vocally repeating them in real time, "verbalizing" them for rehearsal or speech.

The Visuospatial Sketchpad The *Visuospatial Sketchpad* (Baddeley, 2010; Baddeley & Hitch, 1974; Hitch et al., 2025) is a slave buffer dedicated for visual and spatial information for short-term manipulation. According to Hitch et al. (2025), the visuospatial sketchpad has the following properties:

Figure 6
"A more detailed formulation of the phonological loop model based on both behavioral and neuropsychological evidence" Adapted from Baddeley (2010), citing Vallar (2006)



- Information decays within approximately 1.5 seconds unless actively maintained.
- It is primarily object-based, but individual features (e.g., color or shape) may be lost as complexity increases.
- It is well-suited for representing a **single** complex pattern, but it is **not** designed for serial recall.
- Rehearsal within the sketchpad does not substantially draw on the resources of the central executive (see also Baddeley (2000)).

Furthermore, Hitch et al. (2025) subdivide the sketchpad into a visual and spatial component:

- An **Active Inner Scribe** (aka "spatial short-term memory" or rehearsal system), which tracks spatial locations and movement information ("*where* an object is").
- A **Passive Visual Cache** (aka "object short-term memory"), which stores static features such as color and form ("*what* an object is").

However, Hitch et al. (2025) also note that it is “not clear why the visual and spatial components of working memory are grouped together” and suggest the possibility of additional separable components for motor, tactile, and kinaesthetic information.

The Episodic Buffer The *Episodic Buffer*, first introduced in the revised MCM by Baddeley (2000), serves as a passive, limited-capacity store that integrates information across modalities,

time, and space. It is supposed to provides a "bridge between rapid streams of perceptual or motor data" and long-term memory (LTM; Baddeley (2000)). The buffer holds what is sometimes referred to as the "current focus of attention" – to reiterate the management analogy introduced with the Central Executive, the buffer corresponds to the "management dashboard" – i.e. the state of information accessible to conscious awareness and available for monitoring by the central executive. Contrary to some parts of slave-buffers, the episodic buffer is **completely passive** (Hitch et al., 2025).

According to Baddeley (2000) and Hitch et al. (2025), the episodic buffer:

- stores representations in a "multimodal code" that integrates visual, verbal, and spatial features,
- is assumed to have a limited capacity of approximately four chunks or episodes,
- is episodic in nature, binding information across space and (potentially) time,
- is influenced by both perceptual inputs and internal control processes (e.g., goals, plans), and
- is subject to attentional operations such as refreshing and prioritization.

One line of evidence motivating the buffer's inclusion comes from experiments involving dual-task interference. For example, participants were asked to recall a sequence of visually presented digits while simultaneously repeating an irrelevant word aloud. According to the original MCM (Baddeley & Hitch, 1974), which included only two slave buffers – one for phonological information (the phonological loop) and one for visuospatial information (the visuospatial sketchpad) – this task should have been nearly impossible: The phonological loop would be occupied by the repeated word, and the visuospatial sketchpad, being unsuited for serial recall, would be inadequate for storing the digit sequence. Yet, participants performed surprisingly well. This suggested the presence of an additional system capable of integrating visual and phonological information. This integrative function is now attributed to the episodic buffer (Baddeley, 2000; Hitch et al., 2025).

Interactions Between Components and Other Systems

Having introduced the individual components of the Multi-Component Model (MCM) in Section 3.3.1.2, we now examine their interrelations and connections to other subsystems. This analysis necessarily involves interpretation, which we explicitly note where applicable.

Interactions between Components of the MCM Reviewing the iterations of the MCM (Baddeley, 2010; Baddeley & Hitch, 1974; Hitch et al., 2025), we identify two primary channels of interaction between its components.[33] While these are explicitly reported by the authors, we include some interpretation in the first:

1. **Interactions between the Central Executive and the Episodic Buffer**: According to Hitch et al. (2025), the episodic buffer is to the central executive what a dashboard is to the central management of a large organization. Since the central executive is also considered to be stateless, the episodic buffer is said to represent its memory interface. However, the central executive is also reported to not manipulate information directly. We interpret this subtle discrepancy as follows: The central executive can delegate which operations to execute on the representations in the episodic buffer; analogous to how top management typically delegates actions to middle management rather than intervening directly on the factory floor.
2. **Interactions between Buffers**: As introduced, the MCM includes two slave buffers (the phonological loop and the visuospatial sketchpad) and the episodic buffer. Importantly, the episodic buffer is considered to be completely passive while the slave buffers can transfer information from and to the episodic buffer, provided that the involved processes are simple enough to be handled implicitly (Hitch et al., 2025). Specifically, as long as integration processes (e.g., binding or chunking) do not require the central executive's intervention, the slave buffers can autonomously deposit information into, or retrieve it from, the episodic buffer.

Relation to Long-Term Memory and other Models The relationship between WM and long-term memory (LTM) remains an open and debated question (Baddeley, 2010; Baddeley & Hitch, 1974; Burgess & Hitch, 2005). In the MCM, WM and LTM are generally viewed as functionally separable systems: LTM is not a central separate component in WM but interleaved with it. As Baddeley (2010) cautions, "Long-term memory may well influence performance at every stage".

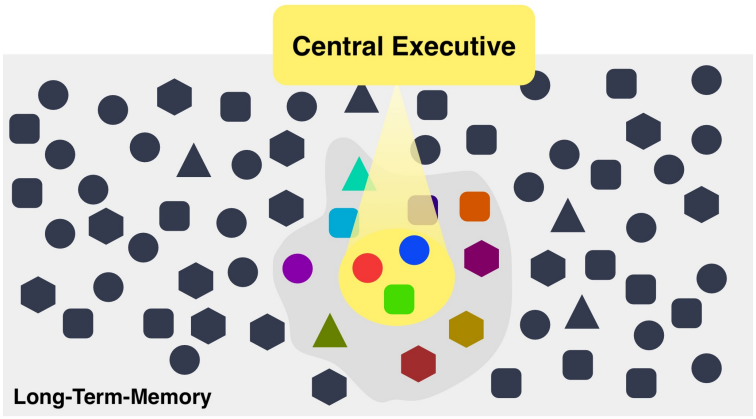
Conversely, the Embedded Process Model (Cowan, 1999) (an alternative to the MCM), conceptualizes the entirety of WM as a subset of LTM; Specifically, it posits a two-tiered system: the first tier consists of "activated" long-term representations, and the second tier is a narrower "focus

[33] Note that this excludes interactions with long-term memory, which we discuss in the following.

of attention" within that activated set (see Figure 7). However, this model appears to be seen as emphasizing different aspects of the WM–LTM interface and has been critiqued for difficulties in accounting for empirical findings (Hitch et al., 2025).

Figure 7

Schematic representation of the embedded process model (Cowan, 1999). The background represents long-term memory (LTM), consisting of numerous inactive representations. The lighter region highlights the subset of "activated" long-term representations (Tier 1), while the bright yellow spotlight indicates the "focus of attention" (Tier 2) within this activated set. The Central Executive directs attention within this subset, selectively focusing on relevant representations for further processing.



On the other hand, computational models of short-term memory (STM; i.e. also WM) and LTM, such as those (reviewed) by Burgess and Hitch (2005, 2006), often conceptualize memory items as being indexed by a distribution over associations to other items – in STM *and* in LTM alike – thereby suggesting that both systems interact through evolving contextual representations. They also hypothesize that chunking may play a key role in the transition from temporary to long-term representations. Similarly, Hebb (1961) (as cited in Burgess and Hitch (2005)) suggested that even a single trial in STM must induce a change in LTM.

Our own (admittedly computer-science-centric) interpretation is that interactions between long-term memory (LTM) and working memory (WM) may be distinguished into two categories, providing background knowledge in the form of learned representations that constitute either:

- items to be processed, or
- processes themselves.

To understand how such processes might be structured and deployed within the MCM, we now turn to the concept of *action schemata*, as introduced by Norman and Shallice (1986) and cited by Hitch et al. (2025) as a key inspiration for the distinction between “implicit” and “explicit” processes in the MCM. This concept has also become central to our own interpretation.

Action Schemata and Contention Scheduling Parallel to the development of the MCM within the MCM Norman and Shallice (1986) introduced the notion of *action schemata*. These are pre-learned cognitive structures that guide the execution of familiar, goal-directed action sequences. Each schema:

- is hierarchically structured, representing a sequence of sub-actions (e.g., the schema for "driving a car" might include "braking", "steering", "checking mirrors", each of which may further decompose into finer-grained steps),
- is associated with an *activation value* that reflects its current relevance (e.g., the activation value of the "braking" schema increases when the car in front is approaching rapidly),
- has an *activation threshold* that must be surpassed for the schema to become active (e.g., "braking" activates only when the distance to the car in front becomes critically small), and
- can represent both external actions (e.g., physical movements) and internal cognitive processes (e.g., mental problem-solving or binding operations).

The activation of schemata is typically automatic, governed by a process known as *contention scheduling* (Norman & Shallice, 1986). The process works as follows:

1. Sensory input delivers "trigger conditions". These are features or cues in the environment that are queried against a so-called "trigger database", to retrieve a mapping of candidate *root schemata* to activation values corresponding to the environment.
2. These candidates compete based on their activation values. If one candidate’s activation value surpasses the others by a significant margin and also exceeds its activation threshold, it is selected for execution and becomes "active".
3. Once a root schema is activated, it automatically activates its child schemata in a cascading fashion, forming a dynamic "horizontal thread" of execution (comparable to a chain of dependent operations in Graph-of-Thought, see Section 3.2.1.4).

4. Schemata continue operating until they either complete their task, their activation value drops below the threshold, or they are blocked by a lack of resources.
5. After execution, a schema's activation threshold may be updated to reflect context, learning, or task dynamics.

In this framework, routine tasks (or what Hitch et al. (2025) call "implicit processes") proceed without conscious oversight. The system relies on automatic competition, where schemata that best match the current situation are selected automatically by the contention scheduler. Crucially, routine selection is *not* controlled by attention. Attention (akin to "System 2" in the dual-process theory (Sloman, 1996)) only becomes necessary when automatic processes fail. According to Norman and Shallice (1986), such failures occur when:

- no schema matches the current situation,
- multiple schemata exceed their activation thresholds simultaneously (i.e., a conflict), or
- a novel situation requires deliberate, non-automatic reasoning.

In these cases, a higher-order control mechanisms, what Norman and Shallice (1986) call the *Supervisory Attentional System (SAS)* and Baddeley and Hitch (1974) attribute to the Central Executive, intervene. The key principle is that, this intervention does not override the contention scheduler directly. Instead, it can only *bias* schema activation indirectly, by modulating thresholds or activation values to resolve conflicts and enable goal-directed behavior.

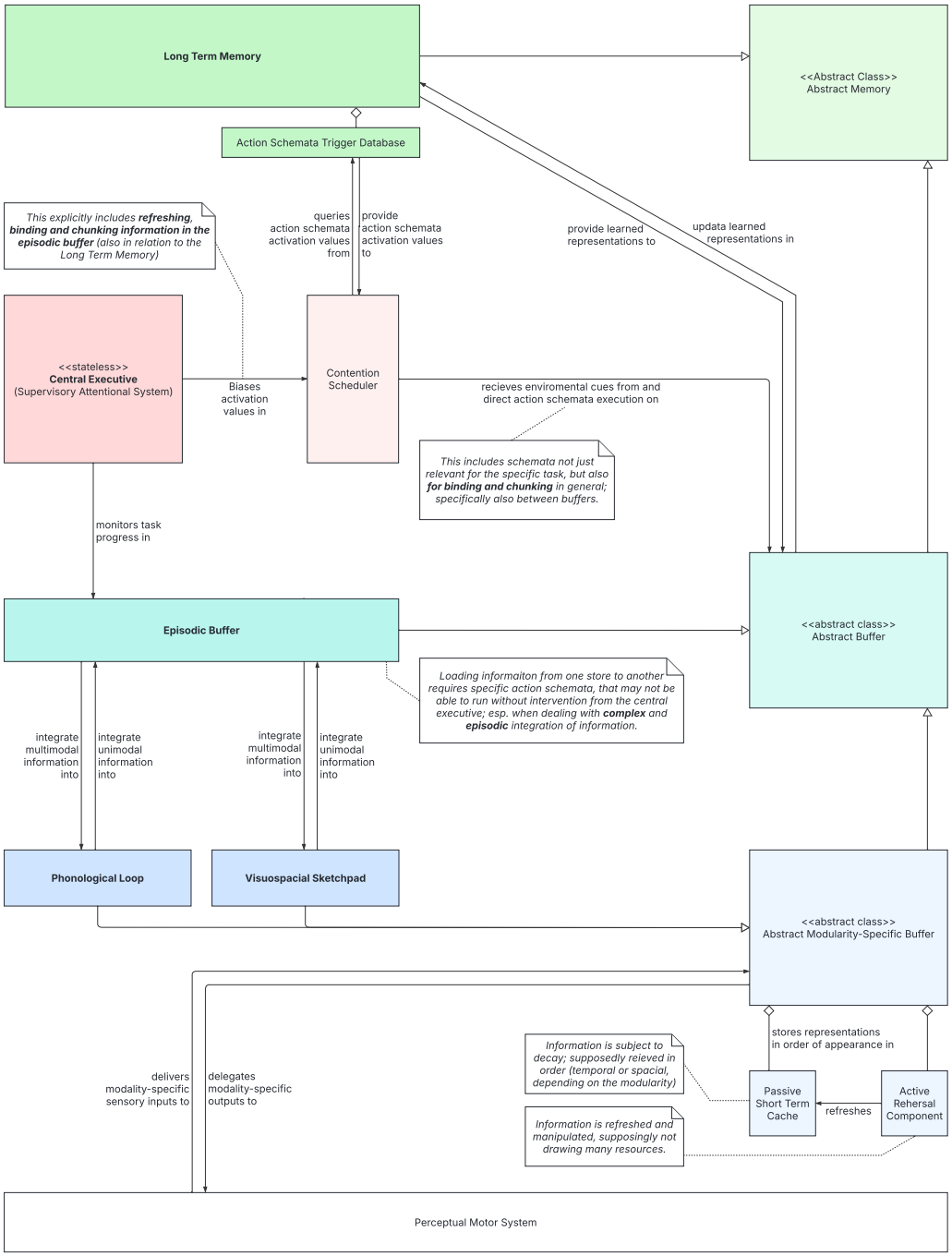
3.3.1.3 Our Interpretation of the Multi-Component Model

We now consolidate our analysis into a computer-science-centric interpretation of the Multi-Component Model (MCM) of working memory. This draws on general concepts in the research of working memory in cognitive science (Section 3.3.1.1), the individual properties of the MCM components (Section 3.3.1.2), and their (ambiguous) relation to other (sub-)systems individual components (Section 3.3.1.2). Another key element in our interpretation is the concept of *action schemata* (Norman & Shallice, 1986) (cf. Hitch et al. (2025), Section 3.3.1.2), which we view as a kind of *transaction* executable on any buffer (or potentially beyond, though this is outside our current modeling scope).

We illustrate Figure 8, which captures our interpretation as a class diagram.

Figure 8

Our interpretation of the Multi-Component Model (MCM) of working memory, expressed as a conceptual class diagram. Implementation details and structural constraints are intentionally abstracted.



The **Contention Scheduler**, schedules action schemata – transaction templates responsible for indexing and compressing (chunking), encoding (binding), or refreshing representations within buffers, or transferring them between buffers, as hypothesized by Hitch et al. (2025) and Norman and Shallice (1986). They are scheduled based on their current relevance, in relation to the current environment (Norman & Shallice, 1986). To our knowledge, it remains unclear how exactly these environmental cues are propagated to the scheduler. We speculate these to be represented in the current state of *all* buffers. Therefore, we see the contention scheduler as a kind of mediator able to access not just the episodic buffer, but also the slave buffers, in order to schedule action schemata (transactions) on, and receive environmental cues from them.

The **Central Executive** enacts "conscious oversight" when no suitable schema can be scheduled automatically. It monitors the state of the episodic buffer, supervises task progress, and reallocates attentional resources by manipulating activation values and thresholds (compare with Hitch et al. (2025) and Norman and Shallice (1986)). This process corresponds to “controlling and allocating attention to cognitive processes” (Hitch et al., 2025), akin to “System 2 thinking” in dual-process theory (Sloman, 1996). In this sense, we follow Hitch et al. (2025) understanding of the central executive as a stateless process manager that “decides what to concentrate on” (Hitch et al., 2025).

A **Buffer Hierarchy** is assumed, where

- **Abstract Memory** is a common abstract base class for all buffers and the long-term memory (LTM; motivated by Burgess and Hitch (2005); see Section 3.3.1.2)
- **Abstract Buffer** is a common abstract base class for all buffers, that internalizes an assumption hardwired to the MCM: that items within buffers are naturally subject to rapid decay unless refreshed (Baddeley, 2000; Hitch et al., 2025).
- **Abstract Modularity-Specialized Buffer** is our interpretation of a common set of features shared by all slave buffers; i.e. the phonological loop and the visuospatial sketchpad. These buffers receive information directly from the perceptual-motor system (e.g., auditory or visual cortices) and can process information in parallel (Baddeley, 2010). For information to flow into the episodic buffer, it must first be encoded (i.e., bound) into a multimodal format (Baddeley, 2000, 2010). This can occur automatically, without explicit attentional control, when integration is simple (e.g., combining visual features like color and shape). In such cases, slave buffers can deposit information directly into the episodic buffer (Baddeley, 2000),

which itself remains passive (Hitch et al., 2025). Additionally, this base class is further modularized into a passive cache and an active rehearsal component (Baddeley, 2000, 2010).

We see the following **Limitations** in our model:

- We decided not to include any memory-item (be it action schemata, chunks, or other representations) explicitly in the diagram, as it remains unclear to us wether action schemata would be stored in any kind of memory or just the LTM. If the latter were true, this would complicate the assumption that LTM and buffers share a common base class, while discerning this further (while adhering to the Liskov Substitution Principle) would introduce additional complexity we want to abstract.
- Consequently, we intentionally omit internal schemata properties, such as activation thresholds or decay rates, to avoid overcommitment to speculative details. However, we consider the eventual specification of a function mapping environmental cues to activation values as reasonable.
- As mentioned, another uncertainty arises from the question how environmental cues would propagate from the perceptual-motor system to the action schemata trigger database or the contention scheduler. Our speculation is that this pathway might depend on information already present in any buffer.
- We did not specify the phonological-loop's ability to recall, or the visuospatial sketchpads inability to do so. This is simply something we abstracted away.
- We also deliberately refrain from specifying multiplicities, particularly between the central executive, contention scheduler, and the buffers. This is primarily due to the uncertainty expressed by Hitch et al. (2025) towards how many more sub-memories or processors there may be.
- Finally, from a technical standpoint, this class diagram is purely conceptual. It includes theoretical constructs without defined operations or attributes, and does not imply conformance to any specific interface.

3.3.1.4 Summary: Core Principles for the Multi Component Model

This subsection has outlined the foundational principles of established human working memory (WM) models in cognitive science, with a focus on the Multi-Component Model (MCM). We summarize the following key insights in accordance with our interpretation of the MCM (Section 3.3.1.3):

1. Model-Agnostic Properties of Working Memory (Section 3.3.1.1)

- WM has **limited storage capacity**.
- It combines **storage and processing** functions, temporarily holding and actively manipulating recent information to support goal-directed behavior.
- **Attention** (i.e., conscious awareness; deliberate planning or reasoning) is essential for maintaining and manipulating increasingly complex information.
- Information not actively attended **decays rapidly**.
- WM **interacts with long-term memory** (LTM); the nature of this interaction remains unclear, though it *"may well influence performance at every stage"*(Baddeley, 2010).

2. Chunking and Capacity Constraints (Section 3.3.1.1)

- Chunks are coherent units of information handled in working memory. They can be considered as *hierarchical* indexing structures.
- Chunks vary in capacity demand depending on type (e.g., numbers vs. words) and complexity (e.g., word length).
- The precise nature of capacity limits remains an open question. Views include:
  - *Decay-based theories*: Forgetting occurs over time unless refreshed.
  - *Resource-based theories*: WM is a shared, limited resource.
  - *Interference-based theories*: Items disrupt or overwrite each other.

3. Architectural Components (Section 3.3.1.2)

- The **Central Executive** is a *stateless process manager* that monitors task progress, reallocates attention, and can be seen as "reasoning engine".
- A **Contention Scheduler** *automatically* selects routine operations (action schemata, see Box 4 below) based on relevance and environmental cues.
- WM consists of multiple storage buffers that can be categorized as:
  - **Slave Buffers**: modality-specific, partially active (e.g., phonological, & visuospatial).
  - **Episodic Buffer**: fully passive, multimodal, integrates data across slave buffers and serves as interface to the Central Executive.

4. Cognitive Processes (Section 3.3.1.2)

- **Action schemata** are **hierarchical** routines akin to transactions.
- Their relevance is a function of environmental cues, which in turn governs their relevance and drives competition for execution.
- While schemata are scheduled automatically by the Contention Scheduler, the Central Executive may influence this process indirectly by adjusting activation values or thresholds, rather than scheduling schemata directly.
- Repetition of representations is thought to reinforce their long-term counterparts, consistent with Hebbian learning principles (e.g., “cells that fire together wire together”). There is reason to suspect that “there must be some change in LTM [even] from a single trial [in WM]” (Burgess & Hitch, 2005).
- General representations may be categorized into hierarchical *transaction templates* (schemata), and the *items* they manipulate, which are structured hierarchically as *chunks*.

5. Component Interactions (Section 3.3.1.2)

- **Central Executive → Episodic Buffer**: monitoring progress.
- **Slave Buffers ↔ Episodic Buffer**: integration can occur implicitly, but must be initiated by the active components of the slave buffers, as the Episodic Buffer is fully passive.
- **Contention Scheduler → Buffers**: selects and executes schemata (transactions) based on environmental cues encoded in buffer states.
- **Central Executive → Contention Scheduler**: biases schema selection by adjusting activation thresholds or priorities, rather than selecting schemata directly.
- **Contention Scheduler ← Long-Term Memory**: retrieves learned action schemata and trigger mappings that guide schema selection.
- **Working Memory ↔ Long-Term Memory**: interaction is pervasive, yet poorly understood; LTM may influence all stages.
- **Slave Buffers ↔ Perceptual–Motor System**: Slave buffers receive modality-specific sensory input and delegate corresponding outputs.

3.3.2 LLMs: Agents and the Ability to Reflect

Guided by the principles identified from cognitive science (in the previous Subsection 3.3.1.4), in this subsection, we explore existing approaches in natural language processing (NLP) that may inform the design of a human-like working memory (WM) for large language models (LLMs). Our goal is not to provide a comprehensive literature review, but to draw inspiration. We selectively highlight paradigms that appear particularly relevant to two foundational questions:

1. Which LLM-related memory architectures for reasoning already exist and how are they structured?
2. What kind of representations could be stored in working memory for LLMs, and how may these enable a targeted abstraction from textual thoughts of an LLM?

As we will see in the first part of this subsection (3.3.2.1), we can address the first question by examining recent work on memory modules within LLM-based agents. These overlap with a common narrative that offered a well-researched direction for answering our second question: The idea of textual reflections (Section 3.3.2.2).

Together, these lines of work provide important conceptual foundations for a subsequent synthesis with models from cognitive science, which we undertake when introducing our approach in Chapter 4.

3.3.2.1 Memory Architectures for Reasoning: Agents

Since our goal is to support on-the-fly reasoning over an arbitrarily growing body of thought material, we are primarily interested in architectures that can selectively retrieve (and potentially distill) prior reasoning material into a focused context. This places our investigation within the broader family of so-called *Retrieval-Augmented Generation* (RAG) methods.

Retrieval-Augmented Generation (RAG) In such architectures, external memory typically stores segmented text passages (e.g., document chunks) and is queried to enrich the model's input prior to generation (Y. Gao et al., 2024). Contemporary RAG systems generally implement this memory using vector databases. These databases rely on comparatively lightweight models to *embed* (that is, to *map*) a given text to a high dimensional vector representation. The more similar two texts are,

the "closer" [34] their corresponding vectors will be in the embedding space. This, in turn, enables efficient retrieval of contextually relevant content: A given query text is embedded into the same space, and its k-best results correspond to its k-nearest neighbors in that space.

Starting with RAG, surveying the literature yielded a diverse set of memory architectures, ranging from task-specific approaches like Retrieval-Augmented Thought (RAT; Z. Wang et al. (2024)) [35] to more advanced frameworks such as GraphRAG (Deng et al., 2024). However, one direction overlapping with RAG distinctly caught our attention and appeared to better match our requirements.

LLM-Based Agents

> "Typically, an LLM-based agent refers to an interactive system that processes environmental observations, maintains context across multiple rounds of dialogue, and outputs executable actions tailored to completing a given task. Memory is one of the critical components of LLM-based agents, involving how agents store and utilize past experiences" (Hu et al., 2024).

Notably, memory in this context is sometimes explicitly referred to as “working memory” (Hu et al., 2024) – and indeed, we immediately recognized the reappearance of some principles previously identified in cognitive models of WM (see Section 3.3.1.4).[36] We highlight two approaches that appeared especially relevant to our goal of building a general-purpose, task-agnostic memory system that supports reasoning across long horizons and domains: Hi-Agent (Hu et al., 2024) and A-Mem (W. Xu et al., 2025).

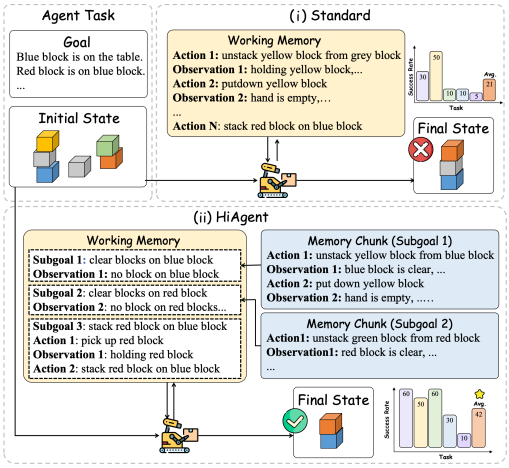
[34] With regard to cosine similarity, see prelude Section 1.1

[35] Retrieval-Augmented Thought (RAT) extends Chain-of-Thought prompting by allowing the model to retrieve additional documents after each reasoning step. It is designed for multi-hop question answering, where answers must be constructed by combining information from multiple documents. While this aligns with our goal of retrieval-based reasoning, it remains tightly coupled to the task of multi-hop QA and is thus not readily applicable to more general, open-ended reasoning settings.

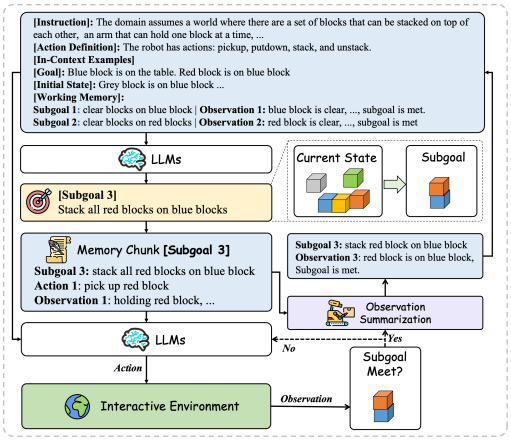
[36] Albeit the fact that we encountered no reference to established frameworks of working memory from cognitive science.

Hi-Agent Hu et al. (2024) introduce a hierarchical memory mechanism tailored for long-horizon agent tasks:

Figure 9
Overview of Hi-Agent's hierarchical memory strategy. Figures adapted from Hu et al. (2024).



(a) *The memory chunking strategy in Hi-Agent (Hu et al., 2024): During the pursuit of a specific sub-goal, the system retains only concise summaries of (action, observation) pairs from previously completed sub-goals, instead of maintaining the full historical trace.*
As seen in subfigure (ii), the current expanded sub-goal is "stack red block on blue block", while earlier sub-goals have been condensed into single summarized pairs. Conversely, subfigure (i) illustrates the standard approach, where the entire interaction history is fully expanded in memory.



(b) *"An overview of the process of HI-AGENT" (Hu et al., 2024): (1) the agent decomposes a high-level goal into sub-goals; (2) it iteratively executes actions and stores (action, observation) pairs in working memory; (3) upon completion, each sub-goal is summarized into a compact representation and replaces the detailed trajectory.*

As Hu et al. (2024) explain, in typical LLM-based agents, memory is organized into sequences of *(action, observation)* pairs: Given an initial state and a specific goal (e.g. "You are in a tidy kitchen"; "Prepare breakfast") the agent carries out an action (e.g. Action 1: "Fill kettle with water"), makes an observation (e.g. Observation 1: "There is water in the kettle, ready to be boiled."), and then appends the (action, observation) pair to its memory.

In the "standard" approach, as Hu et al. (2024) call it, the entire memory sequence is simply kept in the context window of the LLM generating the next action or observation. This is reported to scale poorly and lead to performance degradation on long-horizon settings due to contextual overload. Hu et al. (2024) improve this by decomposing the overall trajectory of (action, observation) pairs into sub-trajectories, each associated with a specific sub-goal. Instead of retaining the full history, the agent keeps only compact summaries of completed sub-trajectories in context.

Specifically (as depicted in figure 9b), given an initial state and a high-level goal, the agent proceeds as follows: (1) It generates a higher-order sub-goal (e.g., for the goal “Prepare breakfast”, a first sub-goal might be “Make coffee”); (2) while the sub-goal remains unmet, it iteratively generates an action (e.g., “Fill kettle with water”), executes it, observes the outcome (e.g., “Kettle is full”), and appends the resulting (action, observation) pair to working memory; (3) once the sub-goal is completed (e.g., coffee is brewed), the full sequence of (action, observation) pairs for that sub-goal is summarized into a compact representation (e.g., Subgoal 1: “Prepare coffee”, Observation 1: “Coffee is brewed and sitting on the table”), and the detailed sub-goal trajectory is replaced by this summary in the agent’s context. This process is then repeated until the end-goal is reached.[37]

The authors report their inspiration indeed to stem from cognitive sciences, where "humans typically decompose a complex problem into multiple subproblems, addressing each individually", also explicitly referencing the hierarchical nature of chunking that we introduced in Section 3.3.1.1. Although they do not explicitly address holistic models of working memory, we consider their approach to be clear evidence for the effectiveness of a summarization-based chunking strategy in reducing contextual load and enhancing LLM performance.

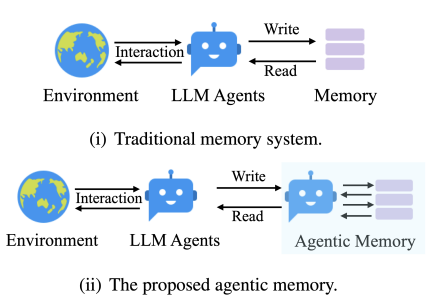
A-MEM W. Xu et al. (2025) introduce "agentic memory" (A-MEM), as a memory store that is *maintained* by an LLM, instead of merely used by it:

As introduced at the beginning of this subsection, RAG memory is typically implemented using vector databases (Y. Gao et al., 2024). However, the underlying embedding spaces are inherently imperfect. Particularly for tasks or domains unfamiliar to the embedding model, performance tends to degrade. To address this, a model is often fine-tuned on in-domain examples to “move certain representations closer or further apart” (cf. Schopf et al. (2024)).[38] However, it is not the embeddings themselves that are directly modified, but rather the function that maps text to these vectors. Put differently: the embedding space is effectively “warped” such that certain points are brought closer together while others are pushed further apart.

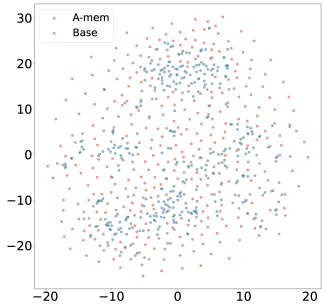
[37] Additionally, Hu et al. (2024) introduce the ability to re-expand "detailed past trajectory information [when it] becomes crucial for immediate decision-making". However, in their methodology, they do not explicitly state how this retrieval mechanism operates, other then stating that: "when the LLM determines that detailed information from a past subgoal is necessary, it generates a retrieval function". Hence, we leave this as a footnote. For further details, see their implementation at https://github.com/HiAgent2024/HiAgent.

[38] This specific fine-tuning approach, typically based on cosine similarity loss or a related objective, is also called *contrastive* fine-tuning. See prelude Section 1.1.

Figure 10
A-MEM (Xu et al., 2025): A memory system where the LLM actively shapes and indexes its own memory.



(a) *The idea behind A-MEM: Using the language model not only to read from memory (i), but to maintain and evolve its structure over time (ii).*
Figure Adapted from W. Xu et al. (2025).



(b) *The efficacy of A-MEM. Adapted from W. Xu et al. (2025):*
"T-SNE Visualization of Memory Embeddings Showing More Organized Distribution with A-MEM (blue) Compared to Base Memory (red) Across Different Dialogues. Base Memory represents A-MEM without link generation and memory evolution." (W. Xu et al., 2025)

A-Mem flips this paradigm: Instead of "warping the space", it modifies the input texts themselves so that items that belong together are embedded more closely. Specifically, it does so by generating contextually more, or less similar metadata in the form of tags, keywords, and descriptions of an item's payload. The key idea: A LLM first (1) *verifies* wether two texts should actually be in the same neighborhood, and then, (2) *manipulates* their embedding context, as to move them closer together. W. Xu et al. (2025) call these two steps "link generation" and "memory evolution" respectively.

Specifically: In A-Mem, a "memory node", i.e. a single contained item in memory to be fed to a vector database, consists of (1) a textual payload, (2), LLM generated keywords, (3) LLM-generated tags, (3) LLM-generated contextual descriptions – these will all be used to generate an embedding of the item – and (4) a set of verified connections to other items.

Link-Generation: For any new memory item, the system retrieves its k-nearest neighbors in embedding space (where per default, $k = 5$). The LLM evaluates these candidates and may choose to establish verified links to those it deems contextually related.

Memory Evolution: Once links are formed, the LLM may update the tags, keywords, and contextual descriptions of both the current and neighboring memory nodes. After evolving a specified number of memories, the updated nodes are re-embedded with their revised metadata,[39] effectively causing

[39] More precisely: To prevent excessive rewriting, A-MEM performs full memory consolidation only after a configurable evolution threshold (e.g., every x successful evolutions), allowing updates to be batched and stabilized before re-embedding. Also note, that in practice both, link verification and context annotation is performed in a single LLM call.

semantically related items to move closer together in the embedding space, while unrelated ones drift further apart.

In doing so, W. Xu et al. (2025) provide a flexible storage system, claiming to establish a new state of the art for long-horizon, open-ended agent tasks requiring persistent memory and context-sensitive retrieval. We regard it as a potential component within a broader working memory framework (e.g., as an implementation of an episodic buffer) or alternatively as a strong baseline for comparison.

After highlighting selected memory architectures for LLMs on, let us now proceed to selected material on the structure of memory items.

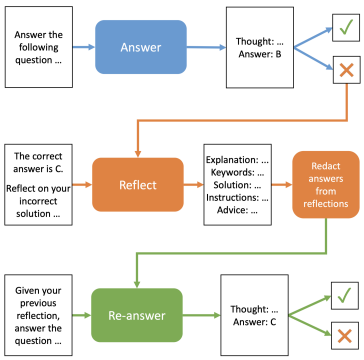
3.3.2.2 Memory Representations for Reasoning: The Ability to Self-Reflect

The importance of "chunking" in cognitive science of working memory (see Section 3.3.1.1 gives reason to suspect that a working memory system serving as a mere retrieval function of raw prior LLM thoughts – as Graph of Thoughts (GoT) may formally suggest (Section 3.2.1.4) – might be suboptimal. For instance, one might not merely want raw instances of prior errors to avoid, but instead desire a set of compiled learnings that enable a more fruitful generation instead (for further discussion, see Subsection 4.3.2). In light of this, we investigated possible approaches most akin to chunking, that may enable a targeted abstraction of thoughts beyond the summarization strategy discussed in the previous section. This leads us to the concept of *Self-Reflection*. [40]

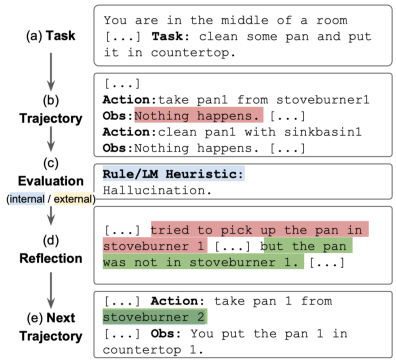
Self-Reflection Self-Reflection can generally be regarded as an umbrella term for a metacognitive strategy in which an LLM is given a type of feedback (error signal) to evaluate a previously generated chain of thought (CoT; introduced in Section 3.2.1.2), and then uses the output to guide a subsequent attempt at solving the same problem (see Figure 11). From hereon, we will simply refer to the output produced during such evaluation, the textual feedback used as surrogate for a subsequent generation, as a *reflection*.

[40] As outlined in the prior Subsection 3.3.2.1 with Hi-Agent (Hu et al., 2024), summarization can be seen as a viable "chunking strategy". However, we treat Self-Reflection separately, as it addresses not just general compression but targeted abstraction of reasoning failures. While the literature on Self-Reflection frequently refers to LLMs as "agents" (X. Huang et al., 2024; Renze & Guven, 2024; Shinn et al., 2023), it has also been described as an "indispensable component" of agentic planning itself (X. Huang et al., 2024), suggesting that a separate treatment may be warranted despite some inevitable conceptual overlap.

Figure 11
Two example methods using the idea of Self-Reflection.



(a) *A Self-Reflection setup in which the correct answer is used as error signal, but later redacted from the produced reflection (not depicted in the figure). Adapted from Renze and Guven (2024).*



(b) *A Self-Reflection setup with variable error signal source, on the tasks of "decision making". Adapted from Shinn et al. (2023); originally included three tasks, truncated here for brevity.*

Self-Reflection is related to traditional reinforcement-learning (RL), even as far as being referred to as "verbal RL" (Shinn et al., 2023). Regardless, it does not provide a formal guarantee for success (X. Huang et al., 2024; Shinn et al., 2023). However, it is advantageous in allowing for more nuanced and (textually) interpretable feedback; capabilities valuable for a memory storing LLM experiences (Shinn et al., 2023).

Various setups apply this idea under different names. For instance

- Madaan et al. (2023) introduce *Self-Refine*, which involves iteratively improving an initial answer by alternating between the following steps, until a stopping condition is met:
  A. **Reflection (or Feedback) Generation**: Prompting the LLM to generate a new reflection in the context of all prior ones. This results in a list of numerical scores.
  B. **Refining (applying the Feedback)**: Incorporating the new reflection into a subsequent generation attempt.
- Shinn et al. (2023) introduce *Reflexion* (Figure 11b), a system with three main active components, that also act iteratively upon another:
  – **The "Actor"**: An LLM, which generates a new trajectory of (action, observation) pairs based on a task prompt, the trajectory from the previous attempt (if any), and a context window containing up to three prior reflections.

  - **The "Evaluator"**: This may be an LLM, a heuristic, or another model. It assesses the latest trajectory along with all previously generated reflections to output a single reward score and a binary decision on whether the task has been successfully completed or requires another attempt.
  - **The "Self-Reflection Model"**: An LLM that takes as input the Evaluator's reward score, the respective trajectory to reflect upon, and all previously generated reflections, to produce a new textual reflection.
- Renze and Guven (2024) survey multiple Self-Reflection formats by utilizing the answer itself as a reward signal, and then redacting it from the generated reflection, as to not (directly) leak it into the subsequent generation attempt (Figure 11a).

These approaches differ especially in the type of error signal used in (generating) the reflection – textual vs. scalar – and the origin of such signal – internal (from the LLM itself) vs. external.(Renze & Guven, 2024).
The latter is especially crucial, as evidence regarding LLMs' ability to reliably identify their own errors – that is, the "internal" variant, where LLMS act as their own source for error signaling – remains inconclusive (Renze & Guven, 2024). We acknowledging this uncertainty, and adopt the working assumption that they can do so to a useful extent, and "only expect this paradigm to get better over time" (Shinn et al., 2023). Thereby, we follow Renze and Guven (2024), who (despite reporting the inconsistency) still state that LLMs can "identify errors [in their CoT], explain the cause of these errors, and generate advice to avoid making similar types of errors in the future."

While this ability may already point to the gap we identified in illustrated reasoning paradigms (Section 3.2.2)[41], we further motivate Self-Reflection as the primary item type stored in an LLM's working memory in Section 4.3.2.

Accordingly, we present best practices for generating reflections, as indicated by the surveyed literature: (1) their different types, ideal (2) contents, (3) frequency, and (4) volume.

[41] However, as already noted, Self-Reflection with internal feedback has been partially incorporated into such reasoning paradigms: Beginning with Tree-of-Thought's ability to score thoughts for backtracking, and continuing with the Refine operation in Graph-of-Thought (see Subsections 3.2.1.3 and 3.2.1.4, respectively).

Types of Self-Reflection Renze and Guven (2024) discerns between different types of reflections to generate. We further sub-divide them into two categories. In their setup (Figure 11a), a model is prompted to reflect on the problem, its solution, and the correct answer, to generate one of the following types of reflection (quoted directly):

1. Reflection **not leaking** Gold-Answers:
    - "Retry – [the LLM is] informed that it answered incorrectly and simply tries again.
    - "Keywords – a list of keywords for each type of error."
    - "Advice – a list of general advice for improvement."
2. Reflections indirectly **leaking** Gold-Answers:[42]
    - "Explanation – an explanation of why it made an error."
    - "Instructions – an ordered list of instructions for how to solve the problem."
    - "Solution – a step-by-step solution to the problem."

They report an accuracy improvement over the non-reflecting baseline (79%) across all reflection variants. Even the simple retry strategy yielded (while yielding the lowest gain), still achieved 83% accuracy. The combination of all strategies resulted in the highest performance, reaching 93%.

Contents of Reflections

- **Combination of Different Reflection Strategies**: As Renze and Guven (2024) show, this variant scored highest. Their results also suggests that Self-Reflections with more structured and abstract feedback tend to outperform minimal strategies like a simple re-try.
- **Feedback that is Strong and Diverse**: As Madaan et al. (2023) note, performance does not increase monotonically with the number of reflections, since quality in one aspect improving can cause another to drop. To counteract this, they include different numerical scores for different quality aspects into their reflection. Additionally, they note that their majority of errors recorded stem from faulty reflections themselves, rather then erroneous attempts to incorporate them (specifically, they report errors to be distributed with 33% due to feedback

[42] As noted by the authors: "It is important to note that the Self-Reflections generated by the Explanation, Instructions, and Solution agents indirectly leak information about the correct answer without directly specifying the correct or incorrect answers. However, they generated this information on their own based on nothing more than being provided the correct answer during the Self-Reflection process." (Renze & Guven, 2024).

inaccurately pinpointing the errors location; 61% due to the suggestion of an inappropriate fix; 6% due to inappropriate implementation of feedback).

- **Feedback that is "Actionable", and "Specific"**: Generated reflections are found to yield better results, if they contain a concrete action that is likely to improve the output (are "actionable"), and identify concrete parts of the output to change (are "specific") (Madaan et al., 2023).
- **Concrete Attempts Showcasing the Error**: In line with the last point, Shinn et al. (2023) show that merely providing the reflection itself as a surrogate is not as effective, as also providing the item that is being reflected on. Specifically, including the most recent trajectory into the actor helped boost performance.

Frequency and Timing of Reflection

- **Multiple Iterations are Beneficial**: Early iterations were found to "significantly enhances the quality of the output, although the marginal improvement naturally decreases with more iterations." (Madaan et al., 2023).
- **Often is Potentially Better**: What directly follows from the above, is that for setups in which a binary classifier decides wether a solution should live though another refinement iteration (as in Shinn et al. (2023)), it is better to simply try again when uncertain. Shinn et al. (2023) show plateauing performance curves. Hence, degeneration may not be feared.
- **Application at any Granularity Level**: Shinn et al. (2023) reflect after every trajectory (not within), regardless if its comprised of one or multiple LLM calls. This indicates that Self-Reflection may be used at any granularity level in reasoning.

Number of Reflections to Include in Subsequent Generation The optimal number of reflections to incorporate in subsequent generation appears as an open question.

For instance, Shinn et al. (2023) include up to three reflections within the context window for each new generation attempt and highlight the need for future work to enhance Reflexion's memory component with more advanced structures. Notably, in the process of obtaining a reflection, Shinn et al. (2023) have their evaluator access all previously generated reflections to produce a numerical score, which is then used in generating the new reflection – but this new reflection itself is generated

only in the context of the current trial. In contrast, Madaan et al. (2023) generate each reflection explicitly in the context of all prior reflections.
Other approaches such as Park et al. (2023), that target a different application for reflection [43] incorporate as many prior items (including reflections) as possible when generating new reflections.

Considering that retrieval errors tend to increase with longer contexts (Liu et al., 2023), we remain cautious. Intuitively, since distractions should be minimized, an excessively long context of reflections could lead to retrieval noise, thereby reducing the effectiveness of reflections in terms of their actionability and concreteness.

This concludes the review of candidates for reasoning-related memory architectures and item representations. Building on these foundations, Chapter 4 synthesizes prior work with cognitive working memory models and introduces our proposed working memory design for large language models.

[43] Park et al. (2023) propose a multi-agent social simulation in which reflection is employed to synthesize high-level insights about an agent's character from accumulated memory logs. This use of reflection focuses on internal coherence and behavioral believability within a virtual society, which distinguishes it from reflection techniques aimed at improving task-specific performance metrics.

Chapter 4

Approach

Contemporary reasoning paradigms exhibit a fundamental limitation: The inability to selectively retrieve relevant information from an arbitrarily growing reasoning graph (Section 3.2.3). In this chapter, we present our approach to addressing this issue by modeling a working memory (WM) system for large language models (LLMs), conceived as a computational counterpart to holistic frameworks of WM in cognitive science; specifically, the Multi-Component Model (MCM; introduced in Section 3.3.1.2).

Our work follows a dual trajectory: While we focus on applying WM within the Graph-of-Thought (GoT) framework (introduced in Section 3.2.1.4), we also pioneer[44] a previously absent link between the MCM and LLMs by synthesizing the core of an extensible architectural prototype for practical integration.

Thereby, we present the core architecture (Section 4.3) and our specific memory variants separately (Section 4.4), the latter of which we evaluate in the subsequent Chapter 5.
We begin by reiterating our research niche, and outlining our central design goals and initial scope in Section 4.1. These serve as the basis for bridging the conceptual gap between cognitive and computational perspectives in Section 4.2, where we further tighten our scope and lay the theoretical foundation for our approach.

[44] That is, to the best of our knowledge, there is no link between holistic frameworks of WM and LLM research.

4.1 Guiding Principles & Initial Scope

In this Section, we restate our research niche, and outline our high-level scope design goals in building a working memory (WM) system for large language models (LLMs).

Our work targets the intersection of two research gaps:

1. **Primary Gap: The absence of a mechanism for retrieving relevant material from growing reasoning graphs** (Section 3.2.3). We attribute this limitation to the leading reasoning paradigm[45] Graph-of-Thought (GoT; introduced in Section 3.2.1.4), which thereby defines our application context for a WM system. The decision to focus on this gap also entails to refrain from addressing another: The staticity constraint of GoT (Section 3.2.2.1).
2. **Secondary Gap: The broader lack of connection between LLM research and established frameworks of working memory in cognitive science** (Section 3.3).

To address the primary gap, we formulated the following research question:

> **Research Question:** How might a WM mechanism for LLMs be designed to selectively obtain relevant material from an arbitrarily large reasoning graph, when generating a new thought in Graph-of-Thought?

After a close investigation of established cognitive models of WM (Section 3.3.1), we tightened our focus on the so-called Multi-Component Model (MCM; Section 3.3.1). This brings us to formulate a secondary, design-oriented question:

> **Design Question:** How might the Multi-Component Model of WM be transferred architecturally to the domain of LLM research?

Accordingly, our approach follows a dual trajectory: Our primary objective is the development of a practical WM implementation for reasoning in GoT. In doing so, we prototype a minimal and abstract core architecture that aims to provide a first computational foundation for transferring the MCM to the domain of LLMs.

[45] With "reasoning paradigm" we continue to refer to prompting strategies that model reasoning structures for LLMs.

In light of this duality, we outline our guiding principles:

1. **No Dynamic Planning of Reasoning Structure**: Our core architecture and implementation in GoT will expect a reasoning structure to be in place. We will not (yet) consider to support dynamically scheduling operations in GoT (i.e., modifying the Graph-of-Operations [46] at runtime), as reflected by our decision not to focus on the staticity constraint of reasoning paradigms (Section 3.2.3).
2. **Maximal Task-Independence**: Our core architecture and implementation in GoT should generally be applicable across reasoning paradigms, requiring no task-specific setup (that is, we aim to not contribute additional rigidness, i.e. staticity, as described in Section 3.2.2.1).
3. **Adaptability to Reasoning Structure and Independence from GoT**: While we focus on GoT, we anticipate that an effective WM system may generalize to both finer-grained and coarser reasoning scenarios [47]. For instance, this could contribute to overcoming reasoning errors within the CoT of contemporary reasoning LLMs at runtime, or allow for improved memory of (or between) LLM agents, respectively (cf. Section 7.1).

   Therefore, our WM core architecture should be applicable across diverse reasoning granularity and structure, without relying on GoT. Specifically, it should remain architecturally flexible along three key dimensions: (1) *Thought Granularity* – the informational scope of a single Thought (e.g., a sentence or a paragraph); (2) *Reasoning Granularity* – the number of Thoughts processed within a single transaction (e.g., one at a time or entire subgraphs); and (3) *Thought Representation Format* – the structural form in which a Thought is encoded (e.g., textual or latent, as in Hao et al. (2024)).
4. **Orientation on Established Principles**: Our implementation in GoT should be centered around relevant findings of memory used for LLM-based reasoning (Section 3.3.2).
5. **Modular and Extensible Architecture**: In accordance with the above, our WM core architecture should be designed as a modular system that aims to support future extensions in alignment with our interpretation (Section 3.3.1.3) of the MCM.

[46] To briefly reiterate from Section 3.2.1.4: In GoT, the currently **static** Graph-of-Operations (GoO) defines the space of allowed sequence of reasoning operations, each with input and output thought sets. Inputs are the union of immediately preceding operations, thus defining dependencies and node degrees in the thought graph. See Section 3.2.2.1 for additional discussion of staticity.

[47] As has been clarified in the beginning of Section 3.2.1:

(a) The granularity of a "thought", a node in the reasoning graph, is highly task dependent.

(b) Fundamentally, any linear sequence of thoughts can be viewed as a walk through an implicitly present reasoning graph. That is, regardless of whether such a graph is captured explicitly, as in Tree-of-Thought (ToT; Section 3.2.1.3) or GoT, or not, as in Chain-of-Thought (CoT; Section 3.2.1.2) or contemporary reasoning models (prelude, Section 1.3)

6. **Minimal Viable Design Focus**: Our approach prioritizes validating core mechanisms under minimal assumptions, deferring complexity until empirically justified.

This defines our high-level scope, which we subsequently refine in two stages: First, in Section 4.2.1, we synthesize insights from cognitive science, related work on LLMs, and the GoT framework, to determine which aspects of the MCM to model and which to defer to future work. We then further narrow our focus when specifying the concrete algorithmic behaviors and design assumptions of our memory implementations in Section 4.4.1.

4.2 Theoretical Foundation

In this Section, we bridge conceptual gaps between cognitive science perspectives on working memory (as summarized in Section 3.3.1.4), the reasoning environment we aim to enhance – the Graph-of-Thought (GoT) framework (introduced in Section 3.2.1.4) – and relevant concepts from existing LLM research (Section 3.3).

In the first part (Subsection 4.2.1), we identify core components of human working memory, particularly those defined by our interpretation of the Multi-Component Model (Section 3.3.1.3), that lie outside of our modeling scope, and which to leave to future work.

In the second part (Subsection 4.2.2), we turn to related work on memory for reasoning in LLMs (Section 3.3.2), and in contemporary natural language processing in general (prelude, Chapter 1), to summarize practical approaches for modeling the remaining components that fall within our scope.

This synthesis lays the theoretical groundwork for the top-down system design that follows, beginning in Section 4.3, where we present the core architecture.

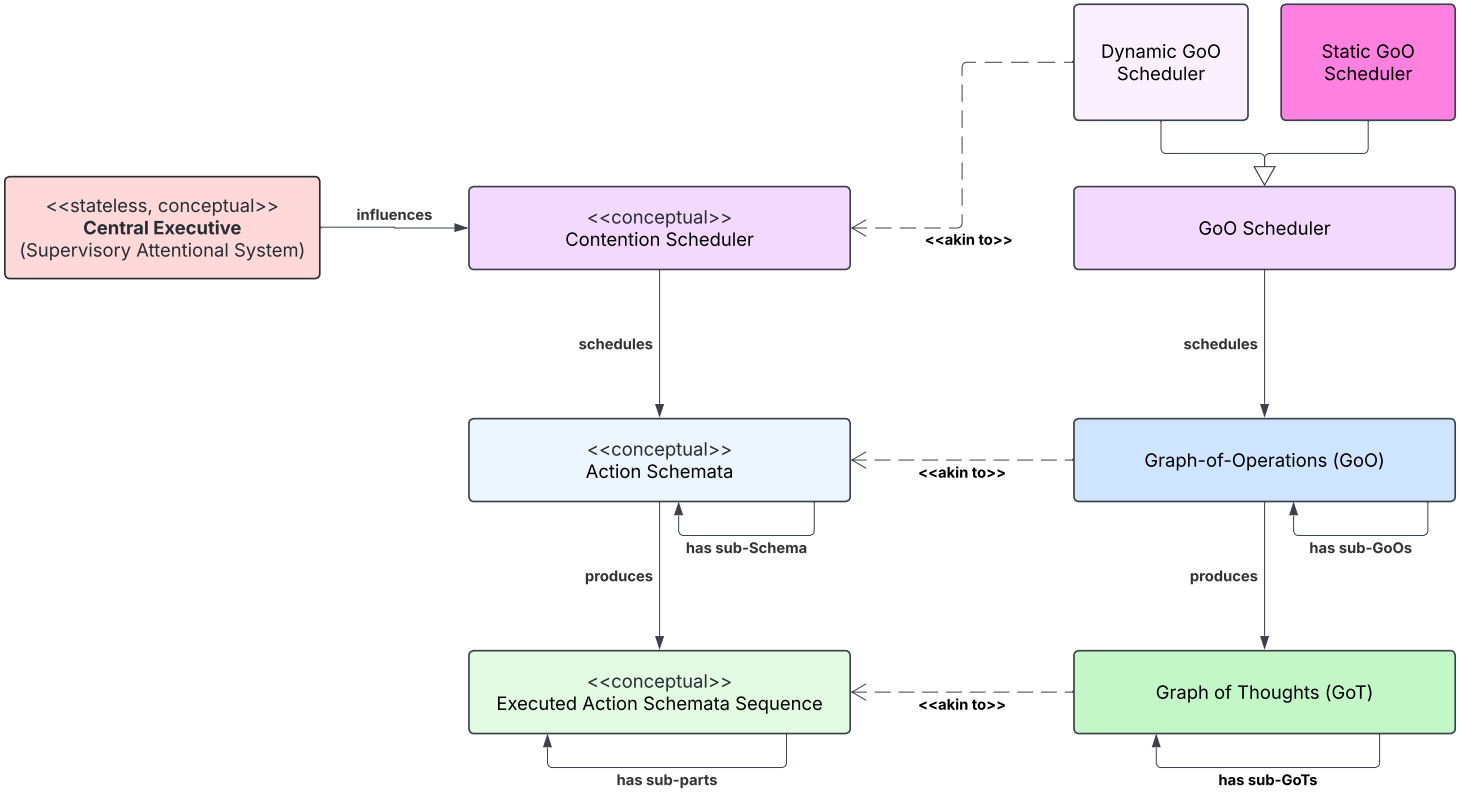
4.2.1 Synthesis I: Scoping Cognitive Science to Graph-of-Thought

We revisit the architectural components of the Multi-Component Model (MCM) of human working memory, as interpreted in Section 3.3.1.3, to determine which elements can be excluded from our modeling scope (Section 4.1) within the Graph-of-Thoughts (GoT) framework.
To summarize up front: We rule out several key components of the original MCM. The Central Executive (CE), the Contention Scheduler (CS), a dedicated long-term memory (LTM) module, and modality-specific slave buffers, whilst acknowledging the potential relevance of dedicated pre- and post-processing mechanisms.

Please note, that we still anticipate the excluded components to be integrated at a later stage, and that this selection reflects our intention to begin with a minimal yet meaningful subset of functionality. As such, we refer to Section 4.3.5 for a discussion on extensibility.

Figure 12

A class diagram mapping theoretical constructs from our interpretation of the Multi-Component Model of working memory (MCM) (left; cf. Section 3.3.1.3) to central components in the Graph-of-Thoughts framework (right; cf. Section 3.2.1.4). Note*: GoT currently employs a* Static GoO Scheduler*, one that schedules operations via breadth-first search over a predefined Graph-of-Operations. This staticity (Section 3.2.2.1) removes the need for a contention scheduler and, by transitivity, bypasses the Central Executive.*



<<conceptual>>: Denotes a theoretical construct from cognitive science. Classes are not defined in terms of attributes, operations, or formal interfaces.

<<akin to>>: Stereotype on UML dependency arrows, indicating that the concrete class functionally subsumes the role of the <<conceptual>> class. Can (per def. of the above) not imply interface conformance but assumes the conceptual role is approximated within the implementation.

Fixed Action Schemata, No Central Executive, No Contention Scheduler As discussed in Section 3.3.1.4, action schemata can be conceptualized as hierarchical transaction templates operating on WM and beyond. Regardless of their cognitive origin, they inherently represent a form of planning. Conceptually, creating an action schema can hence correspond to constructing a Graph-of-Operations (GoO) in the GoT framework (see Figure 12, blue boxes).

Accordingly, a class that **dynamically** schedules operations from a GoO can be seen as functionally akin to the Contention Scheduler introduced by Norman and Shallice (1986). However, if scheduling always leads to the same outcome, i.e., the GoO is **static**, then the need for any such scheduler becomes obsolete. Transitively, this also eliminates the role of the Central Executive, which serves primarily to bias a dynamic scheduling process. (Note that in Figure 12, no dependency is shown between the "Static GoO" and the "Contention Scheduler", and hence, none to the Central Executive.)

Since this precisely reflects our scope, we likewise refrain from dynamically planning memory operations. Instead, we rely on fixed, hard-coded transaction algorithms.

LLM as Long Term Memory LTM is crucial for WM. Though interactions remain unclear, it "may effect processing at every stage" (Baddeley, 2010). In accordance with Section 3.3.1, we summarize the following properties a potential LTM may fulfill:

- It must generally be capable to *learn* representations by abstracting or grouping ("chunking") data.
- Specifically, it should support the idea of hebbian learning. That is, in the context of WM, a single trial in WM should be able to elicit a change in LTM (Burgess & Hitch, 2005).
- It must be capable to hold representations of task-items and action schemata.

We then came up with three candidates for LTM in the context of GoT, and evaluated them against the criteria:

- **LLM Weights**: The LLM itself is widely assumed to support abstraction (i.e., chunking) and can learn via fine-tuning or in-context adaptation.[48] However, while LLM parameters can effectively "store" general trends in the data, they do not replace external storage (Mallen

[48] In this context, it may also be note-worthy to consider a WM as a source for training data. See section 4.3.5 for further notes.

et al., 2023). Instead, LLMs are emerging as "general-purpose engines for data management" (Zhou et al., 2025).

- **Dedicated LTM Memory Modules**: Are the direct implication of the limitation of the former, with approaches such as A-MEM (W. Xu et al. (2025); see Section 3.3.2.1) offering a promising path of employing LLMs as storage managers. However, this would necessitate modeling dedicated mechanisms for (hebbian) learning and chunking.
- **Traditional Storage Media**: Evolving text corpora or databases can store arbitrary data but do nothing more. Essentially holding only a subset of the capabilities of the prior candidate.

Therefore, while a dedicated LTM module may have clear conceptual advantages, explicitly modeling and integrating one lies beyond our current scope. As a result, relying on the LLM itself as a black-box LTM mechanism emerges as the most practical (albeit presumably imperfect) solution.

No Modality-Specific Slave Buffers, But Room for Pre- & Post-Processing As noted (in Section 3.3.1.2, the MCM includes two modality-specific slave buffers that serve as gate-keepers to the perceptual motor system: One for phonological and one for visuospatial information. These buffers are partially active and capable of transferring information into and from the episodic buffer.

This dual role is critical for handling multimodal information in human working memory and suggests potential relevance for cross-modal LLM scenarios (discussed in Section 7.2.2). Moreover, the gate-keeping function – pre-processing incoming sensory data before integration into WM, and potentially post-processing information upon output – may be critical.

While we acknowledge the potential need for such dedicated pre- and post-processing mechanisms when interacting with a WM system, we deliberately avoid modeling intermediary buffers in our setting. Since our scope is limited to text-only reasoning, we consider such additional components unnecessarily complex and therefore exclude them from our approach, while focusing on the central storage unit within the latest variant of the MCM: The Episodic Buffer.

4.2.2 Synthesis II: Mapping Cognitive Science to Related Work on LLMs

In the preceding subsection, we identified which components of the multi-component model of human working memory (MCM), as interpreted in Section 3.3.1.3, fall outside the scope of our approach: The Central Executive, Contention Scheduler, Long-Term Memory, and modality-specific slave buffers. As a result, WM's ability to integrate result oriented processing functionality, has been relaxed to limit the manipulations of items to pre-selected linear transactions operating on WM.

In this subsection, we discuss how surveyed research on memory for reasoning in LLMs (Section 3.3.2), and in contemporary natural language processing in general (NLP; see prelude, Chapter 1), could address the core MCM components that remain within our modeling scope: A textually based **Episodic Buffer**, along with the concepts of **capacity**, **forgetting**, and **chunking**.

This conceptual selection sets the stage for the decisions presented in the preceding Sections of this Chapter; starting with the core architecture design in Section 4.3.

Capacity Limit & Forgetting As outlined in Section 3.3.1.1, cognitive science models universally agree that working memory (WM) has limited capacity. Various theories explain forgetting mechanisms in WM, which we summarized as *decay-, resource-,* and *interference-based* theories.

While interference-based forgetting, where items overwrite or "steal" features from others, may be highly plausible, modeling such mechanisms comprehensively may also be considerably more complex.[49] In contrast, decay-based models simply remove items that have not been attended to for a certain period. This is the mechanism adopted by the MCM (Section 3.3.1.2), making it a practical foundation for our design.

From a computational perspective however, a complementary or alternative approach may be lazy forgetting grounded in the resource-based view of capacity limits: A fixed capacity threshold is set, and once reached, items are ranked (e.g., by recency), with the lowest-ranked items discarded until enough capacity is freed.

[49] In hindsight, we note that this may not necessarily be as complicated as initially anticipated. For instance, semantic drift of repeated summarizations with newer representations of the same concept may effectively represent an interference-based form of forgetting. We discuss this idea, among other optimization potentials and limitations, in Section 6.2.2).

In both cases, whether forgetting is implemented lazily or otherwise, the recency effect is readily applicable and has already been modeled using exponential decay, as demonstrated, for example, by Park et al. (2023).

Chunking The idea of chunking is central in the cognitive science of WM. Although it remains not fully understood, it is commonly interpreted as the process of forming groups or hierarchies of items in working memory (and beyond; Burgess and Hitch (2005) and Hitch et al. (2025); as introduced in Section 3.3.1.2). Therefore, it is directly related with a central question we have to ask when designing a WM for LLMs: Of what format should items in memory be, and how may they be structured?

We highlight practical approaches that may serve as a "chunking strategy" for LLMs:

- **Summarization**: In Subsection 3.3.2.1, we introduced Hi-Agent, where Hu et al. (2024) demonstrate that summarizing prior agent trajectories, sequences of *(action, observation)* pairs, effectively improves agent performance. Their introduction of 1-layered "chunk trees" effectively reduces distractions of non-related trajectory information, of which only the summaries (i.e., roots) are included. Additionally, such a tree can be expanded to show detailed information recorded in its children, when necessary.
- **Self-Reflection**: The process of abstracting a thought into a higher-order representation for future reuse can likewise be understood as a form of chunking. In Subsection 3.3.2.2, we introduced the concept of *Reflection* which is the result of a metacognitive process (Renze & Guven, 2024); an abstraction over the original thought, targeted specifically towards its flaws, or improvement potential.
- **Various Latent Representations**: In NLP, latent representations, such as text embeddings or intermediate activations extracted from specific layers of an LLM, enable abstract comparison of textual inputs in a latent space by encoding semantic similarity beyond surface form. These representations can serve as a basis for clustering or grouping memory items and thus provide an implicit mechanism for chunking, though additional mechanisms (such as similarity thresholds, classification heads) are often required for operationalization (See Jurafsky and Martin (2025, Chapter 6).)

Episodic Buffer The episodic buffer is a fully *passive* store with a capacity limit, that is episodic in regard to time and space (Baddeley, 2000; Hitch et al., 2025). While it is generally conceived to support complex multimodal information, we restrict its scope here to textual data only.

Its passiveness may be understood in varying degrees of strictness. We defer our own understanding to the discussion of our core architecture, in Section 4.3.3.

The interpretation of being episodic in both time and space remains underspecified in the surveyed literature (Baddeley, 2000; Hitch et al., 2025) and is open to design decisions. For instance, "time" may refer to dimensions such as insertion order, last access time, or recency of use. Similarly, "space" may be defined arbitrarily, e.g. through semantic or other forms of distance, likely implying a high degree of task-specificity (that is, the specific notion of space may be highly context-dependent). Implementations may therefore range from linear timestamped lists to arbitrarily complex multidimensional ordering schemes.

In the context of reasoning-related memory in LLMs, however, we find the episodic notion typically reduced to a simple log of *(action, observation)* pairs. In Hi-Agent (Hu et al., 2024), for example, an "episode" corresponds to a goal-directed sub-trajectory within the overall agent trace. This would correspond to walk on a reasoning graph, or (even simpler) the chain of nodes up to its root.

This concludes our design space setup, from which we will now draw, in outlining our core architecture, in Section 4.3.

4.3 Core Architecture

In this section, we present the core architecture of our extensible working memory (WM) system for large language models (LLMs), developed in response to the goals and scope outlined in the preceding sections. The design adheres to the scope constraints defined in Section 4.1 and further refined through our synthesis in Section 4.2, all of which build on our interpretation of the Multi-Component model of WM (Section 3.3.1.3). We explicitly account for out-of-scope components during architectural planning, and provide further discussion of extensibility in Section 4.3.5.

4.3.1 Overview

As depicted in Figure 13, our working memory system follows a hybrid architecture, consisting of four primary layers retaining a vertical control flow.[50] The I/O package on the right centralizes communication semantics horizontally across multiple layers to ensure consistent request and response structures.

We briefly introduce each layer, with essential details provided in the following subsections:

- **Layer 3: Facades**

  This layer contains independent *Facade* classes that expose an application-specific interface to external clients. Internally, each facade delegates to one or more *Dispatchers*, while asserting the I/O types expected by the contained *Transactions* for defensive verification upon construction.

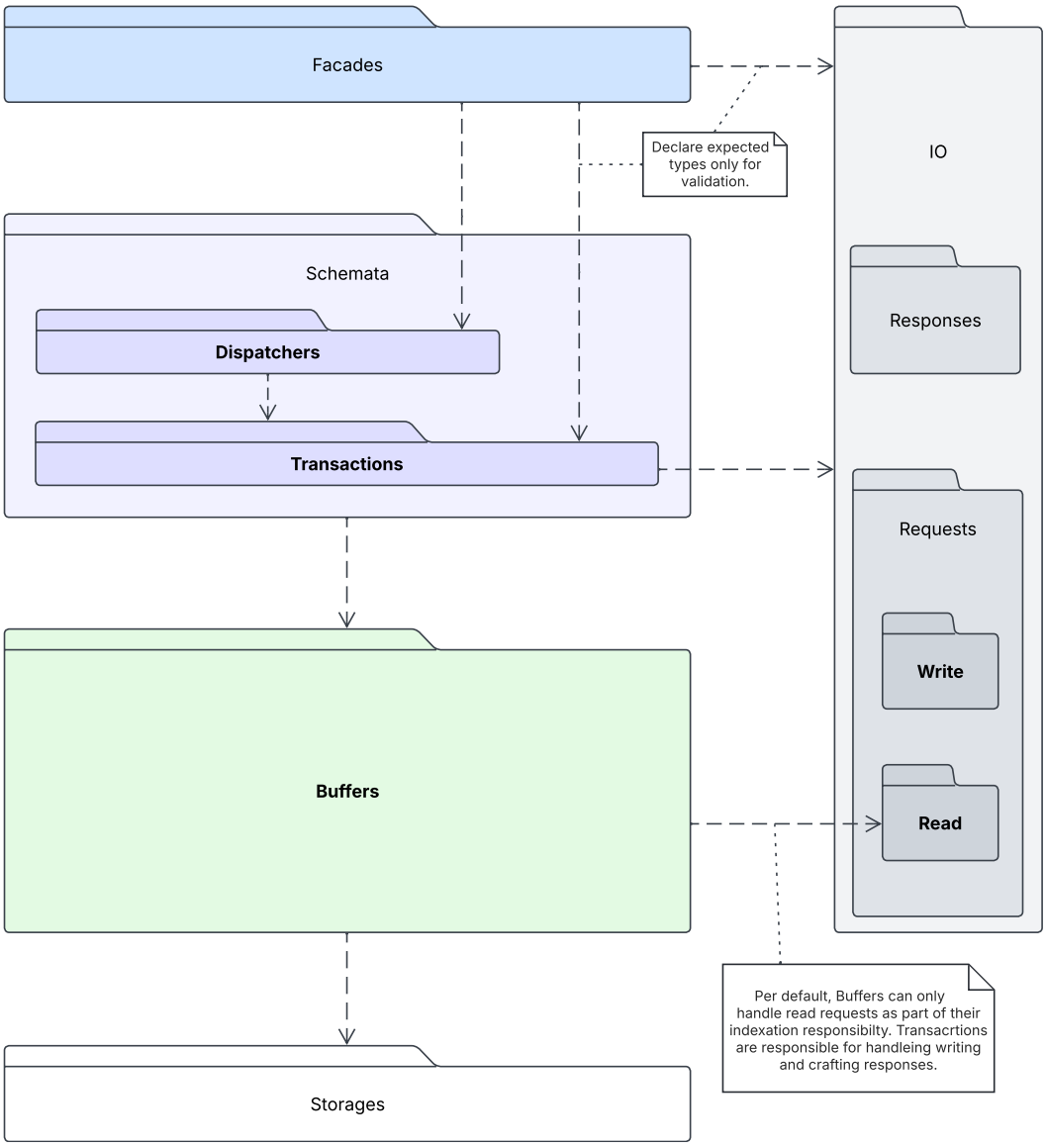
- **Layer 2: Schemata**[51]

  This layer unifies *Dispatchers* and *Transactions*. Dispatchers execute Transactions, and Transactions define their expected input-, output-, and target-buffer- types, to encapsulate templates for concrete operation sequences on buffers. Currently, only regular (i.e., linear and non-nested) transactions are implemented.

[50] At the time of writing, the implementation may still undergo minor structural refactoring to fully align with this package layout. Nonetheless, the core responsibilities of each layer are already clearly defined. Local utilities such as the *SnapshotService* for the buffer layer are omitted for simplicity.

[51] The layer name reflects its conceptual alignment with "action schemata" in cognitive science, and may be extended to support hierarchical orchestration. See Section 4.3.5 for further notes on extensibility.

Figure 13
High-level package diagram of our working memory system. The architecture follows a layered execution model with a shared horizontal I/O module. Core classes are omitted for brevity.



- **Layer 1: Buffers**

  This layer houses the central memory components of WM, namely *Buffers*. Each buffer is typed, among other things, by (1) the kind of memory item it can hold, (2) the request type it can answer, and (3) the storage backend it relies on. These roles are detailed in the following.

- **Layer 0: Storages**

  This layer provides concrete *Storage* implementations as backends to the buffers above. While the current implementation includes only in-memory storage, it can be extended to support traditional concurrency-enabled databases, vector stores, etc.

In the following, we detail the standard I/O types integrated into the core design, leading us directly to the connected high level read and write transactions (Subsection 4.3.2). This, together with the buffer hierarchy (Subsection 4.3.3 and 4.3.4), sets the stage for our implementation variants in the next Section (4.4). The remaining layers are omitted, as their current structure is relatively straightforward and not central to this thesis.

4.3.2 The Standard I/O and Associated Transactions

In the core architecture, the I/O module defines the responses returned from, and the requests that can be send to the WM system. Following the scope of our research question (to obtain relevant reasoning material from a WM store for the generation of a new thought) directly informs the high-level read and write process depicted in Figure 14. This process remains agnostic to the reasoning environment (e.g., the task, the thought granularity, or format) and memory implementation (see Section 4.3.5 for further discussion).

While the architecture generally supports arbitrary request and response types, we integrated a specific variant centered on the idea of Self-Reflection (see Section 3.3.2) into the core design.

This leads us to conceptualize a concrete variant of **working memory as an advanced reflection store**: Materials are injected, internally reflected upon, and subsequently processed into buffer items (Figure 15a). When queried with the context for a new thought, the system retrieves the most relevant stored reflections and synthesizes a compressed set of (query-specific) surrogate reflection(s) as response (Figure 15b).

Figure 14

BPMN diagram of the simplified high-level reasoning process (top lane, white) interfacing with a working memory system (bottom lane, blue) to first retrieve a relevant reasoning surrogate (green subprocess; AbstractReadTransaction*) for generating a new thought, and then update the memory with the result (red subprocess;* AbstractWriteTransaction*). This process accommodates arbitrary levels of thought granularity and query context. It assumes no specific memory structure beyond the existence of a single buffer (abstracting away the actual multi-buffer setup), further defining optional activities for pre-processing write requests and post-processing read results to derive a maximally useful surrogate for subsequent reasoning (Non-BPMN-compliant dataflows go across subprocess boundary to avoid clutter).*

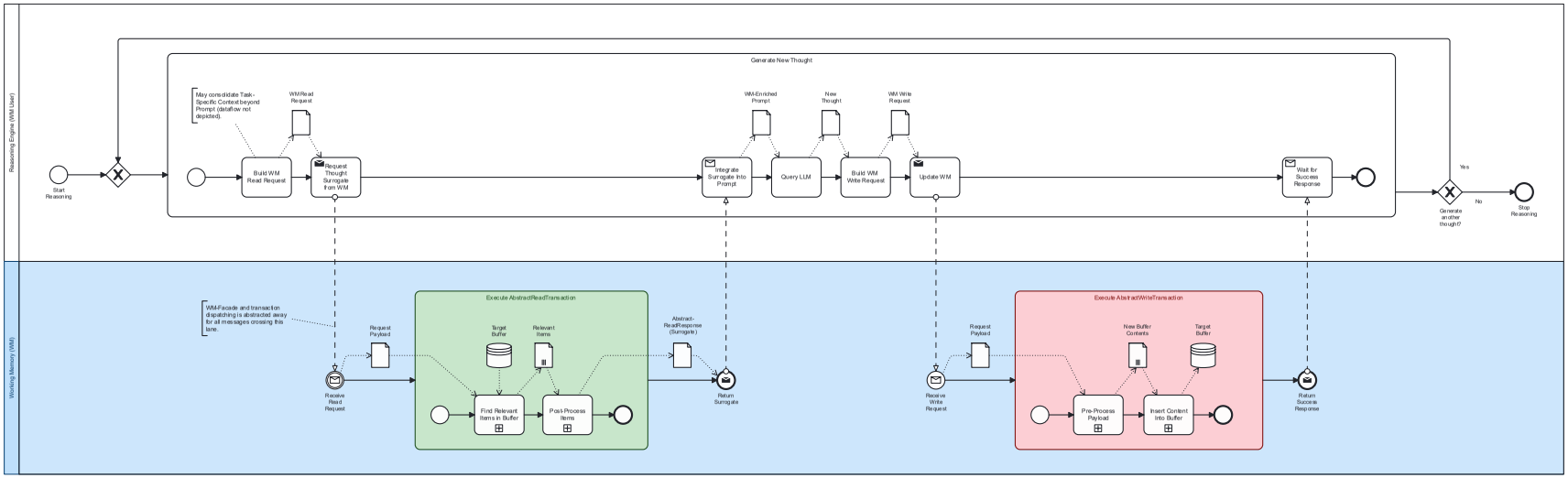
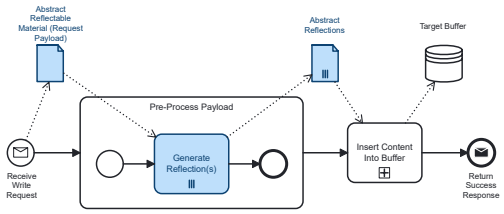
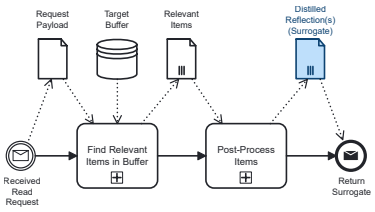


Figure 15
BPMN diagrams of the high-level read and write transactions associated with reflections as I/O. For concrete variants, see Section 4.4. Reflection-specific differences to the root transactions processes (depicted in Figure 14) are highlighted in blue.



(a) *The* AbstractWriteReflectableTransaction *defines the general process of inserting a reflection derived from any kind of* AbstractReflectableMaterial *into a buffer. This transaction is responsible for coordinating the insertion; actual insertion mechanic are implementation-specific (and hence, the sub-process is collapsed) to concrete write transactions. For variants, see Section 4.4. For additional information regarding the extensibility of the transaction hierarchy or* AbstractReflectableMaterial*, see Section 4.3.5 (Non-BPMN-compliant dataflows go across subprocess boundary to avoid clutter; the usability scoring activity has also been omitted).*



(b) *The* AbstractReadTransaction *defines the general process of retrieving and post-processing buffer content to produce a response – in this case, a reflection. As with writing, the actual retrieval and post-processing strategy is implementation-specific (and hence the subprocesses are collapsed). For concrete variants, see Section 4.4.*

This elevates reflections to **one possible** primary storage unit in our system. Conversely, we argue against the simple strategy of storing and retrieving mere raw thoughts in memory, as formally suggested by GoT (see Section 3.2.1.4).

To illustrate this, let us briefly return to the teacher-student analogy from Section 3.1.1 and consider how a teacher might help a student: Rather than dumping all similar experiences they've had, a good teacher synthesizes a specific, tailored piece of advice. Similarly, even if one retrieves an ideal prior thought (e.g., one containing a key mistake to avoid), research on Self-Reflection (Section 3.3.2.2) strongly suggests that the superior way to reuse it, may be by first extracting a reflection and *then* integrating it into the generation of the next thought.

Likewise, our decision is underscored by the idea that the core function of Self-Reflection, to extracting task-relevant insight from ones thought to inform the next, may closely mirror the purpose of a working memory system itself. [52]

[52] We also note the strong link between agents and reflection as suggestive of Self-Reflection's potential (see Section 3.3.2.2). Another speculative perspective is to view WM as a pure abstraction of one's environment. In that case, raw observations would not be stored, only metacognitive derivatives (i.e. reflections) about them. In this view, raw thoughts belong to the external reasoning system and should not be duplicated within WM.
However, the format of WM items may ultimately depend on their reusability and coherence. If certain environmental fragments (e.g., Thoughts) prove consistently useful, storing them directly might be justified and is hence supported by the core architecture, albeit us not considering this avenue further.

Accordingly, while permitting the integration of other approaches in our core architecture, we see little benefit in storing raw thoughts directly. Instead, we leverage the "chunking" ability of Self-Reflection (see Section 4.2.2) to produce more generalizable units of reasoning that can be stored in a coherent format.

With this core interaction established, we now examine the structure of the buffers and the role they play in supporting this process.

4.3.3 The AbstractBuffer and Forgetting

Building on the conceptual buffer hierarchy in our interpretation of the MCM (Figure 3.3.1.3), *AbstractBuffer* serves as the root class for all memory buffers in our architecture. We aimed to design it with maximal generality to accommodate the theoretical ambiguity surrounding WM in cognitive science (see Section 3.3.1).

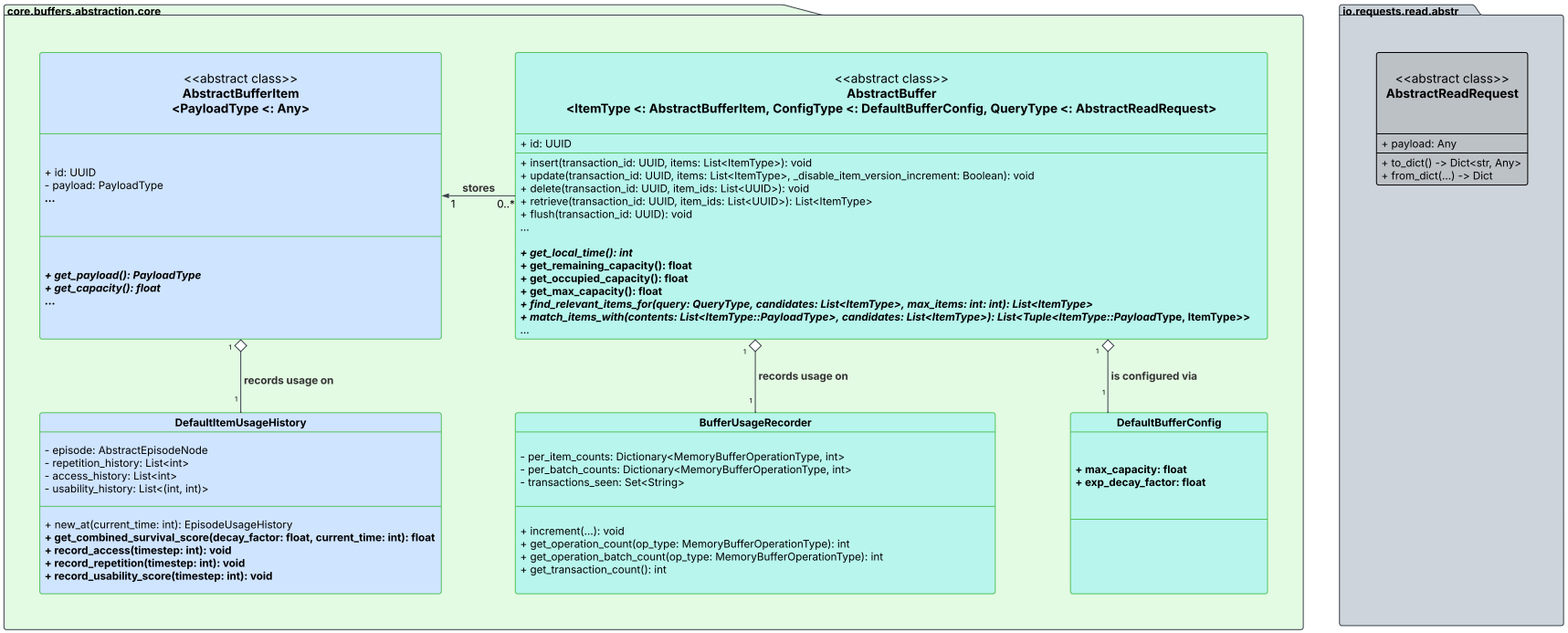
Following the synthesis in Section 4.2.2, we distill three essential properties that any such buffer must satisfy: (1) it must be passive, (2) it must have a capacity limit, and thus, (3) it must support a notion of forgetting. The first follows from the fact that the episodic buffer, as a subclass to-be, is defined to be passive; by the Liskov Substitution Principle, this constraint must therefore also apply to the root class.

Architectural Role The architectural implications of these principles are reflected in the core properties of the *AbstractBuffer*, as indicated in Figure 16:

- **Strictly passive**: It does not perform any operations autonomously, but is modified exclusively through external CRUD (Create, Retrieve, Update, Delete) operations.
- **Owns the item space**: It decides what specific type of *AbstractBufferItem* to accept, and dictates a minimal set of abstract functions that determine how items relate to a given type of request (*find_useful_items_for*), and new types of content to be inserted into the buffer (*match_items_with*).

Figure 16

A non-exhaustive class diagram illustrating main fields and methods of AbstractBuffer*, which defines a required ItemType (an* AbstractBufferItem*), a QueryType (an* AbstractReadRequest*), type of config; the associated* AbstractBufferItem*; respective extension objects.*
(1) AbstractBuffer is strictly passive*: It cannot execute operations autonomously (including those related to forgetting items), but only supports basic CRUD operations.*
(2) AbstractBuffer has limited capacity *and provides utilities to check the currently occupied, maximal, and remaining item capacity.*
(3) AbstractBuffer supports forgetting*, via an abstract notion of local time that can be used to calculate survival scores in an item's* DefaultItemUsageHistory*.*
(4) AbstractBuffer owns the item space*, by defining:*
[4.1]) what type of items to accept.
[4.2]) find_useful_items_for*: An abstract function that determines item relevance against a request.*
[4.3]) match_items_with*: An abstract function that determines whether a given item-payload (that is not yet part of an item) is semantically equivalent with a given item that is already stored. This can be used, e.g., to determine whether an item is about to be re-encountered in WM.*



- **Default query support**: Consequently, it must expose a corresponding type of *AbstractRead-Request* to accept per default.

Capacity and Forgetting We orient our implementation on a lazily enforced resourced-based view on capacity (see Section 4.2.2), while aiming to support various forgetting and capacity constraints, where *AbstractBuffer* further is:

- **Passive in respect to forgetting**, by delegating the evocation of forgetting operations to external transactions.
- **Enforces a hard capacity limit**, rejecting operations that would exceed the buffer's maximum capacity.[53]
- **Requires a notion of local time**, intended to be used for calculating time-based item survival scores. Per default, this is supported through an internal *UsageRecorder*, which monitors metrics such as the number of operations and seen transactions. However, the buffer merely provides its local time; handling it, e.g. to compute survival scores, is delegated to the environment.

Item Semantics In line with the above, the root class for all buffer items, *AbstractBufferItem*, provides:

- A *UsageHistory* to be used by transactions for tracking events and scores relative to buffer time, which may contribute to survival score computation.
- An abstract method that returns the item's capacity cost in a given application context, to be implemented by subclasses.

The base implementation of *UsageHistory* tracks two types of discrete events: *item access* and *item repetition*. These repent any access (read or write) to an item, and any time it is reproduced (e.g., when the same reflection occurs twice), respectively, simply recorded as numerical timestamp. Additionally, it includes a subjective *usability* score, recorded on a scale from 0 to 10, together with a numerical timestamp.

[53] This should not preclude forgetting strategies that interpret capacity differently. See Section 4.3.5 for further discussion.

We follow (Park et al., 2023) [54] by decaying recorded events and scores exponentially over time using a configurable decay factor $\lambda \in (0, 1)$, such that recent events contribute more strongly to the computed survival scores. Formally, the survival contribution of an event or score recorded at timestep $t \in \mathbb{R}$ with respect to current timestamp $T \in \mathbb{R}$ is weighted by λ^{T-t}.

The survival scores are computed as follows:

- The **access score** is the decayed sum of all recorded access timestamps t_i:

$$\text{AccessScore}(T) = \sum_{t_i \in \text{AccessHistory}} \lambda^{T-t_i}$$

- The **repetition score** is the decayed sum of all reproduction timestamps t_i:

$$\text{RepetitionScore}(T) = \sum_{t_i \in \text{RepetitionHistory}} \lambda^{T-t_i}$$

- The **usability score** is the decayed sum of all recorded usability timestamps t_i, each weighted by its respective score value s_i:

$$\text{UsabilityScore}(T) = \sum_{(t_i, s_i) \in \text{UsabilityHistory}} s_i \cdot \lambda^{T-t_i}$$

These components are combined into a **composite survival score** using the following default formula:

$$\text{CombinedScore}(T) = (0.5 \cdot \text{AccessScore}(T) + \text{RepetitionScore}(T)) \cdot (1 + \text{UsabilityScore}(T))$$

Notably, an item access and repetition is recorded immediately, when a *UsageHistory* is instantiated (via a new item), so that does not have a combined survival score of 0. Having established this class, we now continue with the *AbstractEpisodicBuffer*.

[54] See Section 4.2.2 for more context on forgetting mechanics at the intersection of LLM research and cognitive science. Also note that Park et al. (2023) use 0.995 as decay factor, which we also simply adopt as a default value.

4.3.4 The AbstractEpisodicBuffer and Chunks

The *AbstractEpisodicBuffer* is a subclass of *AbstractBuffer*, and thus inherits all traits connected to the strict capacity limit and passivity. As defined earlier, passivity refers to the inability of a buffer to autonomously execute operations, including those related to the forgetting mechanism.

Building on our synthesis in Section 4.2.2, we identify the following additional properties that characterize the *AbstractEpisodicBuffer*:

- **Passivity (inherited)**: The buffer remains entirely passive, consistent with the MCM (see Section 3.3.1.2).
- **Support for hierarchical memory items ("Chunks")**: The episodic buffer is assumed to store complex, structured information. We see the concepts of hierarchical "chunks" to provide a natural framing for such structures – both in cognitive science, where it applies broadly (e.g., to long-term memory; as seen in Section 3.3.1.2) [55], and in LLM research, where abstraction and information grouping can be used to "compress" content efficiently for reasoning tasks (as summarized in our synthesis, Section 4.2.2).
- **Episodicity in time and space**: The buffer must support an episodic organization of memory content (see also Section 3.3.1.2). This may imply that items could be grouped and retrievable according to both temporal and contextual coherence.

The first criteria is fulfilled by inheritance from *AbstractBuffer* (previous Subsection 4.3.3), so we continue with the remaining, in order:

The AbstractChunkNode To support hierarchical memory items, we introduce the *AbstractChunkNode* as the required memory item type for the *AbstractEpisodicBuffer* (see Figure 17).

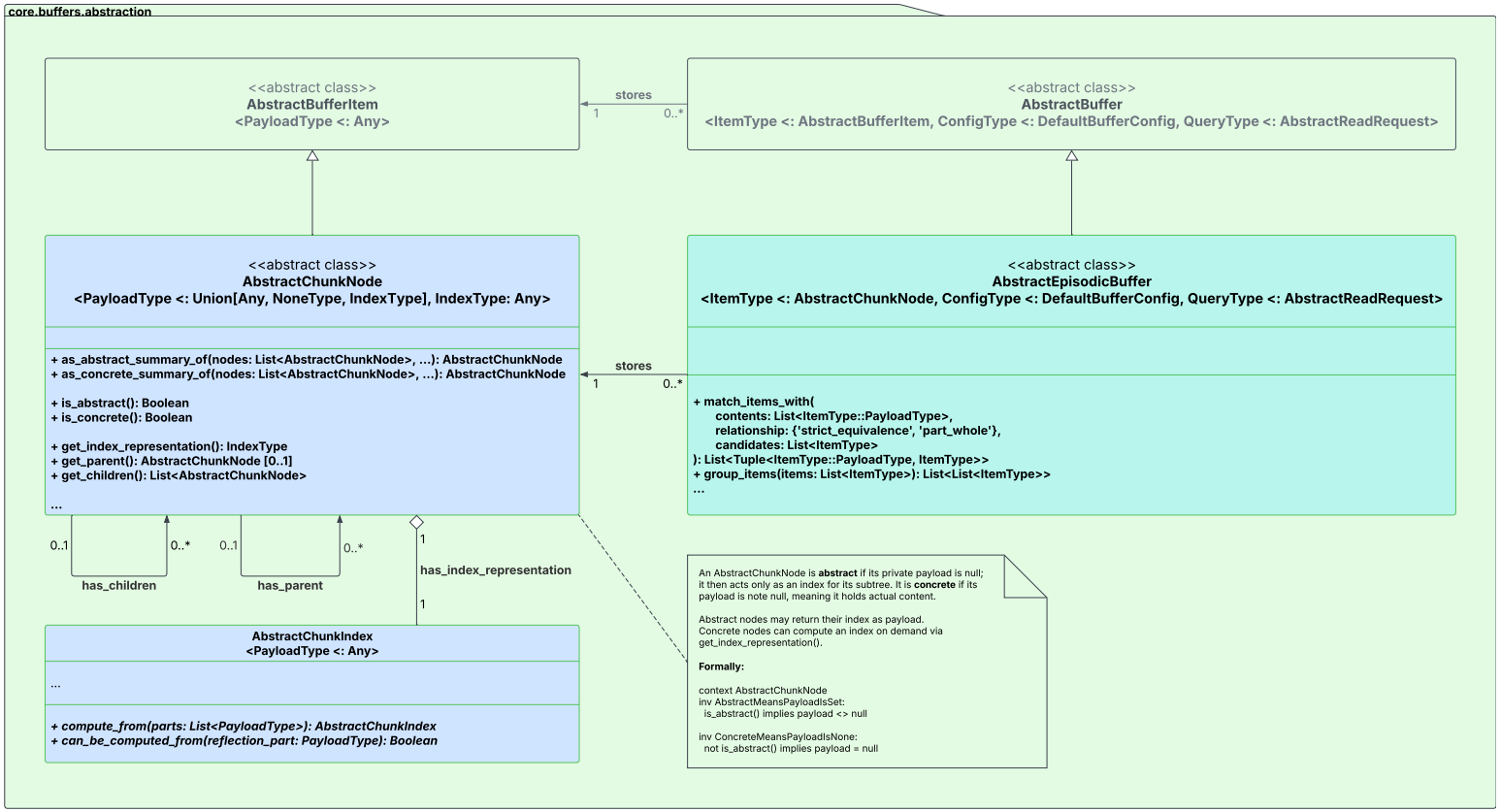
It represents a node in a classic tree data structure, having at most one parent [56] and arbitrarily many children.

[55] Note that therefore, while the episodic buffer may not be the only memory structured to support hierarchical chunks, evidently, there may be other memory components in the MCM that required fundamentally different "item types", such as the phonological loop that requires – supposedly linear – auditory traces. See Section 3.3.1.2.

[56] Which is arguably a limitation of our current design. For further discussion see Section 6.1

Figure 17

A non-exhaustive class diagram illustrating key methods and type relationships of the AbstractEpisodicBuffer *and its associated item type* AbstractChunkNode. *Nodes may be* abstract, *serving as cluster labels that compute index representations from their children,* concrete, *holding payload content, or* concrete summaries, *holding payload content that is a composite of other payloads. Factory methods* as_abstract_summary_of *and* as_concrete_summary_of *implement chunking akin to compression or composition, respectively. Finally,* AbstractEpisodicBuffer *extends item related functions to support the construction of trees, where*
(1) match_items_with *perviously only supported equivalence checks, is extended with an additional* part_whole *relationship, that may be used to determine wether a node should be part of a given tree ("top down" construction, i.e. insertion).*
(2) group_items *may be used to group a set of previously unrelated items for the formation of a new tree ("bottom up" construction).*



A node can have no payload (such as a reflection) at all, only to serve as a cluster label for its children – then it is considered *abstract*, and can compute an *index representation* of itself at runtime, based on its children. Otherwise, if a node has a payload, then it is considered *concrete*.
An abstract node has a relationship to its children where information loss is assumed (cluster labels are rarely as informative as the full set of items they refer to) whereas a concrete node is treated as an informational representative of its children.[57]

In this context, we define "chunking" as the creation of a new node that serves as the parent of a set of existing nodes. This can occur either through compression, resulting in a new abstract node via *as_abstract_summary*, or through composition, yielding a concrete node via *as_concrete_summary* (see Figure 17).

Specifically, in the default implementation:

- *as_abstract_summary*: Given a set of concrete nodes, this function creates a new abstract node and assigns the provided nodes as its children. The parent node holds a concrete subclass of *AbstractChunkIndex* (such as a textual label or a latent representation). This index is computed lazily upon first access and is automatically invalidated whenever the set of children changes,[58] triggering a re-computation on the next access. To compute the index, the payloads of all child nodes are aggregated into the desired summary representation (e.g., to generate a descriptive label from all reflections stored in the subtree).
- *as_concrete_summary*: Given a set of concrete nodes, this function creates a new concrete node that **merges** their payloads into a composite of the same type.[59] All input nodes are added as children of the new merged node. Additionally, the origin of each payload is recorded, allowing the system to determine at runtime whether a concrete node constitutes a *concrete summary* (that is, whether its payload originates from a merger of others) even if the tree structure changes later (or degenerates).

Since the buffer owns the item space, it continues to define critical functions that specify how items relate to one another. To support the creation of chunk trees, the *AbstractEpidosidcBuffer* introduces

[57] That is, we assume that a concrete node can always be retrieved to comprehensively represent its entire sub-tree as a surrogate in the task context.
[58] The invalidation mechanism and lazy re-computation logic are not shown in the class diagram.
[59] This merge function is not shown in any class diagram, as we deliberately omit the definition of abstract payload types (that own it) to reduce diagrammatic complexity.

an abstract grouping function, and extends the abstract function *match_items_with*, by requiring the implementation of an *equivalence* and *part_whole* relationship. The first describing the relationship between abstract nodes and their children, and the second the relationships between concrete nodes that should be merged (see Figure 17).

Episodicity Given the lack of a clear definition of what constitutes episodicness (see synthesis, Section 4.2.2) we opted to absorb this criterion into the higher-order functions of the buffer. Specifically, ordering items in connection to the task is handled via the abstract method introduced by the parent, *find_relevant_items_for* (see Figure 16), while grouping with respect to buffer-internal context is delegated to the abstract method *group_items* (see Figure 17).

While, admittedly, there may be additional benefit in exposing an explicit ordering or grouping mechanism that is context-independent (i.e. can house arbitrary context), we consider this a design limitation. For further notes, see Section 6.1.

This concludes our overview on the core architecture, apart from further notes we provide regarding (future) extensibility considerations, in the next Subsection 4.3.5. After this, we continue to showcase our concrete implementations, in Section 4.4.

4.3.5 Notes on Extensibility

In this subsection, we briefly collect all design decisions aimed at providing an extensible framework for implementing a LLM-centered computational model of the MCM. Because cognitive science remains undecided about the exact structure of WM, and since many complexities were left out of scope in our approach (as discussed in Subsection 4.2.1), we suggest that offering an extensible foundation that could ultimately support the full complexity of a human-oriented WM may be important. Accordingly, we divide the remainder of this subsection into two categories: Extensibility requirements that have already shaped our core architecture, and those that remain unmanifested (along with pointers to where such functionality may be mounted).

Finally however, upon reflection, we acknowledge some theoretical tensions in the design decisions that were made; these are discussed separately, in Section 6.1.

4.3.5.1 Extensibility Within Scope: Realized Architectural Decisions

As has been partially mentioned, the following extensibility points and structural variations are already embedded in the core architecture:

- **Embedded support for the co-existence of multiple buffers of different types**: Buffers defensively verify their item types, and transactions are designed to operate over a *set* of multiple target buffers (currently implemented via a uniform broadcast strategy). This becomes relevant when considering multiple (additional) types of buffers for a single client or multi-client scenarios in which buffers could be shared. The co-existence of multiple buffers ultimately introduces a design socket for consistency support, enabling tracking of buffer states over time. We consider this useful for data mining[60] and analytical purposes.
- **Support for varying capacity demands**: In line with findings from cognitive science (Section 3.3.1.1), buffer items are responsible for declaring the capacity they occupy.
- **Support for multiple forgetting strategies**: To keep buffers as passive as possible, we delegate the responsibility for executing forgetting operations to transactions, and base our mechanism on a lazily enforced, resource-based view of capacity (see Section 4.2.2). The underlying assumption is that sufficient WM capacity must support stable retrieval (as opposed

[60] For instance, when treating the buffer as a source of training data, similar to methods like Self-Distillation (Section 3.1.1), or specifically, in light of the idea of training a model with reflections, as explored by An et al. (2024).

to effects like the "lost-in-the-middle" problem (Liu et al., 2023)) and that retrieval mechanisms should be intelligent enough to distinguish useful from non-useful content. Hence, we see little reason to remove elements from the buffer as long as capacity remains available. Still, we aim to support more complex forgetting strategies. For example, decay-based forgetting could treat capacity as a temporal threshold, beyond which item must be forgotten, and enforce clean-up via auxiliary checks (e.g., via the implemented hooks on transaction begin or before item retrieval) or rejecting further operation otherwise. Depending on the approach, some forgetting mechanisms may require additional internal logic within the buffer; however, in our view, this must not compromise its strict passivity.

- **A glimpse of support for hierarchical transactions**: While we currently refrain from implementing contentions scheduling with hierarchical transactions, we implement a straight forward transaction hierarchy, where each transaction must declare its expected input and output types as well as the types of buffers it operates on. Though currently used only for defensive validation, this mechanism lays a minimal foundation for dynamically scheduling transactions, where each caller knows what to expect.
- **Support for arbitrary thought and reasoning granularity**: Our root process (Figure 15) makes minimal assumptions, so that we do not enforce any fixed reasoning structure. This is left to the client (i.e., facade). Our root transaction class only expects an *AbstractRequest* as input and an *AbstractResponse* as output. A request may thus represent a single thought or span a full reasoning subgraph, allowing to flexibly adapt to varying scopes and reasoning granularity.
- **Support for arbitrary reflection granularity**: This design is propagated to the high-level reflection process (Figures 15b and 15b), where we define *AbstractReflectableMaterial* to represent any type of content that can be reflected upon – in a broader sense, that may go beyond the idea of Self-Reflection (introduced in Section 3.3.2.2). A reflection may thus range from a single thought to a CoT or an entire reasoning subgraph.

4.3.5.2 Extensibility Beyond Scope: Deferred and Speculative Components

As established in Section 4.2.1, several components of the MCM were deemed outside our design scope. However, we aimed to make our architecture extensible enough to support future integration of such elements, and we currently foresee the following mounting points:

- **Support for hierarchical transactions and contention scheduling**: The Schemata package is intended to conceptually align with "action schemata" in cognitive science and may be extended to support cue-based dynamic scheduling or hierarchical orchestration, to bring it closer to contention scheduling (see Section 4.2.1). This could involve meta-transactions that enable hierarchical transaction dispatch based on environmental cues. Such scheduling may become essential, as contention scheduling is arguably the central dependence of the central executive, and thus forms the basis for complex metacognitive reasoning.
- **Support for Long-Term Memory at every stage**: As discussed, cognitive science remains uncertain about the role of long-term memory (LTM) in WM, though it is highly plausible that LTM could be involved "at every stage" (Baddeley, 2010). We chose to treat the LLM as the LTM, while acknowledging that this "homuncular" shortcut may be suboptimal if reliable storage media are required. Despite leaving a dedicated LTM integration out of scope, we offer guidance on what it might look like within our architecture, if chosen to be modeled *explicitly as separate storage instead*.

  To reiterate from our interpretation of the MCM (Section 3.3.1.3), we see two primary representations suited for LTM storage: action schemata (i.e., transaction templates) and buffer item representations. Broadly, we envision LTM interfacing with our system along the following horizontal layers:

  - **Layer 3: Facades** – LTM may enhance read/write requests to improve retrieval or payload construction.
  - **Layer 2: Schemata** – LTM may guide contention scheduling and flow control by providing appropriate schemata at the right moment, providing data essential for conflict resolution mechanisms to be enacted by a central executive.
  - **Layer 1: Buffers** – LTM may exchanges information directly with buffers. For buffers, this could increase its flexibility to adapt to new environments, e.g. by pre-loading prior knowledge or contextualizing new inputs. For LTM, this could imply a learning mechanism that consolidates successful (and context-stable) WM representations, akin to what Burgess and Hitch (2005) propose (see Section 3.3.1.2).[61]
  - **Layer 0: Storage** – Remains unchanged as a passive backend. However, specialized LTM storage backends could be introduced to manifest some of the mentioned capabilities. For example, a "Transaction Template Database" that exposes retrieval functions

[61] We also note that especially this layer may introduce additional conceptual tension or complexity when integrating a LTM. See also our discussion of limitations regarding a dedicated search in Section 6.2.2.

via environmental cues (cf. the trigger database proposed by Norman and Shallice (1986), see action schemata in Section 3.3.1.2).

This concludes our discussion on extensibility.

4.4 Implementation Variants

In this Section, we build on our core architecture (previous Section 4.3) to introduce concrete implementation variants of a working memory (WM) system for large language models (LLMs), which we evaluate in the following Chapter 5.

We first establish three core hypothesis to test and set up the design space into four possible variants (Subsection 4.4.1). This brings us directly to an overview of which components must be implemented and how they are shared across variants (Subsection 4.4.2), followed by a detailed presentation of the resulting groupings: The memory item (chunk; Subsection 4.4.3); a reflection they can store as payload, as well as an index representation (Subsection 4.4.4); the concrete transactions that read and write reflections (Subsection 4.4.5) from and to specific buffer variants (Subsection 4.4.6).

4.4.1 Hypothesis and Design Space Setup

To reiterate:

- In line with our main research question, our primary goal is to contribute a mechanism for flexibly retrieving relevant material from growing reasoning graphs (see Section 4.1), specifically in the context of Graph-of-Thought (GoT).
- We are particularly interested in the idea of *chunking* (see Section 4.2.2 for a practical overview), and established the ability of an (episodic) buffer to store a forest of memory items (see Section 4.3.4).

In addition, we are interested in exploring the discrepancy between the employment of a LLM-only, and embedding-based (or hybrid) storage maintainer, since:

- RAG-style memory architectures are commonly used for memory in reasoning (see Section 3.3).
- In turn, a buffer building a forest of memory items may benefit from embedding-based methods as a superior (compression-based) chunking strategy (again, cf. Section Section 4.2.2) that enables more flexible clustering.

- Overall, RAG systems excel at managing retrieval over large text corpora, whereas LLMs tend to exhibit increased retrieval errors as context length grows (cf. Y. Gao et al. (2024) and Liu et al. (2023), respectively). Conversely, given a small enough WM capacity, one could assume the LLM to be the more flexible retriever.

Accordingly, our implementation variants are designed to support different test scenarios for investigating the following **working hypotheses**:

Hypothesis I: Introducing a working memory (WM) as an advanced reflection store (cf. Section 4.3.2) improves the performance of GoT.

Hypothesis II: A WM utilizing chunking strategies (i.e., one that builds a forest of memory items) should outperform a flat memory setup (i.e., one that uses a simple set of unstructured items).

Hypothesis III: For a WM with limited capacity, a LLM should serve as a superior model for chunking and retrieval compared to embedding-based approaches.

While the first point is trivial in our context, the latter two imply a natural 2×2 design space: (1) systems that employ chunking versus those that do not, crossed with (2) systems that employ an embedding model vs. those that do not.

Since thus far in this thesis, we focused on LLM-only variants, we see the embedding based counterpart as a competitor, and opt for a strong baseline that implements a hybrid approach: A-MEM, introduced by W. Xu et al. (2025) in Section 3.3.2.1 currently considered a state-of-the-art memory for agents. This serves a dual purpose: Even if our system fails entirely in the context of GoT, we still gain the opportunity to benchmark against a top-performing memory system.

Finally, due to resource constraints and to limit complexity, we once again limit the scope of our implementation, in light with our minimal viable design focus:

- We restrict chunking to trees with at most one abstract node, akin to a flat (and not hierarchical) clustering, comparable to Hi-Agent (Hu et al., 2024).
- We do not perform re-organization (i.e., re-clustering) of trees within the memory forest.
- We do not consider reflectable material (see Section 4.3.2) beyond the context of one single thought to be generated.

Again, this aligns well with our non-chunked baseline (since there is nothing to re-organize in a flat set). Additionally, this is in favor of A-MEM based variants, which *does* re-organize its clusters periodically.

4.4.2 Overview

Following the setup of the prior Subsection, consequently, our implemented variants fall into four categories, as shown in Table 1.

It follows from the high-level reflection process (see Figure 15a and 15b) that each variant must come with:

- A concrete read transaction.
- A concrete write transaction.
- A concrete subclass of `AbstractReflectableMaterial` that can produce a corresponding reflection in the context of a single GoT thought (defines GoT interface; shared across variants).

In the context of a concrete implementation of the episodic buffer itself (Figure 4.3.4), it must define

- The specifics of the memory item (shared across all variants), which entail:
  - A concrete subclass of `AbstractChunkNode`.
  - A corresponding subclass of `AbstractChunkIndex`.
  - A definition of item capacity.
- The implementation of inherited abstract functions (apart from CRUD operations; shared across all variants that have the same LLM-only, v.s. embedding-hybrid backbone):
  - `find_useful_items_for` a given query and a set of candidates
  - `match_items`, regarding part-whole and equivalence relationships.
  - `group_items`.

Differences are summarized in Table 1. In summary:

- All variants share the same interface to GoT, memory item definition, and reflection specifications.
- All variants that are using the same chunking strategy (trees v.s. flat) also share the same write transaction.
- All variants share the same read transactions, since reading from a forest subsumes reading from a flat item set (i.e. a maximally shallow forest).
- All variants that are either embedding-hybrid (A-MEM based), or LLM-only, share the same underlying buffer implementation, irregardless of the chunking strategy employed on their content (again, since a flat set of memory items can be considered a maximally shallow forest, which must be considered by a buffer that *can* store a forest, even if it does not really do so in practice).

Table 1
Design matrix of implemented WM variants. Rows vary by indexing method – specifically the implementation of higher order buffer-methods, cf. Section 4.3.4; columns by chunking strategy (write transaction). All variants share the same read transaction (since reading from a forest inherently subsumes reading from flat item set), memory item, and reflection specifics (not depicted).

Buffer Implementation	*Write Transaction Implementation*	
	No Chunking	*Chunking*
LLM-Only (Ours)	***LLM-Based (Flat)***	***LLM-Based (Trees)***
Embedding-Hybrid (Baseline)	***A-MEM-Based (Flat)***	***A-MEM-Based (Trees)***

This leads us to the remaining Subsections, which detail the respective implementations: The chunk (memory item) design including its lightweight index and definition of capacity (Subsection 4.4.3), its concrete payload – a reflection (Subsection 4.4.4), the transaction logic (Subsection 4.4.5), and finally, the buffer implementations (Subsection 4.4.6).

4.4.3 Memory Item (Chunk)

This subsection builds on the item-level details introduced in Section 4.3.4, to outline our memory item subclass implementation: A tree of chunk nodes with defined capacity. The here outlined format is shared across all variants, but expands beyond a single concrete node only in chunking-enabled variants.
While we aim to keep the discussion focused on the design of individual items, some overlap with the later discussion of the writing transaction (Section 4.4.5) is unavoidable, as the handling of individual nodes is inherently tied to their potential tree structure.

Payload, Index, and Survival Scores To this end, we implement a lightweight subclass of *AbstractChunkNode* that houses a specific type of reflection (specified in detail, in the following Subsection 4.4.4) and utilizes all currently implemented survival scores (access, repetition, usability).
This is paired with a concrete implementation of *AbstractChunkIndex* that merely holds a simple textual descriptor of reflections within its sub-tree.[62] The respective prompt for summarizing a set of reflections into such a short descriptor can be found in Appendix A.3.

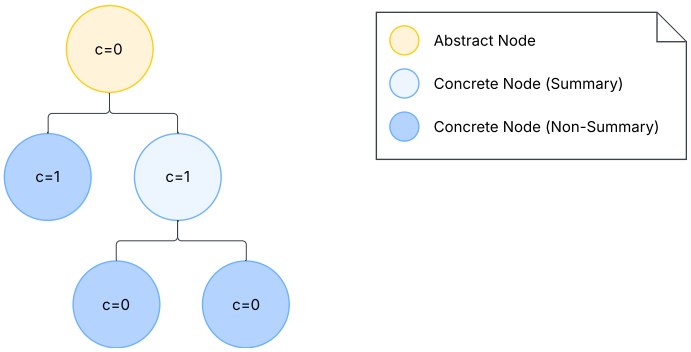
Chunk Design Adhering to our goal of limiting complexity by refraining from building hierarchical clusters, we illustrate our chunk design, as depicted in Figure 18:

- **Abstract nodes** act as cluster labels and are restricted to having only concrete children.
- **Concrete nodes** carry reflection payloads, distinguished further into:
  - *Concrete Non-Summary nodes*: Concrete nodes that contain reflections derived directly from a single thought.
  - *Concrete Summary nodes*: Concrete nodes that contain reflections composed from all reflections of their child nodes.

Concrete summary nodes serve a critical role: They handle repeated insertions of reflections that closely resemble existing ones.

[62] Note that while we would also be able to implement a dedicated embedding-bearing variant for the embedding-hybrid (A-MEM based) variants, for practical purposes, we opted for one that merely stores a LLM-computed summary of its children. Still, this summary is embedded into, and retrieved from A-MEM, as we will see.

Figure 18
Chunk design in all of our implementation variants. Expands beyond a single concrete (non-summary) node only in chunking-enabled variants. Abstract nodes have a part-whole relationship to all of their children; concrete summary nodes have an equivalence relationship to their children, but not to their siblings. The size of the concrete front determines the tree capacity – for each node, the capacity c is 1 if it is concrete and has no concrete parent, and 0 otherwise.



This first occurs when a new reflection a matches the reflection b stored in an existing concrete node, under the buffer's equivalence relation.[63] In this case, we want b's representation to be enriched by a, which, although judged to be conceptually identical, may still bear meaningful variations. Hence, we prompt the LLM to merge a and b into a payload for a newly created concrete summary node [64]. However, consider the same process being repeated multiple times. Here, we suspect that if we do not hold on to the original representations (i.e. only ever merge two payloads, the existing one and the "repetition") we risk semantic drift of the more distant reflection payload(s).[65] Consequently, we hold onto the original reflections and insert them as children of the concrete summary node. Once a new node is found to be equivalent with the summary node, it will be inserted as child, and the summary node will be recomputed as a merge of all the payloads of its children. We further detail this process in our transaction implementation, in Section 4.4.5.

Capacity Since the main purpose of a concrete summary node's children lies in preventing semantic drift, we decide not consider them to contribute to the buffer capacity. Likewise, we decide that an abstract node should not contribute to the capacity. Therefore, we define **the capacity c of a node as $c = 1$ if it is concrete and has no concrete parent, and $c = 0$ otherwise.**

[63] Compare in Section 4.3.4, `match_items`, also in Figure 17,

[64] Strictly speaking, this is a responsibility of the reflection implementation, as it depends on their format; details of which are presented in the following Subsection 4.4.4. Please find the respective prompt in Appendix A.3

[65] That is, if we were to discard the original reflections and retain only the summary node, which repeatedly absorbs new reflection payloads from nodes $n_0 \ldots n_k$, earlier content would increasingly be overwritten. With each summarization, the influence of earlier inputs diminishes, causing their meaning to shift or fade. This phenomenon is also known as semantic drift.

This brings us to the next Subsection, in which we detail the construction of the item payload, the reflection. To continue directly with the full construction process of chunk nodes (the writing transaction), see Subsection 4.4.5.

4.4.4 Memory Item Payload (Reflection)

We introduced reflections in Section 3.3.2.2. Although the core architecture permits omitting reflections (see Section 4.3.2), they are essential in all implemented variants, where they constitute the item payload:

- Their generation occupies the first activity in the `AbstractWriteReflectionTransaction` (cf. Figure 15a).
- They are then kept in memory, and can be merged with one another (see previous Subsection).
- They can be preprocessed at retrieval time in the `AbstractReadTransaction` (second activity in Figure 15b; this step is illustrated in the read transaction itself, Section 4.4.5), and finally
- They can be incorporated into the generation of a next thought, by GoT (cf. top lane, third activity in Figure 14).

We proceed by outlining our (1) reflection format, (2) its generation, and (3) incorporation within GoT. Therein, we will reflect on best practices of Self-Reflection, as formerly established at the end of Section 3.3.2.2.[66]

[66] Note that the grouping of best practices in Section 3.3.2.2 are organized according to reflective capabilities more broadly. Here, we restructure them to align with the concrete three-step process used in our implementation.

Reflection Format We were mainly inspired by the structure used by An et al. (2023) and funneled respective best practices into the following format:

Example Reflection Item

```
<Reflection Item>
    <Error Location>
        "If A is taller than B, then B is taller than A."
    </Error Location>
    <Error Explanation>
        This step reverses the comparison logic;
        the conclusion contradicts the premise.
    </Error Explanation>
    <Error Fixing Suggestion>
        Keep direction consistent: If A > B, then B < A.
    </Error Fixing Suggestion>
    <Takeaway>
        Maintain logical direction when chaining.
    </Takeaway>
    <Scores>
        <Repetition Score> 0.21 </Repetition Score>
        <Access Score> 0.37 </Access Score>
        <Usability Score> 0.64 </Usability Score>
        <Combined Score> 0.58 </Combined Score>
    </Scores>
</Reflection Item>
```

However, this format is partially a result of a compromise with our scope (Section 4.1). Attending to the respective best practices, we can see that **our reflection is "actionable", and "specific"** – that is, it contains a concrete and actionable `<Error Fixing Suggestion>` and identifies the concrete `<Error Location>` of a reasoning steps to avoid (practice established by Madaan et al. (2023)). However **our contained feedback is limited in its diversity**: As a consequence of our goal to remain task-agnostic, we refrain from using targeted feedback scores that could account for trade-offs between quality dimensions, where improving one task-specific aspect may lead to the degradation of another (as noted by Madaan et al. (2023)). We attempt to counteract this notion with

the integration of the item survival <Scores> (established Section 4.3.3). For further discussion of this limitation, see Section 6.2.1.1.

Reflection Generation As per our scope, we generate a reflection in the context of a single prior thought (which, notably, may also already contain a reflection). Thereby, our approach falls into the category of "internal" error signaling, since the LLM itself will have to identify its error.
Given a thought, we prompt the LLM in a single generation to produce *multiple* yet *distinct* reflections of the above format.[67] Herein, we also instruct the LLM to **combine multiple reflection strategies** (e.g. keywords, advice, explanation, etc.), which has been established to be superior by Renze and Guven (2024). However, we leave it up to the model to decide which variants to use. The prompt can be found in Appendix A.3.

Reflection Incorporation This step is specific to the reading transaction (Subsection 4.4.5) and the following integration within GoT. We assert the remaining best practices:

- Our incorporation aims to strike a balance in the **number of reflections included during thought generation**. As previously discussed, this remains an open question. To mitigate retrieval issues associated with longer contexts (Liu et al., 2023), we deliberately introduce a summarization step.[68] This step distills up to 8 reflection from the set retrieved from memory (second activity in Figure 15b).
- Our incorporation encourages **multiple reflection iterations** (cf. Shinn et al. (2023)) to a limited extend: Since a previously used reflection is automatically contained in the next thought to be reflected on, the number of reflection steps therefore ultimately depends on the reasoning structure employed within GoT.
- Our incorporation is not reluctant (**reflecting more often is potentially better**, cf. Madaan et al. (2023)), since we incorporate reflection after every GoT thought [69], i.e. every write operation.

[67] The idea is to have a separation of insights attending distinct types of error, that can be treated as separate entities in storage.
[68] As outlined in the following subsection 4.4.5. Find the prompt in Appendix A.3
[69] Strictly speaking: Every thought that originates from the LLM. Some GoT thoughts can also rely on hand written implementations; e.g., scoring thoughts with a predefined evaluation function.

However, in accordance with our scope, restricting the reflectable material to the local thought foreshadows another compromise: When incorporating the reflection into a new thought to be generated, we do not incorporate the original thought that was **previously reflected upon**, which Shinn et al. (2023) found to be more effective than providing the reflection alone. This might also have been mitigated by expanding the range of error-localization. We revisit this design choice as a limitation in Section 6.2.1.1.

This concludes our reflection design. We continue by outlining our transactions.

4.4.5 Transactions

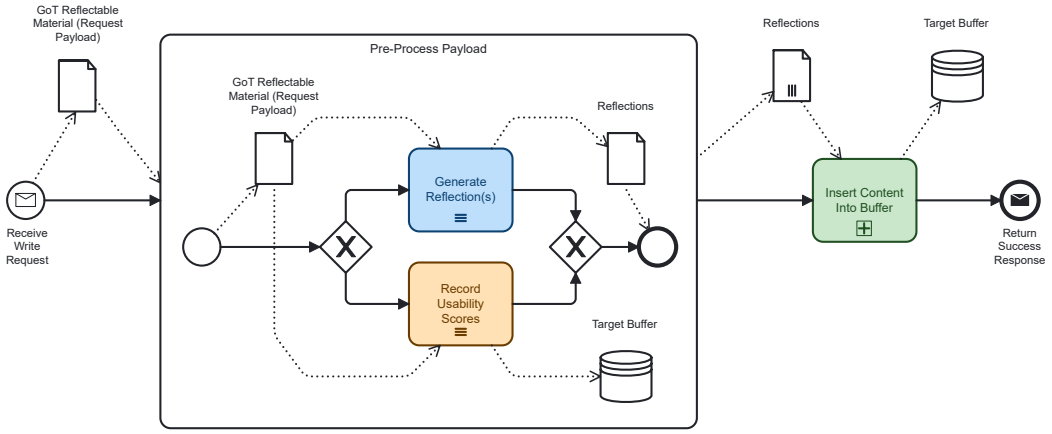
In this subsection, we detail the transactions used within our implementation variants. This builds on the abstract transaction interfaces introduced earlier (cf. Section 4.3.2, Figures 14, and 15 – the first being entirely agnostic, the latter reflection-specific), which we now make concrete for the respective implementation variants.

As outlined in our 2×2 design matrix (Table 1), the non-chunking setup and the chunking setup each require their own write transactions, while the reading transaction is shared. Since however, the writing transaction for the flat variant is almost trivial, we will introduce both write transactions together, and note the differences accordingly.

The Writing Transactions Our writing transactions sub-classes the `AbstractWriteReflectionTransaction` illustrated in Figure 15, which already takes care of the reflection generation (previous Subsection 4.4.4) in the preprocessing step. We now outline the remaining capabilities that make our implemented transaction concrete: (1) recording the usability of reflections stored in already existing nodes, (2) inserting the newly produced reflections as new nodes, and within the insertion: (3) recording access and repetition usage events,[70] and (4) handling the forgetting mechanism.
We outline these steps in the following. All variants only differ in how they handle the insertion step, which we describe separately.

[70] Usage events and scores (e.g., access, repetition, usability) are used for survival score computation, as outlined in Section 4.3.3.

Figure 19
High-level BPMN diagram of the implemented write transaction for all variants. The request payload contains a GoT Thought to be reflected upon, which may also include a reflection from a previous iteration. If still present in memory, such a reflection is evaluated for usability (orange activity). The Thought is reflected upon (blue activity), potentially yielding multiple distinct reflections. These are inserted individually (green collapsed subprocess), where variants differ. Note: Although theoretically parallelizable, blue and orange activities are executed sequentially in our implementation. The optional nature of usability scoring (to only score when there is an item to score) is handled within the activity itself.



Usage Scoring (Figure 19, orange activity)

Usage Scoring is shared across all variants. It requires a given thought to contain a reflection that is still stored within the buffer. In this case, this reflection is evaluated in the context of the thought that employed it (akin to how ToT and GoT rely on self-evaluated scoring)[71], to determine (on a scale of 1-10) how much the reflection has contributed to the reasoning output. The prompt can be found in Appendix A.3.

Insertion for Non-Chunking Variants (Figure 19, green activity)

Given a new set of reflections, the flat variants simply operate by (1) creating a new concrete summary node for each, and (2) inserting each such node as a new root in the forest. Importantly, the flat variants do not construct any concrete summary, or abstract nodes. They do also not record any access or repetition scores on already existing items.

[71] Cf. 3.2.1.3 and 3.2.1.4 respectively. This may also be considered a kind of self-reflection. However, we disregard this minor point for simplicity, and stick closer to the ToT and GoT narrative.

Insertion for Chunking-Enabled Variants (Figure 19, green activity)

Given a new set of reflections, the chunking enabled variants also process them into a corresponding set of new nodes. Inserting such a node into a tree (or chunk) is naturally closely aligned with the intended tree structure (outlined in Section 4.4.3). Building on this design, we give an overview of how, and under which assumptions the insertion algorithm operates.

To reiterate: Our scope confines us to **(1) only build trees with at most one abstract parent** and **(2) not to reorganize the clustering (the forest)**.

In accordance with the latter, the **insertion procedure assumes a perfect clustering to be present**. This includes, specifically, the assumptions that:

- Existing reflection-bearing (concrete) nodes with nearly identical reflections have successfully been consolidated into concrete summary nodes. That is, for a new node, we assume at most one equivalence match to exist within the concrete front at insertion time. [72]
- Existing concrete roots that belong to the same group have already received an abstract parent node.

It follows, that during the insertion of a new reflection-bearing node n_r, relationships (declared by the respective buffer implementation, as seen in the next Section 4.4.6) to any already inserted item n_i, may arise with the following exclusivity constraints:

- **Equivalence**: Either to exactly one concrete root node, one concrete child of an abstract root node (be it a summary or not), or none.
- **Part-Whole**: Either to exactly one abstract root node, or none.
- **Grouping**: To one or more concrete root nodes that are currently sole or disconnected members of their group. We deliberately relax our previous assumption here to allow the new node to serve as a potential missing link.

[72] **Formally**: Let $\mathcal{F}$ denote the set of chunk-nodes of the current forest stored in the buffer, and let $C \subseteq \mathcal{F}$ denote the *concrete front*, defined as the set of all concrete nodes in $\mathcal{F}$ that have no concrete parent:

$$C := \left\{ n \in \mathcal{F} \,\middle|\, n \text{ is concrete} \wedge \left(\nexists\, p \in \mathcal{F} : p = \text{parent}(n) \wedge p \text{ is concrete}\right) \right\}.$$

This is the set containing all concrete nodes that are not conceptually equivalent with any of their siblings. Further, let $\equiv$ denote the buffer's equivalence relation over reflection-bearing nodes. Then for any new concrete node r, we assume:

$$|\{n \in C \mid n \equiv r\}| \leq 1.$$

The insertion algorithm operates in four phases, to insert the new node n_r:

Phase 1: Sibling Search

We conduct a breath-first search over the entire forest, to find potential candidate siblings or duplicates of n_r. These are all nodes satisfying an equivalence relationships with n_r, or whose parent satisfies a part-whole relationship with n_r, respectively. To this end, we maintain the front F of potential candidates. F will be an ordered list of sets $f_0 ... f_n$, where all nodes in a set f_i are also siblings of one another.

- We initialize F with the set of all concrete root nodes R, i.e. $f_0 = R$.[73]
- We expand F, by assessing trees with abstract root nodes. For any abstract root n_a:
  - We check for a part-whole relationship of n_r to n_a.
  - If the relationship is given, we know that n_r should be inserted into the tree of n_a. As such, any child of n_a becomes a candidate sibling or duplicate of n_r.
  - Accordingly, we add the set of all children of n_a to F and do not check any subsequent abstract root.
- At this stage F represents the full front of concrete nodes that may be either siblings or duplicates of n_r.

Phase 2: Merging or Insertion

Supposing a part-whole relationship was detected in the prior phase, we attempt to insert n_r, into an existing tree. To do so, we must first check wether any duplicate of n_r exists that should be merged with n_r. Therefore, we proceed by iterating over each set $f_i \in F$ in-order:

- We asses wether a equivalence relationship between any $n_e \in f_i$ and n_r is given.
- If this is the case, we commit to the first match (since we assume at most one) and replace n_e with a concrete summary node n_s that is the merger of n_r and n_e.[74] As a result, n_s now bears the children of n_e (if any), and additionally, n_r.
- Otherwise, we assume that n_r is unique among all candidates in f_i, and may be inserted as a sibling node:

[73] An exception: These nodes are not direct siblings of one another, but all are candidates to become siblings of n_r.

[74] To reiterate: We use `as_concrete_summary` (cf. 4.3.4) to create a new summary node as the composition of n_e's children (if any), n_r, and n_e itself, if it is not a summary node (i.e. has no children). This replacement inherits the usage history of its predecessor. See chunk design in Subsection 6.1)

  - **Edge case**: If f_i is the set of concrete roots (i.e. $i = 0$), we defer inserting n_e as a new root, to account for a potential part-whole relationship in a $f_j \in F$ with $j > i$.
  - Otherwise, f_i must all be children of the same abstract parent n_a, and n_r should be inserted as a child of n_a. In this case, we do not check for other sets in F, since we continue to assume a part-whole relationship to be unique, and we have already checked equivalence with all concrete root nodes (since $i > 0$).
- At this stage, if n_r satisfies an equivalence relationship with a concrete root, it will be merged into that root. Otherwise, if a part-whole relationship is found with an abstract node n_a, n_r will either be inserted as a child of n_a, or merged with an existing child of n_a to which it is equivalent.

Phase 3: Grouping

If (and only if) Phase 2 detects no equivalence or part-whole relationship with n_r (i.e., yielding no merging and no insertion), we search for groupings among existing concrete root nodes. For each valid grouping of size ≥ 1, a new abstract parent is assigned to all involved members. Groupings of size 1 remain unchanged.

Phase 4: Fallback Insertion

If (and only if) Phase 2 yields no merging and no insertion, and Phase 3 also yields no grouping to assign n_r to, n_r is added to the forest as a new standalone concrete root node (leading to resolve the edge case above).

Accordingly, this strategy follows a strict priority: First, it searches for equivalent nodes; if none are found, it attempts insertion into existing trees; only as a last resort does it form new ones.
This prioritization of existing trees of course, risks early misgroupings that (as per our scope) cannot be corrected later, making it prone to over-aggregation.
However, we preferred this strategy over the alternative to risking multiple smaller but fragmented trees,[75] since our working hypothesis is that a sufficiently capable relationship attributor could yield usable, well-separated clusters early on. We consider this trade-off acceptable within our current scope and asses our design choice in a qualitative evaluation in Section 5.3.3.

Finally, we note that, in practice, the full set of nodes to be inserted is evaluated jointly against all existing candidate nodes (in order to reduce the number of LLM calls in LLM-based variants). Since

[75] For further discussion on the nature of this limitation, see Section 6.2.1.2

this optimization does not affect our theoretical assumptions, we have abstracted it away for clarity of presentation. Specifically, we do not compare each new node individually with each abstract node or with each candidate in a set $f_j \in F$, but instead evaluate all still remaining nodes against all abstract nodes or all nodes in a set f_j, respectively, in a single pass. For both equivalence and part-whole relationships, at most one matching candidate is allowed per new node (per call), thought the same candidate may be matched to multiple new nodes.

Access and Repetition Events In the above algorithm (for the chunking variants), when checking a relationship between the new node n_r and an already inserted node n_i, we record:

- **Access Events** on n_i, every time a relationship between n_r and n_i is checked. This is propagated upwards to all nodes on the path to n_i's root, and downwards to all nodes of its sub-tree.
- **Repetition Events** on n_i, every time an equivalence relationship between n_r and n_i is detected. This is propagated downwards to all nodes of n_i's sub-tree (i.e. the children of a concrete summary node), while ensuring, that the replacing summary node inherits the according usage history correctly from the replaced node.

Forgetting When committing its changes to the buffer, any write transaction first attempts to insert all items into the buffer. If buffer capacity is insufficient, the buffer will raise an `OutOfCapacityError` that details the additionally required amount of capacity. In this case, the writing transaction then ranks all existing items in the buffer by their combined survival score (at the current buffer time, cf. Section 4.3.3), and removes as many as necessary to free the required capacity, starting with those that have the lowest survival scores. This process also ensures that the tree structure remains intact. [76]

The Reading Transaction The reading transaction is the same for all variants. In parallel to the above, our reading transaction sub-classes the `AbstractReadReflectionTransaction` illustrated in Figure 15, which defines two sequential activities: First, find relevant items to retrieve, then, post-process them into the final response payload. These two steps are outlined as follows

[76] For instance, it prevents the creation of orphans or abstract parents without children. Structural validity is additionally verified by the buffer upon insertion.

Item Retrieval (Figure 15, first activity)

1. The transaction defines the maximal set of retrievable items:

   This includes all root nodes from the buffer's forest, expanded up to the content front. In other words, all nodes are eligible for retrieval except those that are children of a concrete summary node (cf. Figure18). This is reflective of the fact that, in our implementation, summary nodes act as representatives of their sub-tree. Hence we do not consider their children.
2. The transaction filters the retrievable set:

   To do so, it hands the set to the buffer to `find_useful_items_for` the given request payload (which acts as query).[77] The result defines the set of items to be handed to the post-processing step.

Item Post-Processing (Figure 15, second activity)

Postprocessing consist of a final compression step:

This is a re-summarization of the retrieved items, i.e. sub-trees in the forest. Herewith, we follow a dual objective:

1. We summarize the retrieved subtrees in the context of the given GoT-prompt (the request sent to WM), to enable a targeted contextualization of the contained reflections.
2. We constrain the number of resulting reflection trees to a maximum size of 8, to obtain a sufficiently dense set of actionable reflections.

Finally, the LLM is asked to map the originally retrieved reflections to the respective summaries, as to record an access event on them.[78] The respective prompt can be found in Appendix A.3.

This concludes our transaction implementation. Let us now finalize this Section, by introducing the underlying buffer variants in the following Subsection 4.4.6.

[77] Here, we use $max\_size = \infty$. This does not in fact represent a definitive, but a **maximal** set size for the result. Thereby, we delegate capping the set size to the buffer implementation of the function; see next Subsection 4.4.6.

[78] To reiterate: Events and scores (e.g., access, repetition, usability) are used for survival score computation, as outlined in Section 4.3.3.

4.4.6 Buffers

In this Subsection, we detail our buffer implementations.
As outlined previously (Table 1), we implement two distinct buffer variants – one that is purely LLM-based (Subsection 4.4.6.1), and one embedding-LLM hybrid that employs A-MEM (W. Xu et al. (2025); Subsection 4.4.6.2) as storage backbone.

All buffers subclass the *AbstractEpisodicBuffer* and must therefore provide (beyond basic CRUD operations, which we will not discuss beyond triviality), three key functions that govern how memory items relate to one another: `find_useful_items_for` (used in the reading transaction), `match_items`, and `group_items` (both used in the writing transaction), as introduced in Section 4.3.4) and framed in the context of our transaction implementation in the previous Subsection 4.4.5. Please note that we use the term node and (memory) item interchangeably to refer to our implementation of a single *AbstractChunkNode* (delineated in Subsection 4.4.3).

4.4.6.1 LLM-Only Variant

The LLM-only variant is rather narrow in scope, as all major functions are delegated to language model calls.

`find_relevant_items_for`(*query*, *candidates*, *max_size*) → *ranked_items* Given a request payload (`query`, i.e., the prompt of the GoT thought to be constructed) and a list of candidate items (`candidates`), this function filters non-meaningful items and ranks the remaining according to the relevance of the given (task-) context contained in `query`, optionally capped at `max_size`. Filtering and ranking is performed in one LLM call. Item nodes (*candidates*) are rendered into the prompt individually. We use slightly different prompts depending on whether the buffer variant is chunking-enabled or not (see Appendix A.3).

`match_items`(*contents*, *relationship candidates*) → *matches* Given a set of *contents* (reflections, either newly inserted or already present) and a list of *candidate* items, this function queries the LLM to identify every *relationship* (part-whole or equivalence) between the two sets. *contents* are matched against candidates, that is, against their contained reflection payload (when concrete), or textual index representation (when abstract). By default, each item may match at most one can-

didate, though the same candidate may be matched by multiple items. In cases where the LLM still assigns multiple candidates to a single item, the last match is taken. Relationship-specific prompts are provided in Appendix A.3.

`group_items`(*items*) → *groupings* Given a list of concrete root nodes (*items*), this function queries the LLM to return one or more groupings that exhibit thematic coherence, to be potentially assigned a new abstract parent. The respective prompt can be found in Appendix A.3.

4.4.6.2 Embedding-Hybrid Variant

The LLM-embedding hybrid buffer variant employs A-MEM (W. Xu et al., 2025) (introduced in Subsection 3.3) as storage backbone, which itself employs a vector database.[79]

A-Mem provides the ability to store simple textual payloads as so-called *MemoryNote*s. This enables us to insert all of our nodes individually. Either their textual reflection, or index summary. However, A-MEM's retrieval functionality is confined to retrieve the k nearest neighbors of a given query string.

This raises a fundamental problem with how we want to operate our buffer logic, since we refuse to choose k arbitrarily: We want `find_useful_items_for` to retrieve the entire *meaningful* neighborhood of items; we want `match_items` to discern between part-whole and equivalence, and `group_items` to capture the entire cluster an item resides in.

Therefore, we build our own retrieval logic that directly communicates with A-MEM's vector database. To this end, we define:

- A **distance measure between text embeddings**: We employ the *absolute cosine distance* defined as $d = 1 - |\cos|$, where cos is the cosine similarity between two embedding vectors. This formulation treats vectors pointing in opposite directions as semantically close (since we care little about semantic negations).

[79] For context on embedding semantics, please see prelude Section 1.1.

- The `max_link_distance`[80] as the maximal embedding distance between two *MemoryNote*s that have a verified connection in A-MEM (with a fallback value of 0.3). This will give us a sense of cluster size, and serve as a threshold.
- An internal helper function that computes distances scores between a query string and a list of candidate items, with score alignment for abstract nodes:

  Recall from Section 4.4.3 that we deliberately chose **not** to implement a custom embedding-based variant of `AbstractChunkIndex`, primarily for reasons of simplicity and integration overhead. Instead, we directly insert the LLM-computed summary string into A-MEM as-is. However, we anticipate that this could introduces a bias during retrieval: Abstract nodes, which serve as summaries of their respective clusters, may receive lower similarity scores than their more detailed children. To counteract this, we redefine the retrieval score of an abstract node n_a as the mean of its native similarity score and the average score of its children n_aC. Formally:

  $$\text{score}(n_a) := \frac{1}{2} \cdot \text{raw}(n_a) + \frac{1}{2} \cdot \frac{1}{|n_aC|} \sum_{c \in n_aC} \text{raw}(c)$$

  where $\text{raw}(n)$ denotes the unadjusted retrieval score of node n, and n_aC is the set of all direct children of n_a.

This enables us to implement our buffer logic as follows:

`find_relevant_items_for`(*query*, *candidates*, *max_size*) → *ranked_items* Given a request payload (`query`, i.e., the prompt of the GoT Thought to be constructed) and a list of candidate items (`candidates`), this function is to filter non-meaningful items and rank the remaining according to their relevance to the given (task-) context provided in `query`, optionally capped at `max_size`.

We employ our helper function to rank all *candidates* according to their embedding distance to the thought contained in *query*, and use `max_link_distance` as threshold to filter for meaningfulness.

[80] Recall that A-Mem assigned **links** between verified neighbors. See Subsection 3.3

`match_items`(*contents*, *relationship candidates*) → *matches* Given a set of *contents* (reflections, either newly inserted or already present) and a list of *candidate* items, this function seeks to identify a given *relationship* (part-whole or equivalence) between items of the two sets.

We use our helper function to compute embedding distances between all *contents* and *candidates*, and filter matches based on a distance threshold associated with the selected relationship type. Specifically, for *equivalence*, a candidate must fall below a hardcoded threshold of `0.05`; for *part-whole*, it must fall below `max_link_distance`.[81] By default, at most one matching candidate is allowed per item – namely, the one with minimal distance under the threshold. However, the same candidate may be matched by multiple items.

`group_items`(*items*) → *groupings* Given a list of concrete root nodes (*items*), this function returns one or more groupings that exhibit thematic coherence, to be potentially assigned a new abstract parent.

To obtain the groupings we perform agglomerative clustering[82] over the embeddings of all *items* and use `max_link_distance` as cluster threshold.

This concludes our implementation Section.

[81] In hindsight, using disjoint threshold intervals may have improved conceptual clarity. However, this may not strictly be necessary in practice, as only one relationship type is evaluated per call.

[82] Where clustering begins with each item in its own cluster and iteratively merges the pair of clusters with the smallest pairwise embedding distance (i.e., single linkage), until a stopping condition is met.

Chapter 5

Experiments

In this Chapter, we present our experiments and their subsequent evaluation.

All experiments share a common foundation (Section 5.1), that is further localized within their respective sections. We conduct two main experiments.

The first verifies a core assumption: That Self-Distillation (J. Huang et al. (2022); Section 3.1.1) can benefit from more advanced prompting strategies (cf. Section 3.1) – specifically, those that model increasingly complex reasoning structures, referred to here as "reasoning paradigms."

Building on this premise, we propose a novel approach to Working Memory (WM; Section 4) for Large Language Models (LLMs), carefully grounded in cognitive science. We implement four concrete variants (Section 4.4) of our agnostic core architecture (Section 4.3), designed for integration into the most recent reasoning paradigm, Graph-of-Thought (GoT, Besta et al. (2024b); Section 3.2.1.4). This configuration is evaluated in our second experiment (Section 5.3).

Limitations are discussed in Chapter 6.

5.1 Global Setup: Tasks in Graph-of-Thought

We specify the overarching experimental setup reused in all of our experiments. This will be localized accordingly, in the respective experiment.

All experiments are conducted within the Graph-of-Thought framework (introduced in Section 3.2.1.4), using the original task implementation provided by Besta et al. (2024b), as summarized in Table 2.

Table 2
Overview of employed tasks and experimental configuration from the original Besta et al. (2024b) implementation, available at https://github.com/spcl/graph-of-thoughts (version `v0.0.2`, commit `f6be6c0`). Note that we do not adopt the Document-Merging task.

Task Name	**Keyword Counting**
Description	Count keyword frequencies in a document
Example I/O	`Input: Text about various countries` `Output: {Germany: 3, France: 2, ...}`
#Samples	100
GoT Methods	IO, COT, TOT1, TOT2, GOT4, GOT8, GOTX
Evaluation Metric	Accuracy

Task Name	**Set Intersection**
Description	Return the intersection of two sets of integers
Example I/O	`Input: Set A: {1,2,3,4}, Set B: {3,4,5}` `Output: {3,4}`
#Samples	100
GoT Methods	IO, COT, TOT1, TOT2, GOT
Evaluation Metric	Accuracy

Task Name	**Sorting**
Description	Sort a flat list of integers in ascending order
Example I/O	`Input: 1 4 6 2 4 9 8 7 5 4` `Output: 1 2 4 4 5 6 7 8 9`
#Samples	100
GoT Methods	IO, COT, TOT1, TOT2, GOT
Evaluation Metric	Accuracy

In addition, the tasks Set-Intersection and Sorting both come in three different variants of either set or list size: 32, 64, and 128.

The implementation of the specific reasoning paradigms (I/O, CoT, ToT, GoT), to which we will refer to as *GoT-Methods* (or simply *methods*) are characterized as follows:

- `I/O` refers to standard-prompting (Subsection 3.2.1.1), executed with a single `GENERATE` operation.
- `COT` refers to Chain-of-Thought prompting (Subsection 3.2.1.2), also executed with a single `GENERATE` operation.
- `TOT` refers to a Graph of Operations (GoO) specifying a full Tree-of-Thought (Subsection 3.2.1.3), executed across multiple `GENERATE` and `SCORE` operations. Across tasks, `TOT1` uses a wider tree, where on each level there are more branches, in contrast to `TOT2` using a tree with more levels, but with fewer branches per level.
- `GOT` refers to a GoO specifying a full Graph-of-Thought, that, in addition to the ToT variants, also employs `AGGREGATE` operations. In Keyword Counting, the different variants specify the the number of passages the text is split into, before being re-aggregated (i.e. the divide-and-conquer depth), where in `GOTX` each sentence is considered its own passage.

We refer to the Graph-of-Thought framework when writing "GoT", but to the respectively used GoT-*Method*, when writing "`GOT`".

Finally, we record a fundamental flaw in the original implementation: Across the above tasks, the scoring function is hardcoded to **reproduce the exact gold score, which effectively leaks ground-truth information**[83]. However, we retain this behavior to remain compatible with the baseline framework [84] and to keep our focus on our own investigation.

[83] Under a scenario assuming maximal task-agnosticism where a scoring function cannot be defined apriori, and must therefore be delegated to the LLM itself – the core intuition behind GoT's predecessor, Tree-of-Thought, cf. Section 3.2.1.3.

[84] Admittedly, we only discovered this issue retrospectively, after the first experimental run was already completed.

5.2 Graph-of-Thought in Self-Distillation

We examine the suggestion by J. Huang et al. (2022) to incorporate advanced generation strategies into the data labeling pipeline of Self-Distillation (Section 3.1.1). Specifically, **we hypothesize that replacing the Chain-of-Thought (CoT; Section 3.2.1.2) strategy with increasingly structured successors, namely Tree-of-Thought (ToT; Section 3.2.1.3) or Graph-of-Thought (GoT; Section 3.2.1.4), yields progressively higher-quality training data, thereby improving the overall performance of the self-distilled model.** However, due to resource constraints, we confine our inquiry to solely compare CoT and ToT.

In the following, we define our experimental setup (Section 5.2.1), analyze the results of the data labeling pipeline (Section 5.2.2), and evaluate the resulting model performance (Section 5.2.3).

To give a high-level summary in advance: We find our core hypothesis supported. Better reasoning paradigms yield higher-quality labels, which in turn lead to improved downstream model performance.
However, we also acknowledge the limitations of our evaluation up front: First, model performance is assessed under minimal reasoning conditions (i.e., standard prompting), which may not fully capture task-specific capabilities. Second, while differing trends in label quality suggest that our tasks differ fundamentally from those used by J. Huang et al. (2022), we do not further pursue the inconclusive patterns observed in label confidence and calibration, as our focus lies on isolating the effects of reasoning paradigms.

5.2.1 Local Experimental Setup

We instantiate the global experimental configuration (cf. Section 5.1) for our local setup as follows:

We run the full Self-Distillation pipeline, comprised of (1) one data labeling and (2) one model fine-tuning step, on all three tasks (Keyword-Counting, Set-Intersection, Sorting) using two GoT-Methods: `COT`, `TOT1`. Each task dataset of 100 samples is split into an 80/20 train/test partition. Both, data generation and finetuning is conducted with `gpt-3.5-turbo-0125` [85].

[85] Documented at https://platform.openai.com/docs/models/gpt-3.5-turbo. Cf. the predecessor model, GPT-3, as introduced by Brown et al. (2020).

Data Labeling & Annotation We follow the label generation and data augmentation procedure proposed by J. Huang et al. (2022), using the same four augmentation variants of each labeled sample. The process is run with the following configuration:

- Sampling temperature: 0.7
- Majority voting size: 32
- Maximum output token limit: None
- Number of samples to label per task: 80 (the respective training set).

where the first two are identical to those used by J. Huang et al. (2022).
As in J. Huang et al. (2022), each labeled instance consists of a *(question, reasoning, answer)* triplet. To extract the reasoning trace prior to augmentation, we proceed as follows: For the `ToT1` method, we collect all reasoning outputs along the path from the final answer node to the root of the reasoning graph, concatenated in order of generation. In contrast, the `CoT` method involves only a single node, and the reasoning trace is extracted directly.

Fine-Tuning We use the OpenAI fine-tuning API [86] to train a dedicated model for each (task, method) pair on the labeled outputs.
Each respective training split is further pruned to at most 50 samples to reduce cost. We apply 5-fold cross-validation on a pruned training set, yielding 40 training and 10 validation samples per fold. Further configuration is as follows:

- Sampling temperature: 1.2 (following J. Huang et al. (2022))
- Batch size: default (set by OpenAI)
- Learning rate multiplier: default (set by OpenAI)
- Number of epochs: 10

Each fold produces a fine-tuned model with multiple checkpoints (set by OpenAI). For each (task, method), among all folds and checkpoints, we select the one with the lowest overall validation loss for final evaluation on the test split. We report model performance before and after fine-tuning, evaluated on all 20 test samples using majority voting with voting size 5, each generated via standard prompting (cf. Section 3.2.1.1) to further reduce inference cost.

[86] As documented at https://platform.openai.com/docs/api-reference/fine-tuning.

5.2.2 Results I: Data Labeling

We evaluate the quality of generated labels and examine the reproducibility of empirical trends reported by J. Huang et al. (2022) in the context of Self-Distillation.

It becomes directly apparent to us that the model was unable to correctly label any sample in the Keyword-Counting task (i.e. $accuracy = 0.0\%$, even across individual votes), and henceforth, we exclude it from the remainder of this experiment. This is surprising, since we use the same model and task implementation as Besta et al. (2024b).

To summarize up front: We find our core hypothesis is supported by the finding that ToT consistently generates higher-quality labels than CoT. However, majority voting fails to improve results for Set-Intersection. Overall, the model exhibits underconfidence in its predictions. As our primary focus is on reasoning paradigms, we do not pursue a deeper investigation into these inconsistencies.

5.2.2.1 Label Quality

We report the accuracy across both tasks and GoT-Methods in Figure 20. As expected, ToT produces superior results on both tasks. However, for Set-Intersection (Figure 20b), the accuracy of the majority-voted answers is notably lower than that of individual generations, indicating that majority voting may suppress correct but less frequent outputs.

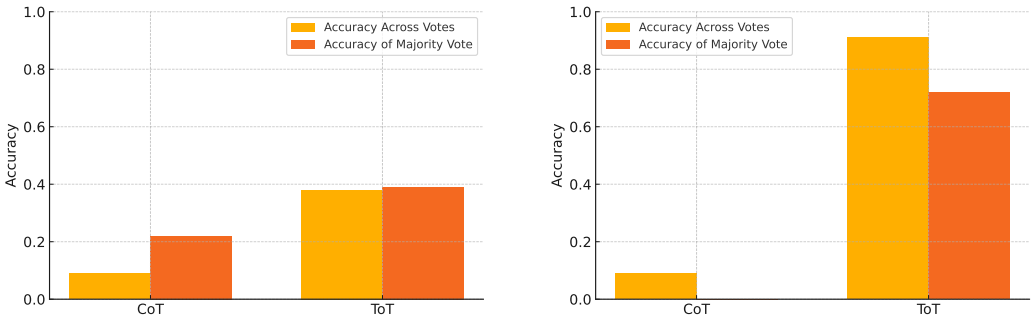
Overall, these results suggest that models trained on CoT-generated labels are likely to perform poorly due to the high level of noise. While ToT consistently reduces noise compared to CoT, the only setting where correct signal *outweighs* noise is Set-Intersection with ToT (Figure 20b), despite majority-voting performing worse. Accordingly, this is the scenario in which we expect the resulting distilled model to benefit most from reliable supervision.

5.2.2.2 Empirical Trends

In the analysis of generated labels, J. Huang et al. (2022) define *confidence* as the proportion of generation attempts that agree with the majority-voted label.[87] Given a set of generated labels and their respective majority voting data, they record the accuracy per confidence level as the fraction of

[87] Recall that J. Huang et al. (2022) employ majority voting, known as *Self-Consistency* (X. Wang et al., 2023), to refine generated labels. See Section 3.1.1.3. Specifically, the final label is selected as the majority outcome across multiple generation attempts. Accordingly, *confidence* is defined as *(size of the majority vote)/(total number of votes)*.

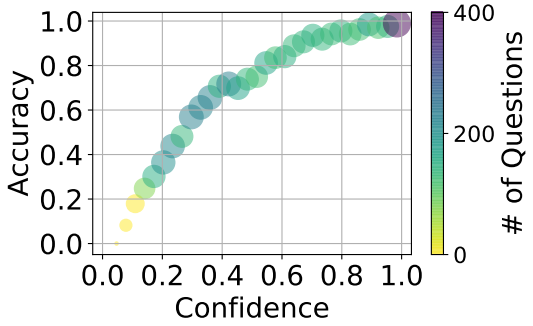
Figure 20
Accuracy comparison of CoT and ToT during the data labeling step of Self-Distillation. We report both the average accuracy across individual generations and the final accuracy of the majority-voted label. Each method is applied to two tasks, where for each, the total number of samples labeled is $n = 80$, with majority voting size 32, yielding $80 \cdot 32 = 2.560$ individual generations per task.



(a) ***Sorting task***. *The ToT method outperforms CoT across both metrics. However, the gap between majority-voted accuracy and accuracy across votes is marginal for ToT, but large for CoT.*

(b) ***Set-Intersection task***. *ToT drastically improves accuracy compared to CoT. Interestingly, majority voting significantly lowers accuracy in both cases. For CoT, to* 0.0.

Figure 21
Accuracy versus confidence on GSM8K training-set questions, reported by J. Huang et al. (2022). Accuracy reflects the proportion of correct answers at each confidence level. Confidence denotes the fraction of generation attempts that produced the majority-voted label.



correct majorities under a given confidence level,[88] together with the number of labels per confidence level, as depicted in Figure 21.

They generate answers to questions in the GSM8K dataset (Cobbe et al., 2021), and observe:

1. **High confidence positively correlates with higher accuracy**: More confident answers are more likely to be correct.

[88] For example: Suppose voting rounds a, b each have 5 participants, where in a, 3 votes are indeed proposing the correct answer, and in b 3 are incorrect. The confidence of a and b is, in both cases, $3/5 = 60\%$. However the accuracy *under* the confidence level of 60% is $1/2 = 50\%$: There are two voting with a confidence of 60%, and only one of them is indeed reaching the correct outcome.

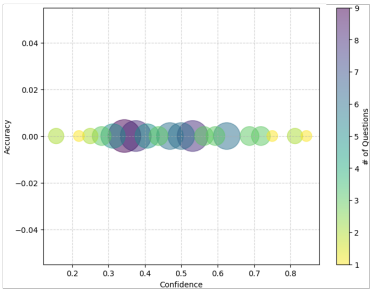
2. **Higher accuracy corresponds to more labels**: Confidence levels with higher accuracy tend to include more questions (as indicated by larger and darker circles in Figure 21).
3. The predicted confidence on GMS8K is *well-calibrated* (Guo et al., 2017).[89] This is the case when there is minimal deviation from the diagonal in the shown diagrams.

In summary, Guo et al. (2017) suggest that incorporating all majority-voted answers as training samples would introduce only minimal noise – because most questions fall into high-confidence regions where stronger majorities are more likely to be correct, and thus outweigh weaker ones.

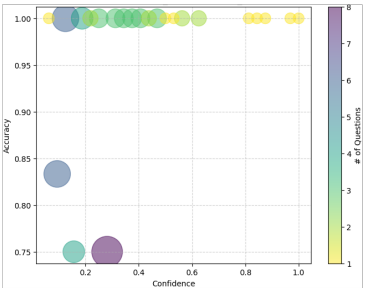
Our Results We aim to replicate the conditions under which incorporating all majority-voted answers introduces only minimal noise. Namely, a positive correlation between confidence and accuracy, and a concentration of predictions in high-confidence regions (which together implies higher average label quality). In addition, we briefly assess the calibration of the model on our tasks.

To this end, we reproduce the analysis of J. Huang et al. (2022) within our experimental setup and present the corresponding results in Figure 22 and Figure 23.

Figure 22
Accuracy versus confidence on generated labels of the Set-Intersection training-set. Accuracy reflects the proportion of correct answers at each confidence level. Confidence denotes the fraction of generation attempts that produced the majority-voted label. For voting rounds resulting in a tie, an answer was chosen at random.



(a) *Accuracy versus confidence on Set-Intersection, with CoT. 0/80 voting rounds resulted in a tie.*

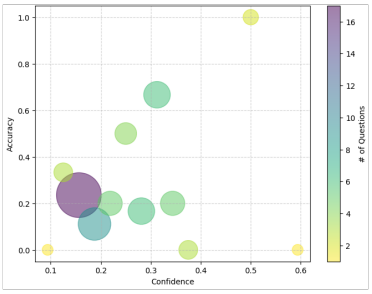


(b) *Accuracy versus confidence on Set-Intersection, with ToT. 18/80 voting rounds resulted in a tie.*

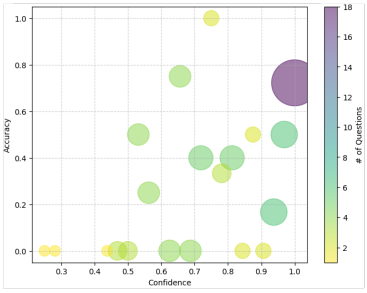
Correlation between Confidence and Accuracy In terms of confidence-accuracy correlation, we observe a clear positive trend in the Sorting task. Under both CoT and ToT, higher confidence tends

[89] Well-calibrated, in the sense of Guo et al. (2017), represents the "trustworthiness" of confidence scores. That is, that confidence scores corresponds closely to the empirical correctness of the answer. I.e. $P(\text{Answer is correct} \mid \text{Confidence} = p) = p$. E.g., when the model predicts to be 80% confident in an answer (because 80% of its attempts gave that same result), that answer really is correct about 80% of the time.

Figure 23
Accuracy versus confidence on generated labels of the Sorting training-set. Accuracy reflects the proportion of correct answers at each confidence level. Confidence denotes the fraction of generation attempts that produced the majority-voted label. For voting rounds resulting in a tie, an answer was chosen at random.



(a) *Accuracy versus confidence on Sorting, with CoT. 18/80 voting rounds resulted in a tie.*



(b) *Accuracy versus confidence on Sorting, with ToT. 0/80 voting rounds resulted in a tie.*

to coincide with higher accuracy – with some outliers.

For ToT on Set-Intersection, one might very cautiously speak of a weak correlation, as there are no high-confidence errors. Nonetheless, the dominant pattern is low-confidence on correct answers, albeit the majority being correct.

For CoT on Set-Intersection, the model fails in the overwhelming majority of cases, rendering confidence uninformative.

Distribution of Confidence Across all tasks and methods, we observe that few predictions fall into high-confidence regions. This suggests that in our experimental setup, the model is either underconfident or less consistent. While these discrepancies point to fundamental differences between our tasks and GSM8K, we refrain from further speculation about the possible causes.

Calibration We do not observe the same degree of calibration as reported by J. Huang et al. (2022) on GSM8K.

For Set-Intersection, the CoT variant (Figure 22a) completely fails to assign any correct labels, despite producing a broad range of confidence values (mostly below 0.5). Since the vast majority of the predictions are inaccurate, this results in systematic overconfidence.

In contrast, the ToT majorities (Figure 22b) yield mostly correct answers, yet overwhelmingly low confidence values, indicating clear underconfidence.

For Sorting, calibration appears somewhat better under both CoT and ToT. Confidence values are

more distributed and some alignment with accuracy is visible, but many predictions still deviate from the diagonal, with the majority residing below, also indicating underconfidence.

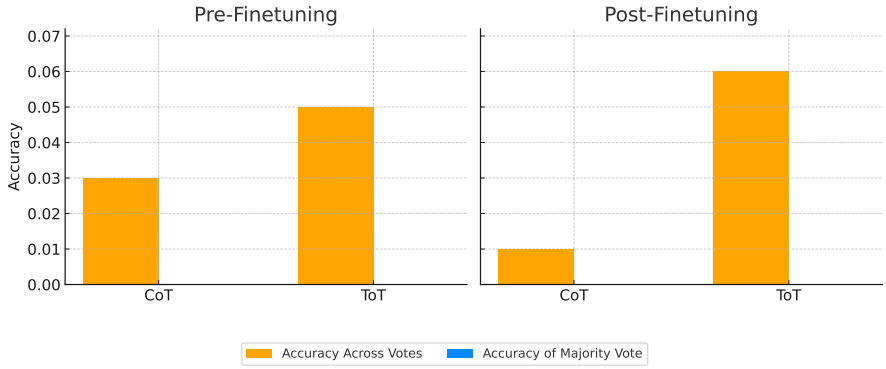
Conclusion Ultimately, while ToT appears largely superior to CoT as a generation strategy – both in terms of raw accuracy and, to some extent, in confidence-accuracy correlation – the results remain inconclusive as to why higher accuracy under ToT does not translate into more high-confidence predictions. However, we do not pursue a deeper investigation into these inconsistencies.

5.2.3 Results II: Distilled Model Performance

We evaluate model performance on both tasks, before and after the finetuning step. To minimize inference costs, we evaluate the fine-tuned models using standard prompting (cf. Section 3.2.1.1).

For the Set-Intersection task, results align well with the observed label quality. Models trained on CoT-generated labels show a slight performance drop of −0.02 accuracy points after fine-tuning. In contrast, models trained on ToT-generated labels improve by +0.01, despite overall low accuracy levels (see Figure 24).

Figure 24
Accuracy comparison before and after fine-tuning on the Set-Intersection task using standard prompting. "CoT" and "ToT" refer to reasoning strategies used during label generation (not inference). We report both average per-generation accuracy (Accuracy Across Votes) and majority-vote accuracy (Accuracy of Majority Vote), using 5 votes per data point (20 points total). ***All majority-vote results yielded*** 0.0% ***accuracy****. The vertical axis is capped to emphasize fine-grained differences.*



This supports our core hypothesis: Better reasoning paradigms yield higher-quality supervision, which in turn translates to better downstream model performance – even when tested under a basic standard-prompting regime.

For the Sorting task, accuracy remains at 0.0%, both before and after fine-tuning. As a result, we omit a corresponding plot.

Conclusion While, standard prompting may not suffice to fully elicit gains on the given tasks, thus restricting interpretability of the absolute performance, the relative differences between CoT and ToT labels remain clearly visible in the label quality (cf. previous section) *and* the final model performance on Set-Intersection. Concludingly, we consider our hypothesis to be supported and

refrain from further investments into more compute-intensive evaluation regimes that would better elicit model capabilities.

5.3 Graph-of-Thought with Working Memory Variants

We evaluate our working memory (WM) variants (introduced in Section 4.4) in the context of the Graph-of-Thought framework (GoT; introduced in Section 3.2.1.4).
Hereby, we seek to examine the central hypothesis that originally motivated the development of our four implementation variants. To reiterate (from Section 4.4.1), these are:

Hypothesis I: Introducing a working memory (WM) as an advanced reflection store (cf. Section 4.3.2) improves the performance of GoT.

Hypothesis II: A WM utilizing chunking strategies (i.e., one that builds a forest of memory items) should outperform a flat memory setup (i.e., one that uses a simple set of unstructured items).

Hypothesis III: For a WM with limited capacity, a LLM should serve as a superior model for chunking and retrieval compared to embedding-based approaches.

To this end, we begin in specifying our concrete experimental setup and integration of WM into GoT (Subsection 5.3.1), present the main results (Subsection 5.3.2), followed by a qualitative evaluation of memory contents (Subsection 5.3.3), and an analysis on token consumption (Subsection 5.3.4).

5.3.1 Local Setup

We evaluate each WM variant introduced in Section 4.4 within a controlled Graph-of-Thought (GoT) setup, using a subset of tasks and standardized parameters, as follows.

Tasks and Data Due to weak results on *Keyword Counting* (cf. Section 5.2.2), and to reduce computational cost, we restrict our evaluation to the *Set-Intersection* and *Sorting* tasks, using their 32-sized variants. We test each of the 4 WM variants with the `IO`, `CoT`, `ToT2`, and `GOT` methods respectively. For each setting, we test 20 samples with a majority voting size of 3 samples.

Hyperparameters We adopt the exponential forgetting factor of 0.995 from Park et al. (2023) (cf. Section 4.3.3). In the A-MEM (W. Xu et al., 2025) based variants (cf. Section 4.4.6.2), we reduce

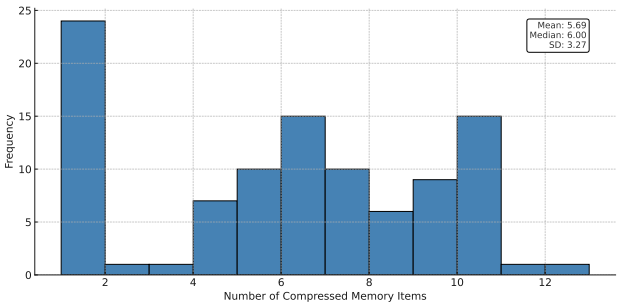
A-MEM's default evolution threshold from 100 to 10 steps to encourage more frequent re-indexation. For all memory variants, we use a fallback similarity threshold of 0.3 (cf. `max_link_distance` in Section 4.4.6.2).

Model Setup All experiments are run using `gpt-4o-mini-2024-07-18`,[90] with temperature set to 0.8.[91]. Within A-MEM, we use the embedding model set as default by W. Xu et al. (2025), `all-MiniLM-L6-v2`,[92] a variant of the model originally proposed by W. Wang et al. (2020).

WM Integration into GoT We integrate the full WM read and write process as outlined in Figure 14. However, WM is only connected to the `GENERATE` and `AGGREGATE` operations, as these are the only LLM-dependent operations in the respective task implementations provided by Besta et al. (2024b).[93] As such, WM is queried before and updated after each `GENERATE` and `AGGREGATE` operation.

Determining Memory Capacity To choose a meaningful memory capacity, we conducted a single exploratory run with unlimited capacity across all WM variants using the `GOT` method. We then analyzed how many items were selected for compression in the read transaction (that is, the output of the "Find Relevant Items" activity depicted in in Figure 15b; cf. implementation in Section 4.4.5). The resulting distribution shown in Figure 25.

Figure 25
Distribution of memory items selected for compression during a single run of the `GOT` method with unlimited WM capacity.



[90] Documented at https://platform.openai.com/docs/models/gpt-4o-mini
[91] Higher temperatures were initially tested to align with the label generation regime (see previous experiment in 5.2), but the model began producing non-English outputs at higher temperatures.
[92] Documented at https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
[93] E.g., in contrast, `SCORE` relies on ground-truth access and `KEEP_BEST_N` performs hard score comparisons.

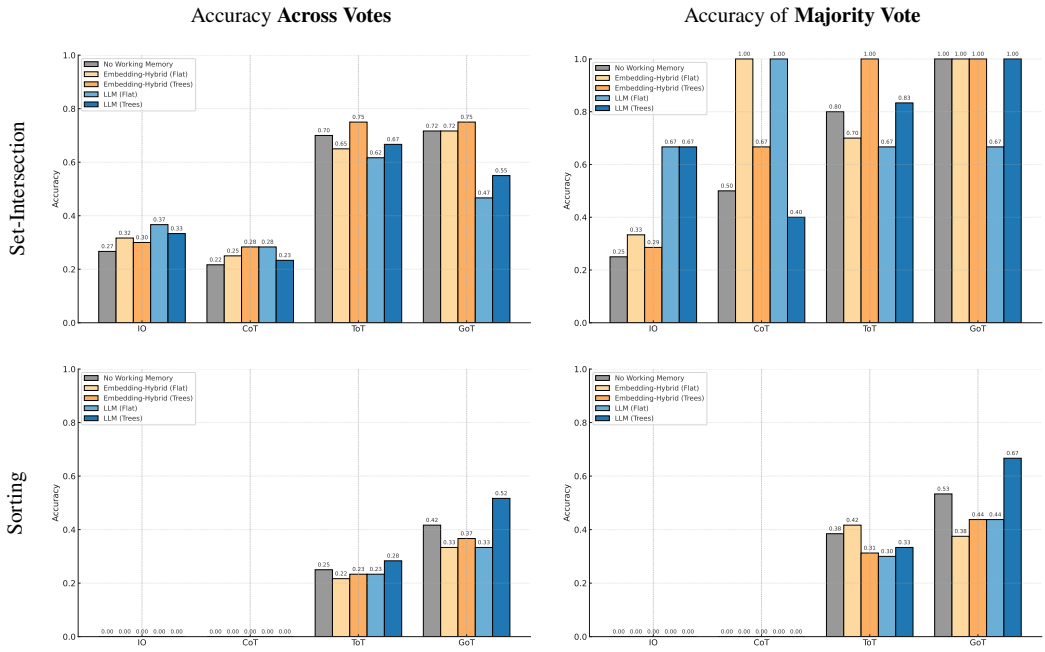
The resulting distribution is approximately tri-modal (or normal, with two outliers at the end) with a mean around 6 and a standard deviation of around 3. This coincidentally aligns with Miller's classical 7 ± 2 formulation of human working memory capacity (Miller, 1956). While based on a limited sample and potentially coincidental, the similarity is intriguing. Motivated by this finding, we set the buffer capacity to 7 for all memory variants.

5.3.2 Primary Results

As base results, we provide (1) the accuracy of all WM variants across all tasks and reasoning methods (Figure 26), in combination with (2) the mean memory pressure across all runs, over time (on ToT and GoT methods; Figure 27). We attend to each of our hypothesis respectively.

Hypothesis I: Working Memory improves Graph-of-Thought Naturally, our hypothesis should be evaluated in the context of accuracy improvement (Figure 26). However, as in previous experiments, our analysis is limited in sample size due to resource constraints. Consequently, the resulting accuracy data should be interpreted with caution. We highlight the following key observations:

Figure 26
Accuracy comparison of all WM variants across tasks (rows) and accuracy types (columns). "No Working Memory" corresponds to vanilla GoT, "Trees" indicate a WM variant that is operated by a transaction employing chunking, "Flat" ones are not (cf. Implementation Section 4.4).



1. **Apparent Instability**: Recall that each sample is evaluated with a freshly initialized working memory, and memory interactions (reads and writes) only occur during `GENERATE` and `AGGREGATE` operations. Since the `IO` and `COT` methods each involve only a single `GENERATE` step, they are structurally incapable of benefiting from working memory. Nevertheless, results suggest improvements even in these cases, which WM cannot influence.

This is clearly visible for Set-Intersection, and cannot be falsified for Sorting (since overall accuracy there amounts to 0.0 for `I/O` and `COT` in all cases). Accordingly, this discrepancy points to a *general* instability or noise in the results.

2. **No Clear Winner**: While the "No Working Memory" baseline appears inferior to at least one WM variant in each setting, we do not observe a consistent or dominant variant across tasks. One might speculate that *Embedding-Hybrid (Flat)* performs best on Set-Intersection,[94] and *LLM (Trees)* excels on Sorting (assuming the outlier in ToT can be disregarded). Our speculative intuition is that *LLM (Trees)* may be superior overall – if one were to disregard the Set-Intersection results as too unstable, which we lack the statistical grounds to justify.[95] As such, due to the aforementioned instability, we refrain from attributing performance effects to the design of individual WM variants alone.

Overall, while we find *LLM (Trees)* to be the most intuitively promising variant and observe no evidence that WM degrades GoT performance across variants, the results remain inconclusive and preclude firm conclusions. A more definitive assessment would require further investigation – either through deeper analysis of result variance, additional computational resources, or inherently more stable tasks.

Hypothesis II: Chunking is Superior to Non-Chunking Our second hypothesis must also be set aside due to the instability of the results. While the “Tree” (chunking-enabled) variants consistently [96] lead across ToT and GoT methods – potentially supporting our hypothesis – this trend cannot be reliably interpreted given the limited robustness of results. However, we note faint indications of support emerging from our subsequent qualitative evaluation of memory content (Section 5.3.3) and token consumption analysis (Section 5.3.4); though both should be interpreted with caution.

Hypothesis III: LLM-Only Variants Are Superior to Embedding-Hybrids While we initially intended to evaluate this hypothesis through task accuracy – an analysis we must defer given the instability discussed above – we nonetheless observe preliminary indications of support from our

[94] Again, we stress caution: Later analysis (in Section 5.3.4) revealed that the A-MEM based variants only read from memory in the `Aggregation` operation, and may be impaired substantially.

[95] This may be defensible in the majority voting case at most, given that our Self-Distillation experiment indicates majority voting suppresses correct answers in Set-Intersection (cf. label quality in Figure 20).

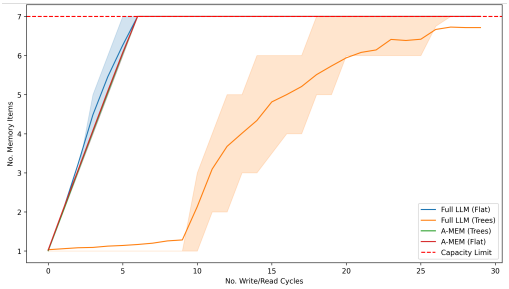
[96] Again, we urge caution: We may suppose that A-MEM variants are partially broken, as per the limited retrieval activity observed in Section 5.3.4.

qualitative evaluation of memory contents (Section 5.3.3) and our analysis of memory pressure (Figure 27). We directly follow with the latter below.

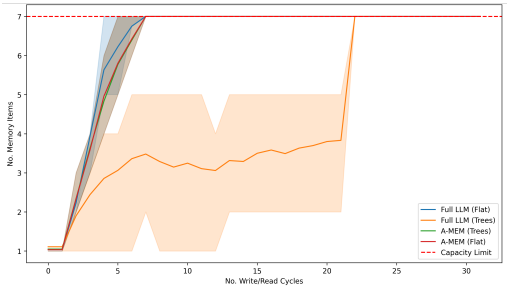
Memory Pressure Attending to the memory pressure plotted in Figure 27, we see the mean buffer capacity evolving over time. Importantly, within our implementation (cf. Section 4.4.3), the only mechanism for reducing memory pressure (i.e. capacity) is the construction of item-trees (chunks).

The flat variants are incapable of constructing chunks. Thus, it directly becomes apparent that the *LLM (Tree)* variant is superior in constructing item trees, whereas the *A-MEM (Tree)* variant appears to do so to a much lesser extent.[97]

Figure 27
Mean memory pressure relative to buffer capacity over read/write cycles, across all tasks. The shaded region indicates the interquartile range (IQR). The y-axis corresponds to the buffer capacity. All items can have a capacity of at most 1 (cf. Section 18). Note: *In the* `GOT` *method, tasks have different numbers of reasoning steps (i.e. read/write cycles; Sorting: 21, Set-Intersection: 31).*



(a) *Memory pressure under the* ***Tree-of-Thought*** *method.*



(b) *Memory pressure under the* ***Graph-of-Thought*** *method.*

This behavior aligns with the broader view that LLMs may serve as "general-purpose engines for data management" (Zhou et al., 2025). In light of Hypothesis II however, we find that our comparison attends a more fundamental question: Are LLMs superior clusterers compared to embedding-based solutions?

Superiority here may be divided into two axes: (1) raw performance, and (2) task flexibility. In terms of quality, both approaches may be said to be roughly on par in general (though depending

[97] While this may reflect a genuine advantage of LLM-based clustering, we acknowledge an alternative possibility – namely, that in some tasks no meaningful tree structure can be formed. However, in our setting, our qualitative analysis suggests otherwise (Subsection 5.3.3). Yet, it appears more likely that the A-MEM variant is inherently flawed, as found in our token usage evaluation, in Section 5.3.4.

on the task at hand; cf. X. Xu et al. (2023); S. Wang et al. (2023)) [98]. However, embedding-based models typically require task-specific finetuning (cf. Schopf et al. (2024)), whereas LLMs, as in our case, operate flexibly without such adaptation. We see our results as a confirmation of this flexibility, which is critical for our broader goal, to obtain a flexible and task-agnostic WM mechanism.

Therefore, this comparison may be considered supportive for Hypothesis III, in terms of flexibility, but only as a preliminary signs of such support in regards of raw performance. As such, we stress: (1) a more rigorous evaluation of raw performance as necessary, as previously outlined, and additionally (2) that our embedding-hybrid variants must not be interpreted as a robust baseline for comparing LLM-based retrieval against embedding models.

[98] For our application however, this comparison is not so easy to make, since we lack the proper integration of a classification-head onto the embedding model (employed within A-MEM). Instead, we define flexible classification threshold in reliance on A-MEM's linkage feature – which is, in turn, LLM-driven. As such, since A-MEM is a hybrid system relying primarily on embeddings but also employing a LLM for storage management, the distinction is difficult to make.

5.3.3 Qualitative Evaluation

We qualitatively examine buffer contents across tasks and implementation variants, guided by three sets of questions. These serve as interpretive lenses rather than strict hypotheses:

1. Is there evidence that supports the use of flat memory structures? What types of errors are recorded? (cf. Hypothesis II; see Section 5.3.1).
2. How well are clusters (i.e., trees in memory) [99] formed, particularly regarding over-aggregation? Do the respective models yield usable and well-separated clusters early on (as hypothesized in the implementation of the write transaction; Section 4.4.5)?
3. Are there any observable differences in buffer content between successful and unsuccessful trials?

To this end, we extract a snapshot of each buffer after the respective trial has completed and manually examine the top-scoring trees (i.e., those with the highest survival score) across all implemented variants.[100]

5.3.3.1 Error Variance as Motivation for Hierarchical Memory

In both tasks, we observe a clear majority in the types of errors produced by the model. For Sorting and Set-Intersection alike, the primary error modes involve either the *duplication* or *omission* of numbers. These two error types are so dominant in the dataset that it becomes difficult to observe other sources of failure. [101]

In fact, across all flat memory variants, we identified *only a single* instance of a non-majority error type – specifically, a reflection that explicitly corrected an earlier faulty reflection. Accordingly, we conclude that flat memory contents are largely homogeneous and capture little variance in the types of reasoning errors.

In contrast, hierarchical memory variants immediately surfaced a broader and more neatly organized spectrum of error types, including:

[99] Please note that we continue to use the terms "chunk", "cluster" and "tree" interchangeably.

[100] Specifically, we investigate only the WM employed with ToT and GoT methods, since these are the only ones that actually accumulate extensive materials across multiple generation attempts (cf. Section 5.3.1).

[101] In this context, we focus on the underlying source of error as the primary criterion for distinguishing reflections. While more fine-grained categorization based on the nature of feedback (e.g., proposing vs. critiquing CoT) is possible, such distinctions are negligible for our aim of evaluating whether coarse-grained clustering by error type is justified.

- Incorrect response formats (e.g., responses missing XML closing tags), [102]
- Faulty separation of lists,
- Incorrect assertions that a sublist was already sorted, despite remaining unsorted,
- Direct repetitions of unsorted input lists without any attempt at sorting.

Thus, even in monotonous tasks with low semantic diversity and seemingly skewed error distributions (dominated by a single failure mode), the introduction of clustering mechanisms appears well-justified as a means to organize distinct error sources.

5.3.3.2 Cluster Structure

Addressing our second set of questions, we qualitatively examine the structure of clusters formed in the memory of variants that employ chunking. We focus on the degree of over-aggregation as a key diagnostic, since our insertion strategy assumes optimal clustering without reorganization (cf. Section 4.4.5) and relies on a capable relationship attributor to discern early on what should (and should not) be structurally associated.
Herein, we further differentiate between higher-order structure and lower-order structure (regarding grouping or part-whole relationships, and equivalence-relationships, respectively; as outlined in Section 4.4.3).

Lower-Order Structure For concrete summary subtrees – those formed based on equivalence relationships – the LLM-Only variant excel. We did not encounter a single ill-formed concrete summary subtree that failed to maintain internal coherence or that overlapped inappropriately with a neighboring concrete subtree, save for a single negligible outlier.

This results in memory structures that successfully summarize multiple instances of a given error type, even if that type appears rare overall. For example, we encountered concrete summary trees that exclusively captured formatting errors of the same kind (e.g., forgetting to include the closing answer tag). In another case, three subtrees with a common abstract parent well captured different granularity levels of frequency-counting feedback:

[102] This issue was anticipated during development. We implemented a two-tiered parser with a strict XML extractor backed by a relaxed fallback mechanism that tolerates missing closing tags. As a result, such malformed outputs were still parsed correctly.

- One subtree provided only feedback targeting the required total length of the output list (e.g., *must contain 16 elements*),
- Another contained exact count mismatches of various numbers (e.g., *stating "1 appears 3 times" is incorrect; in reality, '1' only appears twice*),
- A third focused exclusively on miscounting the digit 4.

In contrast, A-MEM variants frequently produced concrete summary trees that bundled together all sorts of different error types, which were better separated in the LLM-only variants.

Higher-Order Structure For higher-order trees – those formed via grouping and extended via the part-whole relationships – the distinction is less pronounced. While A-MEM variants again produced less structured aggregates, the LLM variant also showed signs of error. For instance, one tree labeled "Sorting Task Errors: Misidentification of Sorting Status *and* Frequency Mismanagement" contained two distinct error types. While each type's summary-subtree was internally coherent, the grouping as a whole was loosely coupled and less interpretable.

While this observation again also favors the LLM-only variant over A-MEM, it suggests that the current prompting used to define grouping or part-whole granularity may either be under-specified, or, express the limitation of our insertion strategy, to not include re-grouping mechanisms, while at the same time prioritizing the integration into new trees over the formation of new ones (see the design in Section 4.4.5; for further discussion, see Section 6.2).

5.3.3.3 Reflection Quality

In addition to our guiding questions, we noticed that in many cases, the actual reasoning error is not accurately captured in the `<Error-Location>` field of the reflection. Consider the example below, where the reasoning steps leading up to the actually erroneous statement are missing.

Example of an Insufficient <Error Location> in a Reflection (Capped)

```
<Error Location>
The resulting sorted list now matches the original frequencies.
</Error Location>
<Error Explanation>
The final sorted output does not accurately reflect [...]
</Error Explanation>
<Error Fixing Suggestion>
Revisit the original list to ensure accurate frequency counts:
0: 1; 1: 2; 3: 2; 4: 5; 5: 2; 6: 1; 7: 1; 9: 2.
The accurate sorted list should thus be: [0, 1, 1, 3, [...]
</Error Fixing Suggestion>
<Takeaway>
Frequent verification of the final output against the [...]
</Takeaway>
```

While this could be attributed to an underspecified prompt, it may also point at the earlier noted limitation (Section 4.4.4) that reflecting on a single thought alone may represent a potential bottleneck in our reflection mechanism, compared to other designs (such as that of Shinn et al. (2023); a limitation further discussed in Section 6.2.1.1).

5.3.3.4 Effect of Trial Success on Buffer Contents

We did not observe any systematic differences in buffer contents between successful and unsuccessful trials.

Exploratory Consideration: Still, this result left us unsettled – and we wondered whether buffers with particularly long-surviving memory items might reflect especially useful reflections contributing to success.

Diverging slightly from the qualitative nature of this evaluation, we conducted a preliminary analysis to assess whether survival scores could predict trial success. The underlying idea is that trials in which confusion dominates working memory are less likely to succeed, whereas those in which coherent chunks of information form and persist – because they are repeatedly attended to – may be

more successful.
Specifically, we fitted a series of logistic regression models per implementation and task variant, using the maximum, mean, and standard deviation of survival scores as features.

While this analysis revealed no consistent or interpretable patterns overall, one exception stood out: The LLM-Tree variant showed notably higher regression coefficients across both tasks – albeit for different score metrics in each case. Although this inconsistency undermines direct interpretability, the result may still hint at a weak correlation between survival-based accumulation and task success in this variant. At the very least, it could suggest that the LLM-Tree variant is more aligned with the intended function of accumulating and organizing useful content over time.

We did not pursue this analysis further. For completeness, we report the empirical results in Appendix A.2, though without confidence intervals or significance tests.

5.3.3.5 Conclusion

In summary, our qualitative investigation leads to the following observations. We attend to our core hypothesis established in Section 5.3.1, when fit:

1. In tentative support of Hypothesis II (Section 5.3.1):
   Multiple distinct sources of error are present in the stored reflections, across tasks and variants. While this may justify the use of clustering as a practice in general, to separate different sources and styles of feedback, it does not directly imply that a hierarchical memory structure (i.e., a chunk) should be used.
2. In tentative support of Hypothesis III (Section 5.3.1):
   LLM-based variants *appear* more capable than A-MEM variants at flexibly discerning clustering relationships under a limited insertion strategy that favors existing clusters and assumes implicit abstraction:
   - They produce internally coherent and semantically meaningful summary trees,
   - Yet sometimes struggle to maintain coherence at higher levels of abstraction.
   - This is indicative of a broader constraint: Relying on *implicitly* defined hierarchical relationships without re-organization (further discussed in Section 6.2.1.2)

3. However, due to instability in our quantitative results (cf. Section 5.3.2), we cannot conclusively determine which clustering structure performs best, leaving open the possibility that mixed clusters may offer synergistic advantages downstream.
4. As a minor finding, in tentative support of Hypothesis II and III:
   Regression coefficients linking survival scores to trial success were strongest in the LLM-Tree variant, suggesting superior performance in accumulating useful content – though results were inconsistent and remain exploratory (cf. Section 5.3.3.4).
5. Lastly, we emphasize that the narrow variability of errors across both tasks leads to homogeneous memory content, which may fundamentally limit the benefits of WM in these settings (see Section 6.3 for further Discussion).

5.3.4 Token Consumption Evaluation

We evaluate the amount of input tokens used within our WM variants, applied in the Graph-of-Thoughts (GoT; Besta et al. (2024b)) framework. The number of output tokens is omitted, since we found it negligibly small in comparison, and without significant variance. We focus on the most heavy method, the `GOT` method.[103]

We initiate with an overview of operations and their roles (Section 5.3.4.1), followed by a brief visual analysis of reading and writing behaviors across reasoning steps (Section 5.3.4.2). We then turn to a comparison of token scales and totals (Section 5.3.4.3), analyze the relative costs between operations (Section 5.3.4.4), and conclude with a broader perspective (Section 5.3.4.5).

5.3.4.1 Overview

Table 3 provides an overview of all operations stemming from our WM system, together with their cost drivers, frequency, and transactional type. Reads only use `Find` and `Compress`; all other operations occur during writes. Table 3 does not include operations stemming from Graph-of-Thought (Besta et al., 2024b) and A-Mem (W. Xu et al., 2025), which we describe below.

[103] Token usage histograms for the `TOT` method can be found in Appendix A.1. Histograms for the `IO` and `COT` methods are omitted, since they consist only of a single writing step.

Table 3
Overview of WM Operations in the context of our implementation with Cost Drivers, Frequency, and Transaction Type

Operation	*Description*	*Cost Driver*	*Frequency*	*Transaction*
Score Usability	Scores a reflection's usefulness in its associated thought.	No. reflections used in a thought	Per reused reflection	Wrtite
Reflect	Produces reflections from a thought's prompt.	No. total thoughts	Once every write	Wrtite
Chunk (Merge)	Merges a set of concrete nodes considered equivalent	No. of children of target node	On replacement of concrete node	Wrtite
Chunk (Summarize)	Summarizes a set of concrete nodes for the creation of an abstract parent node	No. of children for abstract node	On access, when abstract node is invalidated (lazy)	Wrtite
Group	Groups concrete roots. Groups > 1 receive an abstract parent (via chunk-summarize)	Reflection novelty	When reflection can't be inserted into an existing tree (other concrete roots exist)	Wrtite
Match (Part-Whole)	Matches a reflection to abstract node payloads	No. abstract nodes (matching candidates)	Every write (if abstract nodes exist)	Wrtite
Match (Equivalence)	Matches a reflection to concrete nodes	No. concrete nodes (matching candidates)	Every write (if concrete nodes exists)	Wrtite
Compress	Recompiles a set of reflections in context of query (to at most max 8)	No. retrieved reflections	Every read	Read
Find	Filters all nodes in the forest, given a query. This does not attend to children of concrete nodes.	No. nodes in forest up to content front	Every read	Read

Common Operations and Variant-Specific Extensions

- Across all variants, only four operations are consistently used: `Reflect`, `Score Usability`, `Compress`, and `Find`. Flat variants do not perform any operations related to chunking. This includes both the chunking operations themselves (`Chunk (Merge)`, `Chunk`

(Summarize)) and the operations required to identify chunk candidates (Group, Match (Equivalence), Match (Part-Whole)).

- A-MEM variants delegate a number of operations to their embedding model – namely Find, Group, or any matching operations. However, they introduce an specific A-MEM operation for internal indexing and validating memory links (cf. introduction of A-MEM in Section 3.3).
- The applicant of our WM system, GoT, has its own operation to Generate a set of new thoughts, and to Aggregate a set of thoughts.

To provide a visual intuition for these operational differences across reasoning steps, Figures 28 to 31 show mean input token histograms per reasoning step, across both tasks for all four variants. Table 4 complements this by summarizing the typical token usage scales (in thousands) per read and write operation, based on upper-bound observations from the histograms. As in the remainder of this evaluation, we generally distinguish token consumption across the three lanes shown in each histogram: (1) tokens consumed by the application of WM within GoT, (2) tokens used during writing to WM, and (3) tokens used during reading from WM.

Disclaimer I: Likely Insufficient Retrieval in A-MEM Variants At this time, we should note that it is clearly visible (in Figure 28 and 29) that A-MEM variants generally fail to retrieve anything for the most part (in fact, nothing at all for the ToT methods as seen in Appendix A.1, and only for aggregation-related queries in the GOT methods). This points to our similarity threshold lacking robustness, in line with our observations of memory content (Section 5.3.3). Therefore, and given the general instability of accuracy (cf. Section 5.3.2), we interpret the results optimistically in regard to A-MEM variants applying too few operations. That said, it is equally plausible that the opposite is true.

Disclaimer II: Bug in `Compress` Additionally, while writing this section, we discovered a duplication bug in the implementation of Compress. To reiterate: Compress receives a subset of all nodes in the forest that are deemed relevant from Find. This excludes children of concrete nodes apriori, so that only abstract roots, concrete roots, or concrete children of abstract roots may be returned. Within this set however, we accidentally rendered each node into its subtree individually,

without filtering first. This introduces a duplication bug, since an abstract node's children can be rendered twice: Once for themselves, and once in the context of their parent. As such, the mass of input tokens used for `Compress` should regarded as too high for the tree variants. The flat variants are unaffected, since they only contain concrete roots.

Regarding functional implications, this may only impair the application of tree variants partially, as duplicate and hence distracting prompt content may only reduce the quality of the output produced by `Compress`, but not necessarily overwhelm the downstream application with too much distracting content.

In our subsequent analysis, we will only cautiously assess the costs of `Compress` itself, while treating the distortion introduced to the relative cost-shares between other operations as negligible in comparison.

5.3.4.2 Temporal Inspection

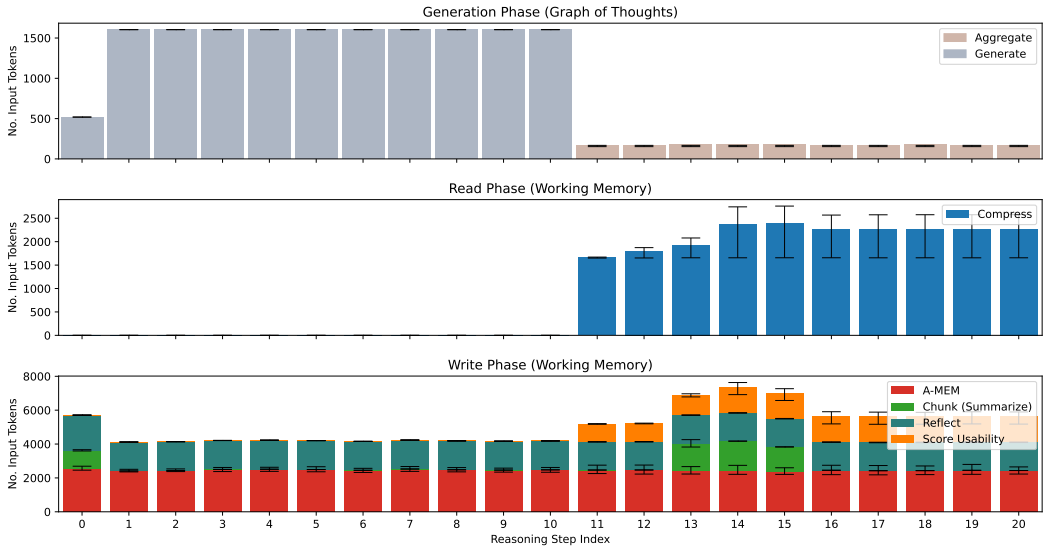
We provide a visual analysis of reading and writing behaviors based on the histograms in Figures 28 to 31. Additionally, we compare these patterns with the average memory pressure (Figure 27b) to approximate when capacity may have been reached. However, since memory pressure is plotted *across tasks* of differing lengths, this comparison must be treated with caution and should be considered only supplemental.

Table 4
Qualitative comparison of typical mean input token scales (in thousands) per read/write operation. Values are upper-bound estimates based on observations from Figures Figures 28 to 31, intended to convey relative scale only.

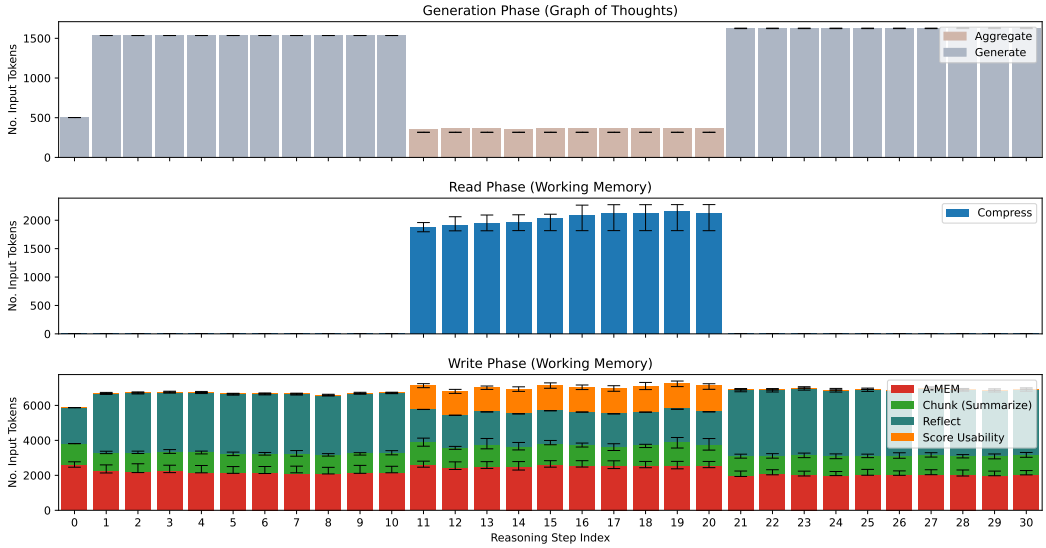
Method Variant	Read	Write
LLM (Trees)	20	30
LLM (Flat)	10	8
A-MEM (Trees)	2	8
A-MEM (Flat)	3	6

A-MEM Variants Writing costs in A-MEM variants remain relatively stable, with slight variations introduced when reading is triggered during the aggregation phase in GoT, which in turn prompts usability scoring in the subsequent write step. The A-MEM Tree variant consistently applies `Chunk (Summarize)` operations in the Set-Intersection task but appears to struggle in Sorting (Figure 28). Once reading begins, A-MEM Flat shows an upward trend in token usage for `Compress` (Figure 29), signaling that more material is being retrieved with each step. While this may resemble a stable

Figure 28
*Histograms of mean input tokens per reasoning step with interquartile ranges (IQR) for the chunking A-MEM based WM variant (**A-MEM Trees**) across both tasks, via the `GOT` method. **Caution**: Duplication bug in "Compress" increased its number of input token unnecessarily (see Section 5.3.4.1).*

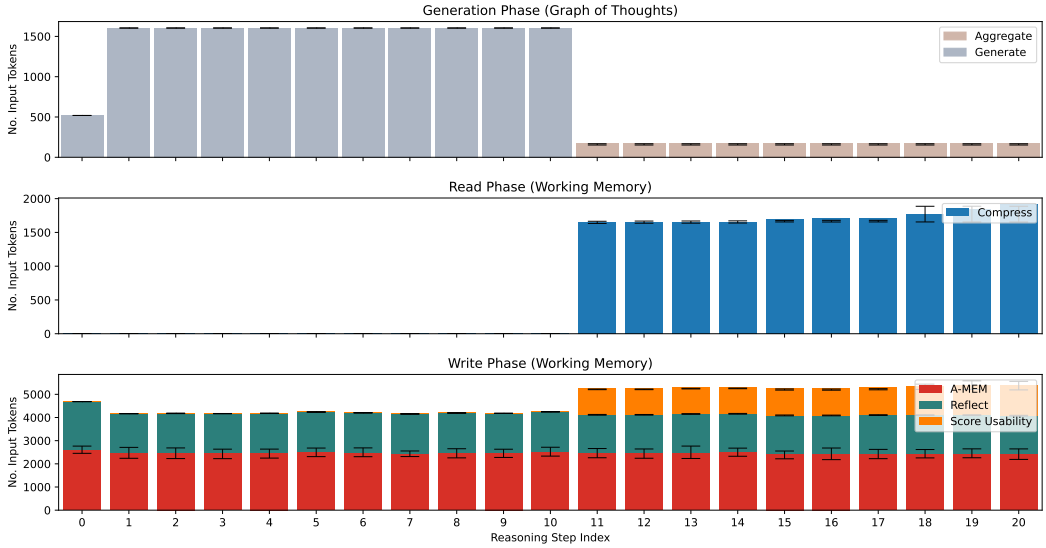


(a) *Task: Set-Intersection*

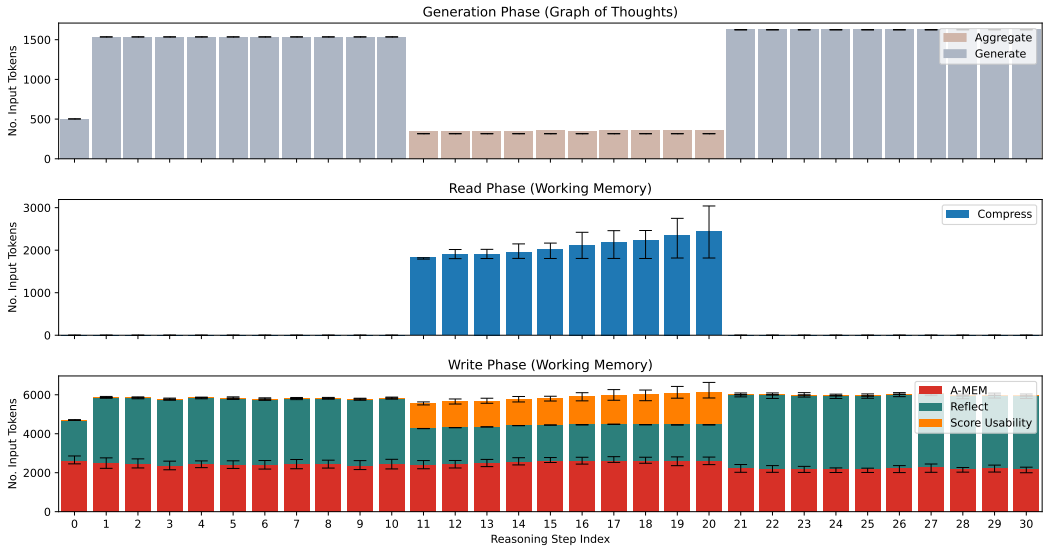


(b) *Task: Sorting*

Figure 29
*Histograms of mean input tokens per reasoning step with interquartile ranges (IQR) for the non-chunking A-MEM based WM variant (**A-MEM Flat**) across both tasks, via the `GOT` method.*



(a) *Task: Set-Intersection*



(b) *Task: Sorting*

plateau overall – especially given that the buffer has likely reached capacity by the time reading is triggered[104] – the variability remains too high to support reliable conclusions from histogram data alone.

LLM Variants In LLM variants, the pattern is more pronounced. Token consumption generally increases during the initial reasoning steps as the buffer "warms up," then stabilizes (for reading *and* writing) in the LLM-Flat variant (Figure 31) – suspiciously close to the sixth reasoning step, where memory capacity appears to be reached on average (cf. Figure 27b). In contrast, the tree variant tends to reach capacity near the end of the trial, which may account for the slight upward trend observed.[105] Overall, the different points at which capacity is reached – combined with the clear stabilization observed in the flat variant – strongly suggests that the observed stabilization is driven by capacity limits, as one would expect.

Notably, and more reliably across all LLM variants, we observe a sharp drop in `Compress` costs precisely when the `Aggregate` operation is first triggered, followed by a gradual return to earlier levels. This suggests that the WM system responds to differences in reasoning operations and can selectively retrieve and accumulate relevant material.

For the writing phases, a similar drop is observable in parts, though less pronounced. In the LLM-Flat variant, it appears only in one of the tasks – possibly due to differences in the size or number of reflections across GoT operations.[106] For the LLM-Tree variant, a drop is also visible in Sorting. In Set-Intersection, however, the opposite occurs: We observe slightly increased activity in matching, chunking, and grouping operations (Figure 30a), indicating that newly retrieved content may be actively organized. [107] Taken together, these observations tentatively suggest that novel content may trigger heavier structural processing in tree-based variants, potentially to enable more efficient retrieval. While this behavior aligns with expectations for a storage system that actively maintains an optimized index, the comparison based on histogram data alone remains inconclusive.

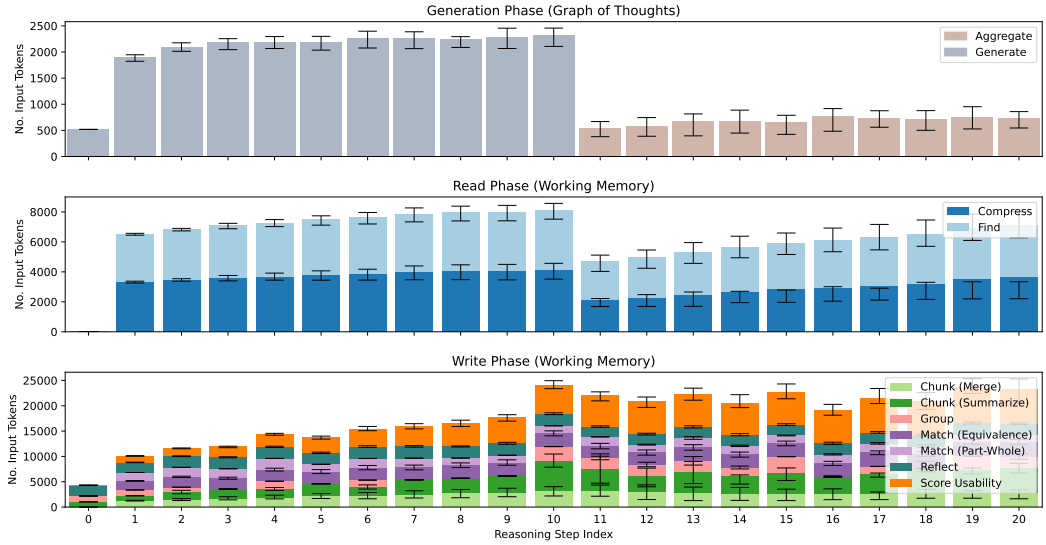
[104] Around the seventh reasoning step on average; while reading typically begins only around the tenth (cf. Figure 27b).

[105] This marks a limit in comparing with memory pressure plots (Figure 27b), as the trials differ in length.

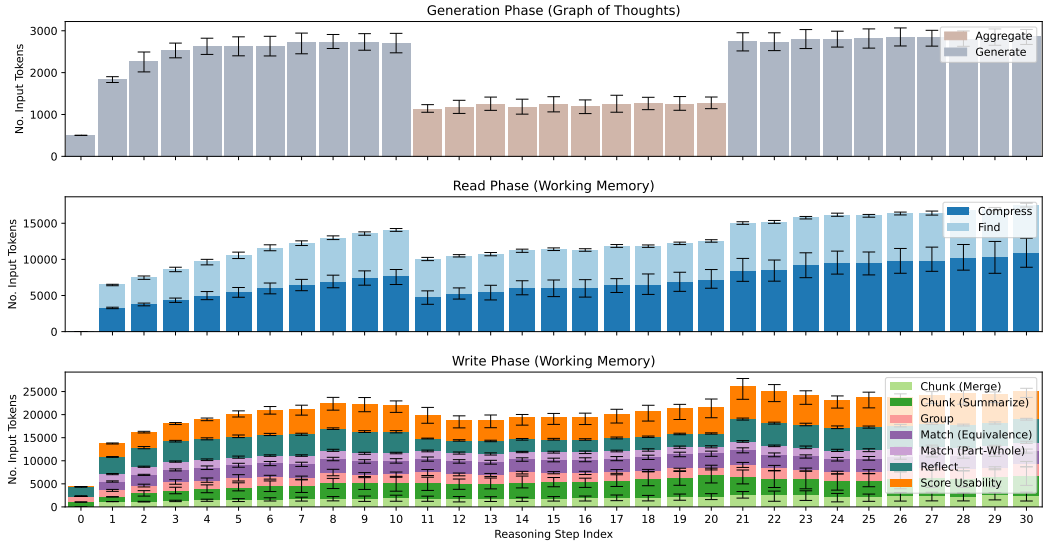
[106] As shown in Figure 31, only `Reflect` and `Score Usability` are used, but the drop is clearly visible in the Sorting task, while minimal in Set-Intersection. This suggests that GoT operations may differ significantly in the size or number of reflections per task.

[107] Alltough this increase is already showing one step prior to the `Aggregate` operation being triggered (Step 10); which could also be an outlier compared to the prior steps, where this activity is comparatively small. Additionally, once `Aggregate` is triggered, usability scoring also increases in activity, indicating that more material has been retrieved in prior steps.

Figure 30
*Histograms of mean input tokens per reasoning step with interquartile ranges (IQR) for the chunking LLM-only WM variant (**LLM-Trees**) across both tasks, using the `GOT` method. **Caution**: Duplication bug in "Compress" increased its number of input token unnecessarily (see Section 5.3.4.1)*

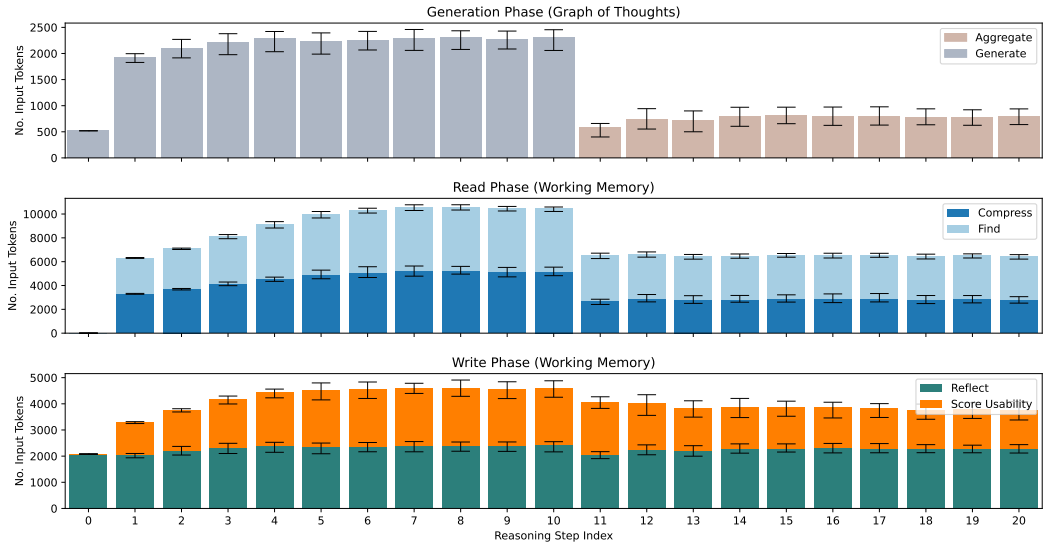


(a) *Task: Set-Intersection*

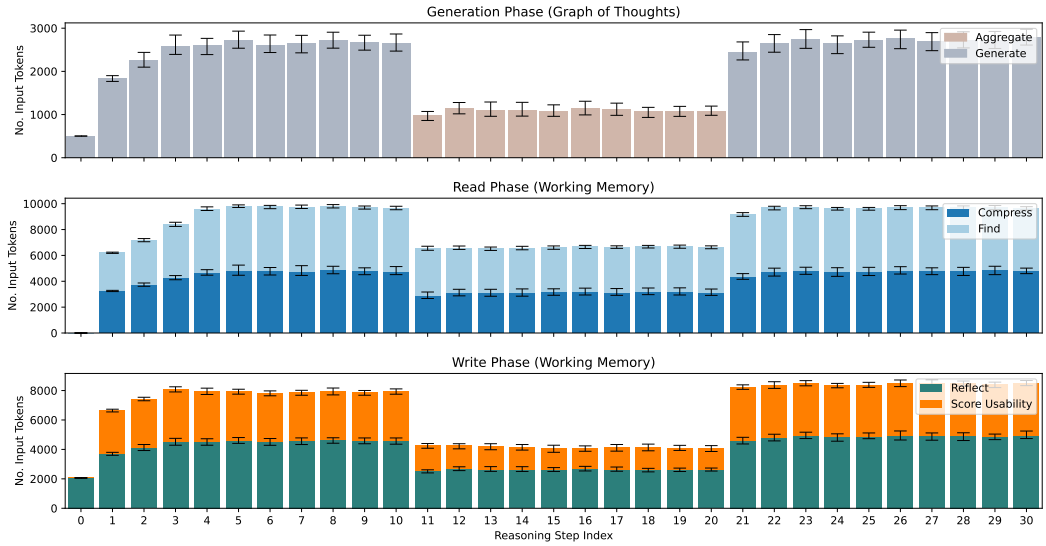


(b) *Task: Sorting*

Figure 31
*Histograms of mean input tokens per reasoning step with interquartile ranges (IQR) for the non-chunking LLM-only WM variant (**LLM-Flat**) across both tasks, via the `GOT` method.*



(a) *Task: Set-Intersection*



(b) *Task: Sorting*

5.3.4.3 Scale and Total Cost

We first focus on total cost and mean cost scales between variants. A qualitative comparison of typical input token scales is summarized in Table 4, and the overall totals can be found in Table 5. We do not refer to them constantly but use them as global references for this Subsection.

Table 5
*Total number of input tokens (in **millions**) per operation using the `GOT` method across all tasks and working memory variants (n = 120; AM = A-MEM, F = Flat, T = Trees). "Compress" for Tree variants is compromised due to a duplication bug discovered in hindsight.*

Operation	Overall	Flat	Trees	AM (F)	LLM (F)	AM (T)	LLM (T)
App (GoT; Besta et al. (2024b))							
Aggregate	2.9	1.4	1.5	0.3	1.1	0.3	1.1
Generate	15.0	7.4	7.5	2.9	4.5	2.9	4.6
Total	*17.8*	*8.9*	*9.0*	*3.2*	*5.6*	*3.2*	*5.7*
Read (WM)							
Compress	29.3	12.2	17.1	0.3	11.9	~~0.3~~	~~16.8~~
Find	26.9	12.7	14.2	–	12.7	–	14.2
Total	*56.2*	*24.9*	*31.3*	*0.3*	*24.6*	*0.3*	*31.0*
Write (WM)							
A-MEM	26.8	10.5	16.3	10.5	–	16.3	–
Chunk (Mer.)	11.6	–	11.6	–	–	–	11.6
Chunk (Sum.)	8.6	–	8.6	–	–	1.2	7.4
Group	1.2	–	1.2	–	–	–	1.2
Match (Eq.)	11.6	–	11.6	–	–	–	11.6
Match (P.W.)	2.6	–	2.6	–	–	–	2.6
Reflect	35.9	17.9	18.0	7.7	10.2	7.7	10.3
Usability	22.8	7.1	15.8	0.2	6.9	0.2	15.6
Total	*121.2*	*35.5*	*85.7*	*18.4*	*17.0*	*25.5*	*60.2*

LLM-based variants are generally the most token-heavy, whereas A-MEM variants tend to consume fewer tokens due to their embedding-based backbone. Within these categories:

Write Phase Flat variants generally use fewer tokens for writing than tree variants, as they insert reflections directly and do not perform chunking:

- **LLM variants** show this behavior, where the flat variant writes on a mean scale of up to 5–8k input tokens per operation and totals at 17M tokens. The tree variant, in contrast, goes up to 30k on average and totals at 60.2M.

- **A-MEM variants** likewise. The flat variant writes up to a 6k tokens on average and totals at 18.4M. Interestingly, we see the overhead of `A-MEM` operations leading to A-MEM-Flat outweighing LLM-Flat in total cost. The tree variant exhibits higher spikes (up to 8k on average) but totals at only 17M. Notably, this is despite the A-MEM based variants omitting from using the relatively costly `Score Usability` operation most of the time.

Read Phase Reading occurs on empirical mean scales of up 10–20k between LLM-Flat and LLM-Tree, and a mere 2–3k for A-MEM based variants, demonstrating clear token-efficiency advantage of embedding-based backbones, with retrieval cost only attributable to `Compress`. Yet, we remain cautious, in light of the overall indication of retrieval failure expressed earlier.

For LLM-Tree specifically, we expect the total cost for `Find` to be lower then that of LLM-Flat, since LLM-Flat cannot subsume concrete nodes via `Chunk (Merge)`.[108] However, we are presented with the inverse scenario: With costs for `Find` totaling at 12.7M for the flat and 14.2M for the tree variant. This contradicts our assumption, suggesting that the number or size of inserted reflections themselves may vary significantly – warranting further investigation beyond our current focus. [109]

5.3.4.4 Relative Costs

We discuss relative costs of individual operations, mainly referring to Tables 6 and 7, which respectively report the mean token cost and the relative input share per operation, providing complementary perspectives. Again, we do not refer to them constantly but use them as global references for this Subsection.

[108] To reiterate: Tree variants can subsume concrete nodes via (via `Chunk (Merge)`), while also holding abstract nodes – that only contain brief descriptors and are therefore negligibly small to the full reflections stored in concrete nodes. In contrast, flat variants cannot subsume any nodes and only store concrete roots. The `Find` operation processes the entire forest, except for subsumed concrete nodes (which are considered practically identical to their parent and are therefore excluded). Therefore, only the Tree variants can reduce the number of searchable items significantly.

[109] To elaborate: If we assume the mass of abstract nodes to be negligible, logically, the total mass of input tokens for the `Find` operation in the tree variant should only outweigh that of the flat one, if the flat variant contains significantly fewer items. Per definition of our capacity, the number of allowed items is exactly the number of items maximally retrieved by `Find`. If both variants therefore hold the same number of items, **assuming equally sized payloads (i.e. reflections)**, there should be no difference. Only a tree variant may be able to reduce this load, by subsuming concrete nodes. According to 11.6M tokens used by the tree variant for `Chunk (Merge)` (Table 5), this reduction is occurring relatively often. Therefore, we conclude that this contradiction likely stems from our assumption of uniform number or size of reflections.

Table 6
*Mean number of input tokens (in **thousands**) per operation using the `GOT` method (n = 120; AM = A-MEM, F = Flat, T = Trees). **Caution**: Duplication bug in "Compress" is distorting results for Tree variants.*

Operation	Overall	Flat	Trees	AM (F)	LLM (F)	AM (T)	LLM (T)
App (GoT; Besta et al. (2024b))							
Aggregate	0.6	0.6	0.6	0.3	0.9	0.3	1.0
Generate	1.9	1.9	2.0	1.5	2.4	1.5	2.4
All	*1.3*	*1.3*	*1.3*	*0.9*	*1.6*	*0.9*	*1.7*
Read (WM)							
Compress	3.4	3.0	3.8	2.0	4.0	2.1	5.6
Find	2.2	2.1	2.4	–	4.2	–	4.7
All	*2.8*	*2.5*	*3.1*	*2.0*	*4.1*	*2.1*	*5.2*
Write (WM)							
A-MEM	1.1	1.2	1.1	2.4	–	2.2	–
Chunk (Mer.)	0.5	–	1.1	–	–	–	2.2
Chunk (Sum.)	1.2	–	2.3	–	–	1.1	3.5
Group	0.5	–	1.0	–	–	–	2.0
Match (Eq.)	0.7	–	1.4	–	–	–	2.8
Match (P.W.)	0.4	–	0.8	–	–	–	1.7
Reflect	2.9	2.9	2.9	2.5	3.3	2.5	3.3
Usability	2.6	1.9	3.3	1.3	2.5	1.4	5.2
All	*1.2*	*2.0*	*1.7*	*2.1*	*2.9*	*1.8*	*3.0*

Read Phase For A-MEM, which does not use `Find`, `Compress` trivially accounts for 100% of the reading cost.

A first glance at the LLM-based variants suggests that `Compress` and `Find` each account for roughly half of the total reading token cost. This would suggest that the mass of *pre*-filtered items may be approximately on par with that of the subsequently compressed, indicating that only a limited number of items is being filtered. However, a closer look at total costs reveals that `Compress` lowers the relative share in reading cost, but only by 7% for the flat variant.[110] A separate log file analysis revealed that the LLM filtered out, on average, 19.45% of items for the LLM-Flat (and 21.06% for the LLM-Tree variant). Taken together, this tentatively suggests that primarily smaller items were filtered out; however, we may only presume this for the flat variant, which is not obscured by the duplication bug previously disclaimed.

[110] And increasing it by 18% for the tree variant, which is representative of the duplication bug, since the total mass of tokens filtered could not enlarge otherwise. See absolute values for Compress/Find in Table 5: $11.9/12.7 \approx= 0.93$ and $16.8M/14.2M \approx 1.18$, for LLM Flat and LLM Tree respectively.

Table 7
Share of input tokens per operation, relative to the respective WM variant (column). `GOT` method, averaged across all tasks (% rounded; AM = A-MEM, F = Flat, T = Trees). ***Caution****: Duplication bug in "Compress" is distorting results for Tree variants.*

Operation	Overall	Flat	Trees	AM (F)	LLM (F)	AM (T)	LLM (T)
App (GoT; Besta et al. (2024b))							
Aggregate	1%	2%	1%	1%	2%	1%	1%
Generate	8%	11%	6%	13%	10%	10%	5%
All	9%	13%	7%	15%	12%	11%	6%
Read							
Compress	15%	18%	14%	1%	25%	1%	17%
Find	14%	18%	11%	–	27%	–	15%
All	*29%*	*36%*	*25%*	*1%*	*52%*	*1%*	*32%*
Write							
A-MEM	14%	15%	13%	48%	–	56%	–
Chunk (Mer.)	6%	–	9%	–	–	–	12%
Chunk (Sum.)	4%	–	7%	–	–	4%	8%
Group	1%	–	1%	–	–	–	1%
Match (Eq.)	6%	–	9%	–	–	–	12%
Match (P.W.)	1%	–	2%	–	–	–	3%
Reflect	18%	26%	14%	35%	21%	27%	11%
Usability	12%	10%	13%	1%	15%	1%	16%
All	*62%*	*51%*	*68%*	*84%*	*36%*	*88%*	*62%*

Write Phase In the write phase, variant differences are more pronounced, as A-MEM variants offload many operations to their embedding backbone. Overall:

- `Reflect` is the most expensive read operation, with 18% overall cost share, 26% in flat variants, and 14% in tree variants. It represents a general bottleneck, as it is invoked every time a new set of reflections is created.
- `Scoring Usability` is also costly – making up 12% overall, 10% in flat, and 13% in tree variants. This is significant, given its (potential) use after each write.[111]
- A-MEM's internal memory operation accounts for roughly half of all write-related tokens in A-MEM variants and contribute a comparable portion relative to all operations in the LLM-flat variant.

[111] Note that `Score Usability` depends on the number of reflections used in the prior read step. In both A-MEM variants, where almost no items are retrieved, usability scoring is rarely triggered. This results in a low share of 1%, that would be higher if more reflections were available.

- Interestingly, in the LLM-tree variant, the operations `Chunk (Merge)`, `Match (Eq.)`, and `Reflect` contribute nearly identical shares to the overall cost, while while `Group` and `Match (Part-Whole)` only contribute negligibly in comparison.

5.3.4.5 Conclusion

We highlight the following key insights from the above, partially attending to our core hypothesis established in Section 5.3.1:

- Obscuring Hypothesis III:
  Retrieval in A-MEM based variants appears partially broken (retrieves nothing in most cases; cf. Figure 28 and 29). This points to an overconfident reliance on A-MEM's internal validation logic, on which we define our similarity threshold that appears to be lacking robustness. This is in line with empirical observations of memory content (Section 5.3.3). As such, Hypothesis III can likely not be assessed as reliably.
- In tentative support of Hypothesis III:
  In contrast to A-MEM variants, LLM variants consistently exhibit a sharp drop in `Compress` costs when the `Aggregate` operation is first triggered, followed by a gradual return to prior levels (cf. Figure 31 and 30). This indicates that the system may adapt to the reasoning context while selectively retrieving and accumulating related material. While, additionally, write behavior appears to change *in the same intervals* as well, histogram data alone leaves no firm concision to draw in this specific regard.
- Contrary to our expectations, the LLM-Tree variant has higher read costs than LLM-Flat (cf. Table 5). We assume this to likely stem from a non-uniform number or size of reflections; but cannot conclude reliably. Hence, further investigation would be needed.
- A minor finding: The `A-MEM` operations account for about half of all WM write operations in A-MEM variants; making A-MEM-Flat more expensive than LLM-Flat during writing (Table 5) despite only rarely scoring prior reflections.

However, we highlight the overall clear token-efficiency advantage of embedding-based methods. Accordingly, a hybrid approach could strike a favorable balance – leaving only selected, high-value operations to the LLM. (further discussed in Section 6.2.1.3).

Comparing the different systems:

- **Compared to A-MEM** (W. Xu et al., 2025) as a competing memory framework for reasoning, we observe that our WM wrapper consumes noticeably more tokens, while also introducing costly read operations. Specifically, read operations consume a total of 0.3M tokens in A-MEM-based variants versus 24.6–31M in LLM-based WM, while write operations range from 17–60.2M in A-MEM compared to 18.4–25.5M in our WM variants (cf. Table 5)
  Still, we find reason for cautious optimism in the latter comparison, as LLM-Flat already outperforms A-MEM-Flat in total write cost (cf. summary above), and the increase in cost is only twofold compared to standalone A-MEM (see above).
- **Compared to GoT itself** our WM system introduces drastic overhead: In the most expensive configuration (LLM-Trees), GoT's `Aggregate` and `Generate` steps consume just 5.7M input tokens, while the WM layer adds 91.2M on top. Across all variants, GoT still accounts for only 9% of total input tokens (cf. Table 5).[112]

However, we note that matching GoT's token cost is not our design goal: GoT's costs scale linearly ($\Theta(n)$), generating each thought from a bounded set of predecessors – the very limitation our approach seeks to overcome. We aspire toward reasoning LLMs, which scale quadratically in reasoning growth ($\Theta(n^2 + n)$). As such, we believe there is reason to be optimistic. Our prototypes were not built with a focus on token efficiency, but exploration. Further, with rigorous optimization, even within our existing variants, we believe substantial savings could be realized. For a brief discussion on optimization opportunities for our concrete setup, see Section 6.2.2.

Finally and in hindsight, we acknowledge a minor limitation of our analysis to lie in the varying number (and possibly size) of reflections, which we did not normalize for.

[112] At this point, we remind the reader that the tasks implementation in GoT, does not employ LLM-based scoring (cf. Section 5.1). If it did, its cost would increase by an estimated ≈ 0.99M input tokens, or 11% of its current LLM budget. Our estimate, in the context of the `GOT` method and all LLM variants, is derived as follows:
- Cost per scored reflection: 47.263 reflections used 22.5M input tokens → 22.5M/47.263 ≈ 476 input tokens per scored reflection
- GoT uses 420 `SCORE` operations. Each scores on average 7.28 thoughts; conservatively assume 5 thoughts scored per operation (accounting to score multiple at once). → 420 × 5 = 2.100 scored thoughts in total
- Assuming equivalent token cost per scored thought and scored reflection → 2.100 × 476 = 999.600 tokens.

Chapter 6

Limitations

In this Chapter, we elaborate practical limitations encountered throughout this thesis, each of which implies targeted opportunities for future work within the scope defined by our background material (Chapter 3). In contrast, the following discussion in Chapter 7 will explore broader conceptual implications and research directions that extend beyond this scope.

We attend horizontally to all major components of our work: The design of the working memory (WM) core architecture (Section 4.3), the implemented WM variants (Section 4.4), our experiments (Chapter 5), and finally, design-relevant experiments that were identified but not conducted.

6.1 Limitations of the Core Architecture

The following limitations concern architectural decisions made in operationalizing cognitive concepts from the Multi-Component Model (MCM) of WM (Baddeley, 2000; Baddeley & Hitch, 1974; Baddeley & Logie, 1999; Hitch et al., 2025), as introduced in Section 3.3.1.2. While, in the design of the architecture, we already outline design provisions for extending toward out-of-scope MCM components (Section 4.3.5), the present Section focuses on the *in*-scope limitations. That is, concrete design choices that were implemented but later revealed tension.

Non-Episodic Episodic Buffer

In the MCM, the Episodic Buffer is said to be episodic in regard to time and space, which however, appears underspecified in the literature (cf. Baddeley (2000) and Hitch et al. (2025)). Critically though, (as noted in Section 4.2.2), temporal and spatial structure can be defined in various ways – and may be highly dependent on context.

However, as briefly mentioned in the introduction of the *AbstractEpisodicBuffer* (Section 4.3.4), we refrained from the integration of an accordingly arbitrary "episodification" mechanism. Specifically, while an abstract grouping function has been implemented to cluster items – and temporal relevance is indirectly supported via decaying survival scores – there is no mechanism for grouping or ordering items based on flexible task context; the current approach is limited to static, content-based grouping without support for conditioning on an external query.

Notably, such a context signal could go beyond the immediate thought or task and reflect broader structures in reasoning – for example, previous goals, decision stages, or thematic shifts across a session.

We see this as a limitation of our design, uncovered in hindsight, since the main characteristic of the episodic buffer – its episodicness – is not explicitly modeled, and leave it up to future work to explore.

Non-Interleaved Item Trees

As previously specified (in Section 4.3.4), the item structure of the *AbstractEpisodicBuffer* is a tree of arbitrary degree. While the general nature of chunks remains debated in cognitive science, their hierarchical character appears evident (see Section 3.3.1.1). However, some computational models in the field also operate on more flexible, network-like structures rather than explicit trees (cf. Burgess and Hitch (2006)) – yet, these models do not explicitly represent "chunks" as distinct units, as our approach attempts to do.

However, in our setting, the `AbstractChunkNode` may inhabit real practical limitation: It may only have a single parent node. This restriction may limit the buffer's expressiveness, as it prevents flexible clustering where an item could naturally belong to multiple groups, or tree structures in which levels may be skipped. As a result, more complex hierarchies may be required to simulate shared membership, potentially introducing unnecessary depth and structural overhead.

Future work may thus explore a synthesis of graph and hierarchical structures to enable more flexible forms of interaction.

6.2 Limitations of the Implemented Variants

We discuss the limitations arising from the concrete WM variants implemented. These reflect design trade-offs, simplifications, and deferred generalizations made during development, as well as insights gained during our evaluation in Section 5.3. The variant designs are presented in Section 4.4, and their integration into Graph-of-Thought (GoT; Besta et al. (2024b)) is detailed in our experimental setup (Section 5.3.1).

The remainder of this section is divided into high-priority foundational limitations (Section 6.2.1), and suggestions on concrete optimization steps within the existing setup (Section 6.2.2).

6.2.1 Foundational Limitations

We outline fundamental design limitations inherent to all variants implemented (in Section 4.4): Specifically, we elaborate the limits on the integration of Self-Reflection (Subsection 6.2.1.1), the wiring mechanism of tree-based variants (Subsection 6.2.1.2), and foundational limitations between LLM-only and embedding-hybrid storage backend (Section 6.2.1.3).

6.2.1.1 Foundational Limitations in the Integration of Self-Reflection

Missing Task-Specific Scores As has been illustrated in Section 4.4.4, we limited the diversity of reflection feedback to preserve task-agnosticism. Specifically, this approach precluded the integration of task-specific scoring schemes, which have shown to be beneficial (Madaan et al., 2023).
Although we attempt to address this through general-purpose item fitness scores originating from item usage (see Section 4.3.3), targeted scores may not only capture item fitness more easily, but – importantly – they enable stabilization in cases where a reflection improves one quality aspect while simultaneously degrading another (Madaan et al., 2023).
This may potentially hint at a broader opportunity: Diverse scoring signals could not only improve reflection quality, but also play a role in shaping targeted forgetting mechanisms. However, this would then raises the question of how such scores should be defined and compared across tasks with fundamentally different objectives.

Minimal Reflection Granularity As introduced in Section 4.4.4, we limited our scope to restrict the reflectable material to the local thought context: When incorporating reflection into a new thought to be generated, we do not include the previously reflected thoughts, or a window of such. Herewith, we effectively dismiss insights from prior work, that has shown that providing both the reflection and the entire reflected on trajectory (i.e. the *entire* CoT to the root, and not only the erroneous reasoning steps) can improve performance (Shinn et al., 2023).
In hindsight, we could have alternatively prompted the model to include the full line of reasoning (within a single thought) leading up to the error, rather than isolating only the erroneous step itself. This might have produced more informative reflections and yielded closer alignment with prior findings, but was not considered at the time.
We therefore regard this as a promising direction for future refinement and have designed our core architecture in anticipation of scenarios where reflectable material extends beyond the granularity of a single thought.

6.2.1.2 Foundational Limitation in the Writing Mechanism

In the design of our write transaction (Section 4.4.5) for hierarchical memory variants, we follow our scoping (cf. Section 4.4.1), in deliberately excluding dynamic re-organization of memory, and the formation of higher-order hierarchical clusters.

Accordingly, our insertion strategy operates under the simplifying assumption that the buffer maintains an (approximately) optimal clustering state at all times, and greedily commits to the first valid structural match.
In addition, we made the design choice to prioritize the merging of duplicates, and the incorporation into existing clusters over the formation of new ones. While this avoids fragmentation, it may lead to *over-aggregation*, where material is forced into existing trees. As a result, conceptual boundaries may blur, and *early misgroupings remain uncorrected* due to the absence of re-clustering. In contrast, a strategy that favors the formation of new clusters could better preserve semantic separation, albeit being prone to fragmentation. However, in this case, widening the scope to consider deeper hierarchical structures, could allow for later consolidation that could subsume fragments of duplicate or overlapping clusters, which may lead to an efficient re-organization of memory.

Additionally, we empirically show in our qualitative evaluation (Section 5.3.3) that over-aggregation may be avoided if relationships are defined strictly. However, this, in combination with refraining

from re-organization, contradicts with our goal of being task agnostic: If a WM is to be maximally task-agnostic (or rather, even *case*-agnostic) it cannot enforce a taxonomy that is both flexible to generalize across *every conceivable case of any task*, while also being strictly defined a prior – at least according to our limited experience.[113].

As such, we consider both the extension of our scope to embrace dynamic reorganization, deeper hierarchical structures, and further experimentation with the alternative insertion strategy (that favors new clusters) as key opportunities for future refinement.

6.2.1.3 Foundational Limitations of Implemented Variants

The General Tension between LLM and Embedding-Hybrid Variants While our quantitative results (Section 5.3.2) remain unstable, our qualitative evaluation (Section 5.3.3) suggests that, in contrast to embedding-based approaches, LLMs may be key to retaining a task-agnostic writing mechanism, as they appear more flexible in discerning clustering relationships. Therefore, in our current view, LLMs function as a "general-purpose engine for data *management*", as described by Zhou et al. (2025), and should be responsible for operations that require maximum flexibility.

However, this potential may stand in tension with the substantial token cost. As laid out in our evaluation of token consumption (Section 5.3.4), embedding-based approaches offer a clear advantage in this regard. At the same time, we are concerned that they could introduce additional rigidity, thereby reducing agnosticism by requiring task-specific fine-tuning procedures. Accordingly, we propose that future work should critically assess which operations qualify as both routine and task-agnostic, and may therefore be suitable for delegation to lightweight embedding models.

This stands in close connection with the following limitation.

The Need for a Robust Embedding-Based Baseline Our embedding-hybrid variant integrates A-MEM (W. Xu et al. (2025); Section 3.3.2.1), a state-of-the-art memory system designed for LLMs. To support retrieval beyond a fixed neighborhood size, we leverage A-MEM's internal validation logic to derive a dynamic similarity threshold without independent calibration (details in Section 4.4.6.2).

[113] Specifically, we see two degrees on which a general ontology, for that matter, could be strict here: First, by defining a set of strict relationships (that may still be assigned flexibly), and second, by also defining a strict taxonomy in which every layer is explicit separated.

Originally intended as a competitive embedding-based baseline, the A-MEM variants exhibited noticeably weak retrieval performance compared to their LLM-based counterparts (see Section 5.3.4). While we cannot conclude with certainty (due to our unstable evaluation; discussed in Section 6.3), this may suggests a lack of robustness in our integration of A-MEM.
As such we emphasize, in the strongest terms, that our embedding-hybrid variant must not be interpreted as a robust baseline for comparing LLM-based retrieval against embedding-based approaches. We did not fine-tune a domain-adapted embedding model, nor did we calibrate or train a regression head to learn task-specific similarity thresholds (cf. Schopf et al. (2024)). Instead, we relied entirely on A-MEM's internal logic – a design shortcut that reflects a deliberate scoping choice.

Crucially, task-agnosticism is a central design goal throughout this work, and as such, stands in tension with the idea of one-time supervised fine-tuning for a specific downstream task. However, as previously mentioned, it is highly plausible that an embedding model fine-tuned not to a task, but to general-purpose memory operations, could be favorable over LLM-based methods. Thereby we reassert our prior suggestion, to carefully distinguish which memory operations truly require the full flexibility of LLMs (as indicated in our qualitative evaluation, Section 5.3.3), and which can be effectively encapsulated into specialized components via do-it-once fine-tuning, without compromising task-flexibility.

In this connection, we stress the need for a solid and properly controlled embedding-hybrid implementation. Without such a reference point, comparisons could risk overstating the apparent flexibility of LLMs-only solutions. Hence, while our findings tentatively point in that direction, they should be interpreted with caution in the absence of a stronger baseline.

6.2.2 Optimization Potential

The following limitations highlight targeted opportunities for reducing the input-token costs of our implemented WM variants. As shown in our evaluation of input token consumption (Section 5.3.4), *all* implemented WM variants consume a staggering number of input tokens, in relation to the system they are applied to, GoT; ranging approximately 6 to 16 times more, depending on the variant.

As such, not all variants are equally affected by token-related limitations. Most notably, the A-MEM-based variants offload a significant portion of memory operations to their embedding backbone, thereby reducing token overhead. For a comprehensive overview of memory operations, we refer the reader to the detailed comparison provided in our token consumption evaluation Section 5.3.4.1.

Note on Comparability to GoT Token Usage:

We note that the observed token discrepancy between our WM-enhanced variants and GoT (cf. Table 5) should be interpreted with caution:

In vanilla GoT, each thought is generated independently from a bounded set of predecessors. This results in a total input cost that grows linearly with the number of thoughts:

$$n \cdot t \cdot d_{\text{in}} \in \Theta(n)$$

where n is the number of thoughts, t is the average number of tokens per thought, and d_{in} is a small constant, the maximum node input degree (e.g., 1–3 in practice).
In contrast, our WM-integrated setup aspires towards reasoning-LLM behavior, where each new "thought" (i.e. token) is generated conditioned on all previous ones, leading to cumulative context growth. The total input token cost thus scales as:

$$\sum_{i=1}^{n} i \cdot t = t \cdot \frac{n(n+1)}{2} \in \Theta(n^2 + n)$$

which introduces an upper bound on token usage that far exceeds GoT's static cost, hence reflecting a fundamental difference in reasoning architecture.

We provide a set of potential optimizations. The majority of these can be considered jointly, as a set of combinable strategies that influence and reinforce one another.

6.2.2.1 Optimizing Integration: Number of Requests at Different Reasoning Granularity

Currently, our integration within GoT pushes the number of memory requests to its upper bound by reading from and writing to WM before and after every new thought produced by the LLM. This is likely unnecessary, depending on the use case. A more conservative strategy would be to consolidate read or write operations at the level of complete "trajectories" (as referred to in the

context of LLM agents; cf. Section 3.3.2.1), i.e. at coarser reasoning levels, corresponding to completed (sub-)Chains-of-Thought (CoTs), or entire subgraphs of the thought process.

- **Fewer Writes and Richer Reflection Context**: Consolidating multiple thoughts into a single write request could not only significantly reduce overhead, but also enable more coherent, context-rich reflection, as suggested by prior work (cf. Shinn et al. (2023)). Hence this may be considered equivalent to the prior suggestion of widening reflection granularity (Section 6.2.1.1). Depending on the use case, one might choose to write only after a CoT has been abandoned,[114] or more generally, after each scoring step. Optimizing the granularity of a "trajectory" will likely depend heavily on context and may imply a dual-objective optimization problem: (1) maximizing useful reflective context, while (2) minimizing write cycles.
- **Fewer Read Requests**: In line with the above, read requests could also be deferred to occur only at the beginning of a new trajectory – that is, immediately after the previous one has been consolidated and written. At the same time, it may be beneficial to retain the option of additional read operations within a trajectory to support targeted retrieval for specific reasoning steps.

6.2.2.2 Optimizing the Write Mechanism and Storage Format

As shown in our evaluation on token-usage (Section 5.3.4), the wringing mechanism is generally the most costly.[115]

Our current variants may be improved by the following concrete suggestions:

- **Limiting the number of reflections per thought**: As described in the design of our reflection generation (Section 4.4.4), we prompt the LLM to generate not one, but a set of distinct reflections that attend to individual reasoning errors, so that they can be represented as separate entities in memory. However, depending on the use case and granularity of the reflected material, this may not be necessary. Instead, one could simply confine the generation to a single reflection to reduce cost. In hindsight, this would also have facilitated subsequent analysis, as it would have eliminated the need to normalize for the number of reflections – a step we regrettably overlooked.

[114] Though this may limit the ability to capture insights from *successful* trajectories; see also Section 6.3.

[115] This comparison exudes the LLM-Flat variant, which has a total read/write cost ratio of 24.6M/17M tokens, but does not employ any significant storage operation during writing. In contrast, the flat and tree variants of A-MEM consume 0.3M/18.4M and 0.3M/25.5M, while the LLM-Tree variant consumes 31M/60.2M tokens, respectively. See Table 5.

- **Compress conservatively**: We leverage the `Compress` operation not only to compress a number of reflections down to at most 8, but also to re-write a reflection in the context of the given read-request. This also happens if there are fewer then 8 reflections, to encourage a target recompilation. However, this may be unnecessary and could be refrained from. In the analyzed scenario, across all variants, this would reduce the cost of `Compress` substantially, by 97.8%, directly halving the overall reading cost.[116]

The following suggestions specifically target tree-based memory variants capable of performing chunking operations:

- **Generating Summary Indices for All Root Nodes**: Currently, summary indices (i.e., short textual descriptors) are generated only for abstract nodes during chunking operations. In contrast, concrete root nodes incur substantial token overhead during tree formation, as their full payloads must be attended to. Generating summary indices for *all* root nodes – including concrete ones – could enable more efficient grouping during writing, as it would allow structural decisions to be made based on lightweight representations, without requiring full inspection of each item's content upfront; potentially enabling a staged grouping process. Crucially, the generation of summary-indices could be done directly *within* the generation of a concrete node itself (as is arguably already the case with the "insight" field in our reflection format; Section 4.4.4), and may thus not introduce any additional cost.
- **Deferring chunking operations to the end of a writing transaction**: If writing multiple reflections remains a goal, one could consider that in principle, for a single request, the insertion order of reflections should be irrelevant. In this case, one could defer the chunking operations to the end of a writing transaction, and present every item with the same memory content. This would limit the possibility of re-creating a new concrete summary or index node with every subsequent insertion,[117] while simultaneously allowing for parallelization. In the analyzed scenario, this would roughly half the costs of the `Chunk (Merge)` operation

[116]This is also a finding from a later log file analysis. For clarity: Using the `GOT` method across all WM variants, we recorded a total of 6094 compression operations, 5962 of which used fewer then 8 reflections, i.e. 97.8%. The remaining 132 invocations only contained an average of 9.11 reflections, and are therefore not substantial deviates. This makes the cost of `Compress` negligible, which accounted for roughly half of the total reading cost overall (cf. Table 5).

[117]Which of course, conversely, could also be tempted by first consolidating the obtained reflections in a preprocessing step.

in the LLM-Trees variant (reducing its total writing cost by 9%), while the cost of `Chunk (Summarize)` would remaining unchanged overall.[118]

- **Embracing semantic drift**: Finally, it may not be necessary to prevent the semantics of concrete nodes from drifting across successive mergers (see Section 6.1). While this could impact the coherence of summaries over time – a possibility that warrants further investigation and likely necessitates dynamic reorganization – it may also be seen as a form of *interference-based forgetting* (cf. Section 4.2.2), potentially aligning with principles from cognitive science. This could reduce the cost of the `Chunk (Merge)` operation by an estimated 55%, translating to a 10% reduction in writing costs for the LLM-Tree variant. [119]

6.2.2.3 Optimizing the Retrieval Mechanism

As outlined in our qualitative evaluation (Section 5.3.3), a WM may benefit most from the capabilities of the LLM by leveraging it as a "general-purpose engine for data *management*" (Zhou et al. (2025); cf. Section 6.2.1.2). In our view, this primarily concerns the writing mechanism, which was discussed in the previous section.

When considering the design of a next-generation WM variant, this costly capability may not be necessary for *retrieving* information from an already well-managed memory forest. In accordance, we highlight the following avenues for future work to explore:

- **Performing a dedicated search**: To reduce computational overhead, the current `Find` operation could be optimized to avoid expanding the entire memory forest into a single retrieval query. Instead, index representations (e.g., short summary strings) could be leveraged for early-stage filtering – identifying which trees are worth expanding into the search front – as is already the case during insertion. This staged approach would allow unpromising candidates to be filtered early, potentially reducing token cost substantially (given that a typical reflection spans far more tokens than its corresponding summary).

[118] A log file analysis revealed that this would reduce the costs of the `Chunk (Merge)` operation by a factor of 1.99 in the LLM-Trees variant (the only variant applying this operation; from 11.6M down to 5.8M tokens; a total cost reduction of 9% for writing, cf. Table 5), while the cost of `Chunk (Summarize)` would remain unchanged, over all variants.

[119] This estimate assumes the number of mergers not to increase. **For reference:** We estimate a typical merger of two reflections to contains about 1.2k tokens (While no concrete average is available to us, we simply tokenized 10 random reflections (3186 tokens), divided by 5 ($\approx$ 638 tokens), and added the cost of the prompt ($\approx$ 470), yielding a total of 1.107 which we again ceiled to the second digit). The average number of input tokens of the merger in our evaluation is 2.2k (Table 6). Hence, the former is $1.2k/2.2k =\approx 55\%$ of the latter. The LLM-Tree variant is the only one using the respective operation via the LLM, and (across all runs using the `GOT` method) has a total write cost of 60.2M tokens, 11.6M of which originate form the merging operations (Table 5), i.e. $\approx$ 19%. Thus, yielding an approximate $\approx$ 10% reduction in writing cost, when consistently only ever merging two reflections.

However, this optimization may raise a conceptual tension: Our current approach reflects the idea of working memory as the full active context, accessible all at once, and constrained only by cognitive limits such as the lost-in-the-middle effect (Liu et al., 2023). A staged or selective search could imply that only the actively traversed front constitutes working memory content, while the remainder would reside in long-term memory (LTM), to be loaded as needed. Yet, this may be actually reflective of the tight integration of LTM and WM.

We leave a discussion of the design implications for such a hybrid WM–LTM scheme to future work (see also Section 4.3.5 on extensibility with respect to LTM integration, if to pursued as separate storage module).

- **Assertive retrieval to reduce the need for compression**: Future work may experiment with only retrieving a limited set of stored reflections, and entrust them to be well synthesized for the next reasoning step. This would reduce the need for token-heavy post-processing operations – albeit the risk of reducing targeted compilation of WM material to the given request.
- **Caching request–response pairs**: In addition, depending on the variance of read requests within the specific reasoning environment, it may be feasible to cache them along with their results (though this may limit more targeted retrieval). In GoT, for instance, the number of possible request prompts corresponds to the (limited) set of available operations. While such requests may include previously retrieved reflections that introduce variance, this can potentially be mitigated through heuristic handling, as the WM system can re-identify and contextualize these reflections.[120]

[120]To elaborate: Since such reflections can be identified by the WM – as is the case for usability scoring – this variance may be mitigated through a flexible retrieval heuristic: For a request containing both a prompt and a reflection, where the prompt matches a previously cached request, the context of the included reflection (e.g., its location in the forest) could serve as an additional signal within a lightweight retrieval strategy.

6.3 Limitations of the Experimental Setup

We address limitations of the conducted experiments (Chapter 5). The overarching setup is defined in Section 5.1 and further instantiated for our experiments on Self-Distillation (SD; J. Huang et al. (2022)) and on the WM variants integrated into Graph-of-Thought (GoT; Besta et al. (2024b)) in Sections 5.2.1 and 5.3.1, respectively.

Instability

As has been elaborated in the main results of our WM experiment (Section 5.3.2), in our concrete setting, the GoT tasks adopted from Besta et al. (2024b) seem to inhabit a significant level of instability.

Given that expected differences in label quality emerged consistently in our SD experiment, and that we were still able to compare changes in the fine-tuned model's performance, this limitation did not yet pose a critical threat to the evaluation. However, we had already observed that evaluation via standard prompting restricted interpretability of the different SD variants.
This early observation proved consequential: Our primary quantitative evaluation of WM was ultimately undermined by the same underlying instability. As a result, our WM evaluation was effectively compromised, and is – in its current form – inherently limited by sample size, and by extension, by our resource constraints.

In hindsight, it becomes important to note that greater stability may have been achievable by lowering the temperature to a minimum. Regrettably, we did not pursue this setting, as the original task were implemented by sampling thoughts independently.[121] Consequently, generating a set of out-branching thoughts would have resulted in near identical outputs, defeating the purpose of parallel exploration. While the alternative strategy of jointly generating multiple thoughts could have enabled the use of a substantially lower temperature, it was not adopted since it diverged from

[121] Cf. Tree-of-Thought (Yao et al., 2023b), the predecessor of GoT, which initially proposed two distinct generation strategies for creating outgoing thought branches – both of which are adopted by GoT. In one, thoughts are sampled i.i.d., while in the other, the model is prompted to generate a set of thoughts jointly.

the original setup, which at the time, did not appear necessary and would have introduced non-trivial complexity for integrating WM, along with increased token costs.[122]

Tasks

We consider it likely that the setup of our WM experiment is fundamentally limited by the nature of the tasks it employs.

Set-Intersection and Sorting are tasks that do not require natural language understanding. LLMs, however, are pattern-based sequence models rather than symbolic reasoners. As such, they are known to perform poorly on tasks that require precise quantitative reasoning (Lewkowycz et al., 2022). Consequently, while such tasks may serve to test whether LLMs can be prompted to execute divide-and-conquer strategies – effectively, or rather *exactly*, what Besta et al. (2024b) test with GoT – they are not tasks one would reasonably entrust to an LLM natively.

Instead, standard practice is to delegate such non-linguistic operations to external tools (cf. Lewkowycz et al. (2022)), such as a Python sandbox, where sorting a list or computing a set intersection becomes a single line of code – a "sentence" or atomic unit of language that LLMs can effectively generate.

Accordingly, for the purpose of prototyping WM, a more suitable evaluation would have involved a task grounded in natural language understanding and semantic diversity.[123]

As a final note in this regard: We posit that the semantic diversity (of errors) a task admits may be a good proxy for the potential usefulness of reflection mechanisms, and by extension, of our implemented WM (see our qualitative evaluation in Section 5.3.3). Tasks that are inherently "dull" (i.e., low in semantic variation and primarily procedural) may be unlikely to benefit from WM unless paired with a dynamic planning scheme that can utilize WM to reduce contextual load (as does GoT).[124]

[122] Costs would increase since every new thought would essentially have to be generated in the context of all of its previously generated siblings. However, this is also reminiscent of increasing the granularity of the reflected-upon material (see Section 6.2.1.1) and may thus be realizable in a more coherent setup that could also bear additional synergy effects. Cf. also Section 6.3.

[123] Albeit the fact that, for us humans, WM is precisely what enables us to tackle such non-linguistic procedures beyond mere intuition. However, leveraging WM to turn LLMs into capable reasoners – even for such tasks – requires a solid prototyping phase rooted in language-centric settings.

[124] Something that lies beyond the scope of this thesis (cf. Section 4.1).

No Backtracking

Lastly, we note that the current integration of WM into GoT utilizes the default `Controller` implementation – the class responsible for scheduling operations in the Graph-of-Operations (cf. introduction of GoT in Section 3.2.1.4) – and therefore does not implement a depth-first generation of the thought graph (i.e., a generation strategy that uses explicit backtracking). As a result, thoughts are not generated linearly, as one might expect from a natural reasoning process, but rather in a breadth-first manner.

This has implications for WM application: No coherent thought trajectory can be completed before the next one is started. In setups where different chains of thought address substantially different (sub-) problems, this can result in minimal accumulation of related items in memory, as constant context switching prevents any item from being revisited frequently enough to avoid rapid forgetting.[125]

While this limitation was anticipated, we did not integrate a depth-first generation strategy into GoT, precisely because the different strands of reasoning consisted of the same operation. Admittedly, however, such an integration might have been beneficial even in this uniform setting.[126]

We therefore encourage future work to integrate WM into environments that support depth-first traversal and backtracking, allowing sufficiently coherent thought trajectories to unfold before context shifts occur at a rate that does not exceed the system's capacity to retain relevant memory.

Deferred but Relevant Experiments

To document our design rationale and open directions, we briefly list several experiments that were identified as valuable but could not be pursued due to time constraints. We see them as meaningful next steps:

[125] One may consider the analogy of performing different calculations in parallel: Frequent context shifts quickly impair the ability to maintain a coherent WM. Naturally, this effect depends on individual WM capacity, the depth of problems, and the frequency at which context switches occur.

[126] See, for example, our qualitative evaluation (Section 5.3.3), where we encountered a memory tree dealing exclusively with errors in counting the digit 4. Even if all strands involve counting, it may be advantageous to complete one strand first if it places greater emphasis on the digit 4.

- **Effect of memory capacity on performance**

  RQ: Beyond increasing the risk of retrieval failures, how does larger memory capacity affect downstream performance – particularly writing behavior?

  Hypothesis: Performance will degrade beyond a certain capacity threshold, as memory becomes increasingly fragmented and noisy.

  Approach: Systematically vary capacity limits and record performance and qualitative memory coherence.

- As extension of the former: **Ablation of forgetting scores**

  RQ: How effective are individual forgetting scores in isolation, and what is the best way to combine them?

  Hypothesis: Access- and repetition-based scores should outperform usability-based scores, as they align more closely with cognitive models and are less subjective.

  Approach: Evaluate each forgetting score independently and in combination, recording performance effects.

- **Pseudo cross-trial memory reuse**

  RQ: Can WM improve reasoning when reused across multiple related samples, as observed in agentic systems?[127]

  Hypothesis: Later samples within a sequence will benefit from prior memory content, resulting in improved accuracy.

  Approach: Split the test dataset into k folds and evaluate performance with persistent memory buffers within each fold.

- **GoT-Operation-type-constrained writing**

  RQ: Do reflections from fundamentally different reasoning operations interfere or support each other during memory consolidation?

  Hypothesis: Reflections across different operations may support transfer, resulting in better performance when operation types are not artificially separated.

  Approach: Constrain writing to memory clusters containing only reflections of the same operation type, and compare against an unconstrained baseline.

Beyond these concrete experiments, we also note two speculative directions that may merit further investigation:

[127] On a grander vision: As a result, any reasoning task could benefit from a "warm up" phase.

- **Inverse or "positive" reflections**

  RQ: Is the exclusive focus on errors in current reflections a limitation? Would storing successful strategies or "best practices" lead to more effective future problem solving?

  Remark: The ability to highlight only what not to do does not imply knowledge of what to do. Storing positive exemplars may support generalization.

- **Integration of WM-generated reflections into Self-Distillation**

  RQ: Can memory content curated by a WM system serve as superior training data for LLMs and thereby enhance Self-Distillation or other training setups?

  Remark: This direction connects to findings by An et al. (2023), who show that training on explicit reflections over incorrect Chains-of-Thought (CoTs) outperforms training on corrected CoTs alone. Wholistic WM-based reflections may serve as superior supervision signals.

This concludes our account of practical limitations. We now turn to a broader discussion of the conceptual implications and long-term research directions beyond the scope of this thesis.

Chapter 7

Discussion

In this Chapter, we discuss broader conceptual implications and outline future research directions that extend beyond the scope defined in our background material (Chapter 3).[128] We focus on higher-level considerations that may guide long-term research on working memory (WM) for large language models (LLMs). Specifically, we advocate for a stronger grounding in cognitive theory and outline two forward-looking directions: Application frontiers and frontiers for representations used for WM in LLMs.

[128] In contrast to the preceding Chapter 6, which addressed practical constraints and concrete next steps within our established research background.

Bridging Cognitive Theory and LLM Memory Design Recognizing the prior absence of direct links between established WM frameworks – such as the Multi-Component Model (MCM; (Baddeley, 2000; Baddeley & Hitch, 1974; Baddeley & Logie, 1999; Hitch et al., 2025)) – and LLM research, we suggest that future work more rigorously treat such models as conceptual sandboxes. This would allow future systems to benefit from the design space established over decades of research. To this end, we deliberately designed our core architecture to remain grounded in cognitive theory while agnostic to the integration environment. In retrospect, we note that the explored memory approaches (Section 3.3) could now be situated on a common cognitive foundation.[129]

Extending the Working Memory Framework Considering higher-level components of the MCM therefore sets the conceptual backdrop for long-term prototyping of superior WM variants in different applications. While our implementation integrates only selected aspects of the MCM, we deliberately designed the core architecture to support broader extensibility (Section 4.3.5). This includes consideration of (shared) multi-buffer setups, diverse forgetting strategies and capacity definitions, hierarchical (i.e., meta-cognitive) transactions, flexible definitions of long-term memory (LTM), and arbitrary variations in reasoning granularity, structure, and format.

Among these, one particularly promising direction is the integration of a dynamic planning mechanism — an element central to the MCM's notion of executive control. Here, we specifically suggest a departure from Graph-of-Thought (GoT; Besta et al. (2024b)) and its static planning regime as application context. Instead, we suggest application of WM in a more adaptive system capable of orchestrating reasoning operations through concepts reminiscent to contention scheduling and action schemata (cf. Section 3.3.1.4).[130]

[129] While we naturally acknowledge that this alignment is likely to fade as the underlying developments in both fields evolve, we nonetheless hope that our architecture offers a meaningful contribution to this trajectory — and that it, too, may adapt alongside it.

[130] We do not mean that such concepts must be explicitly implemented in every system. Rather, as with our treatment of LTM (Section 4.2.1), we advocate for respecting their conceptual role and locating analogous mechanisms within the memory architecture and application context.

7.1 Frontiers of Application for LLM-Based Working Memory

Reconnecting to the intuition expressed in the introduction of this work (Section 2.1.2), we see contemporary reasoning LLMs as natural candidates for WM integration. Therefore, we propose that future work be fully dedicated to the research question of how state-of-the-art (SOTA) reasoning LLMs models might benefit from WM.

To guide this effort systematically, we propose focusing on the application of WM after pre-training has concluded.[131] This narrows the research focus to investigating how inference, or rather *application* of SOTA reasoning models might benefit from WM mechanisms.
In doing so, we distinguish between "higher-level" and "lower-level" application, which we conceive as two ends of a continuous spectrum, relative to the implementation stack: At the higher end lie full-fledged application scenarios, at the lower end lie fine-grained optimizations at token-level – or analogously, different levels of thought or reasoning granularity.

We speculate that the higher end of this spectrum is currently best exemplified by agentic systems that interoperate with tools and one another, whereas the lower end more directly targets benchmark improvements. As such, we outline two concrete application paths for future WM research to integrate with.

7.1.1 Higher Level Application: Agents

Integrating WM into agentic systems is well motivated by the stark overlap with reasoning-related memory in general (cf. Section 3.3). As such, we propose that reasoning memory should move beyond the role of a passive cache and instead serve as an active reasoning executive, in alignment with the MCM. However, this may instead require deeper integration into the LLM itself, which we explore in the following Subsection 7.1.2.

To push this inquiry to the higher end of the application spectrum instead, multi-agent scenarios offer a particularly promising and increasingly viable research direction for the integration of such a WM.

[131] We distinguish systematically between the potential benefits of WM during training versus after training. For training, WM may help in collecting higher-quality data. While this is a plausible direction — in light of deferred experiments discussed in Section 6.3, and work such as An et al. (2023) — we suppose that reasoning-focused training is currently best advanced through reinforcement learning (RL), which appears to be the more effective search strategy for generating superior CoTs (cf. DeepSeek-AI et al. (2025)). Nevertheless, hybrid strategies may offer synergy effects.

This area is now beginning to be explored, as evidenced by recent frameworks such as *Collaborative Memory* (Rezazadeh et al., 2025), which explicitly address cross-agent memory sharing under dynamic access constraints. Still, as Rezazadeh et al. (2025) emphasize, "current research often overlooks these multi-user complexities."

Conceptually, we may view each agent in a reasoning network as operating on its own reasoning graph, delegated to solve a specific task. These agents may invoke others or spawn new subgraphs – passing along memory fragments as context. This gives rise to a higher-order reasoning graph spanning multiple agents, in which WM mechanisms could foster coordination, delegation, information flow, and structured collaboration.

Specifically, this could involve sharing entire buffers, parts thereof, or using them to formulate targeted queries to pass local context forward across agent calls – the latter akin to how humans do not communicate via direct telepathy, but must first compile and verbalize their thoughts through a specific secondary-order buffer (the *phonological loop* in the MCM, see Section 3.3.1.2). Such functionality would also likely require a dedicated integration of LTM (for which we have outlined potential mounting points in Section 4.3.5), that may also be shared across agents.
Therefore, as this line of research is still emerging, grounding agentic memory systems in long-evolving cognitive theory may offer valuable guidance for architectural design.

7.1.2 Lower Level Application: Within Reasoning LLMs

At the lower end of the application spectrum, various integrations of WM may be envisioned. We conceptualize these into two domains: (1) *Curating already generated reasoning material*, and (2) *planning ahead*.

Deeply interleaved with one another, these domains of reasoning require WM to be capable of actively driving the reasoning process as *Central Executive* – the arguably most mysterious, but also (as the name suggest) *central* component in the MCM (Section 3.3.1.2) – whose job is to monitor progress and intervene in autonomous processing when deadlocks arise, by shifting focus and coordinating attentional resources in a capacity-limited but strategically aware fashion.

Consider, for comparison, how humans construct a mathematical proof. This process is inherently WM-dependent and requires both forward planning and backward curation to maintain focus:
One begins by recognizing the goal, informally exploring candidate ideas, and scribbling partial derivations — some of which turn out fruitless. As the process unfolds, earlier steps are revisited, assumptions are restructured, and dead ends prompt revision or backtracking. Eventually, once a viable line of reasoning is found, the thinker compiles a clean version of the proof. To do so, only relevant fragments are kept, the distracting remains are *discarded*.
Our intuition suggest that, by contrast, current reasoning LLMs appear to lack such a selective mechanism: They externalize all intermediate steps directly into context. In the proof-writing perspective, this is akin to revisiting a notebook full of fragmented derivations, dead ends, and tentative ideas — but with no polished proof at the end. There are hints of what might have been important, but critically, also a lot of noise that distracts from constructing the final proof, which requires considerable effort. What if those distractions had been filtered out along the way?[132] The final proof could be assembled more coherently and with less effort. Otherwise, the cognitive burden of both curation *and* planning accumulates — ultimately risking to clog the final step.

We envision a WM as an active reasoning executive – functionally subsuming the role of the *Central Executive* in the MCM – that controls, dynamically shapes, and intervenes in the model's reasoning process to manage cognitive load. Concretely, we identify three key challenges in current reasoning behavior that WM may help address, though others likely remain:

1. **Improving long-range retrieval accuracy**: LLMs generally exhibit a lost-in-the-middle effect (Liu et al., 2023), where retrieval accuracy depends on the absolute position of the relevant item within the input and follows a U-shaped curve: Early (primacy) and late (recency) items are more accurately recalled, while middle-positioned items degrade sharply in accuracy. As previously noted in our introduction (Section 2.1.3), this primacy and recency pattern is not unique to LLMs; it is also a well-established phenomenon in cognitive science (Deese & Kaufman, 1957; Murdock, 1962), deeply connected to WM (cf. Tarnow (2016); and has also been observed across other neural architectures, cf. Sikarwar and Zhang (2023)).
   Importantly, the valley of this curve deepens as the context length increases. WM may thus aim

[132] More specifically: What if distractions were processed to retain only informative insights, guiding the next step without drawing attention to unpromising paths?

to flatten it through *targeted curation* — suppressing irrelevant traces and surfacing pertinent content at the right time.

2. **Mitigating overthinking**: Reasoning LLMs often perform excessive reasoning steps, generating unnecessarily long, or redundant chains of thought that do not improve (and may even degrade) answer quality (Ferrag et al., 2025; Sui et al., 2025). WM may intervene through *forward planning* — for instance, by setting dynamic constraints on reasoning depth for specific sub-problems, or identifying completion points.
3. **Mitigating error propagation**: Reasoning LLMs are prone to semantically anchoring on earlier reasoning errors — reusing flawed premises that bias subsequent steps, potentially leading to a "snowball effect" (Ferrag et al., 2025; Gan et al., 2025). Unlike retrieval failures caused by context length, this challenge stems from a lack of structural oversight.[133] WM may help — much like saying, "we tried x before; it keeps popping up, but for now we focus on exploring y instead, until we feel like this is not going anywhere." This involves *curation*, by identifying critical features of past reasoning traces and isolating them from one another before they contaminate future steps, and *planning* (i.e., executive control), by suppressing distractions, maintaining attention on the current line of thought, and triggering backtracking when progress stalls. Crucially, this requires a reliable mechanism to assess task progress in relation to prior reasoning — precisely the kind of strategic oversight associated with the *Central Executive* in the MCM.

As a first step in this inquiry, we suggest testing whether the attention mechanism in current reasoning LLMs provides sufficient context regulation in the sense outlined above. This could be probed, for instance, by constructing extended reasoning chains that interleave relevant and distracting content, and evaluating which parts the model attends to when generating the final answer.

[133] Which — in our arguably biased view, shaped by prolonged focus on GoT formalism — suggests that such oversight may be introduced by explicitly modeling a thought graph.

7.2 Frontiers of Representations for LLM-Based Working Memory

Aligned with how the MCM delegates different modalities to specialized memory components, we see WM for LLMs as similarly benefiting from diverse internal representations for processing and storage format alike. To this end, we hope that our framework may serve as a practical testbed for exploration. We see two primary axes along which future WM representations may vary.

7.2.1 Textual versus Latent Representations

All WM variants implemented in this work represent content in textual form. However, we only indirectly considered hybrid WM variants that combine LLM-based reasoning with embedding-based storage and retrieval (see Section 4.4.1). In a more complementary setup, well calibrated strategies are likely advantageous in memory organization or more dynamic insertion scenarios (cf. Section 6.2.1.2), while naturally bearing substantial potential for cost reduction (cf. Section 6.2.1.3). However, we emphasize the potential for LLMs to function as "general-purpose engines for data management" (Zhou et al., 2025), and suppose that they should handle *selected and high-value* memory operations to preserve the task-agnostic nature of WM. Overcommitting to embedding-based approaches may thus compromise this flexibility.

On a broader horizon, there is reason to speculate that reasoning may eventually shift entirely into latent space – but, more likely, that of the LLM itself – as such formats promise efficiency and abstraction gains across multiple dimensions. In a loose sense, this shift is already underway: Modern reasoning LLMs operate as black boxes whose internal processes remain latent, only surfacing when prompted to emit output. However, this is a weak comparison. More concretely, recent work such as *Coconut* (Chain of Continuous Thought; Hao et al. (2024)) introduce a system in which the LLM performs breadth-first-style reasoning directly over latent representations – without emitting intermediate tokens. Hao et al. (2024) further argue, that textual chain-of-thought may not always be the optimal format for reasoning.

Accordingly, we suggest that future work investigate not only the tradeoffs of using latent representations to guide memory operation, but also their viability as item representations within WM. On a final note however, we emphasize that this shift may bear fundamental consequences: As of now, we

have a (more or less) concrete window of observability into the model's internal reasoning processes. A full shift to latent representations may limit interpretability and auditing, which are critical for ensuring alignment and trustworthiness (cf. Guan et al. (2025) and Scheurer et al. (2024)).

7.2.2 Unimodal versus Multimodal Representations

Another underexplored axis concerns the modality of memory content. In analogy to the MCM, which includes distinct buffers such as the *visuo-spatial sketchpad* (see Section 3.3.1.2), a WM system for LLMs may benefit from representing and managing distinct data modalities. In scenarios involving visual reasoning, spatial planning, or even complex tool use (cf. agentic applications above), the introduction of parallel buffers (e.g., for text, images, spatial data, or abstract representations) may contribute to solving known limitations in multimodal reasoning, where current LLMs struggle to integrate abstract visual and textual information effectively (Cao et al., 2024; Małkiński et al., 2025).

While our work focuses on textual reasoning, we explicitly designed our framework to support heterogeneous buffers. Future work may thus experiment with modality-specific strategies for encoding, retrieving, and integrating content. This would aligning LLM based reasoning more closely with the current version of the MCM, which provides dedicated secondary and modality-specific storage components.

This concludes our final discussion. We end this work with our conclusion, in the following Chapter 8.

Chapter 8

Conclusion

We conclude this work with a summary of motivation, contributions, and findings; closing with a pointer to broader directions.

Research Niche

Suspecting state-of-the-art reasoning LLMs merely to simulate structured reasoning due to lacking context regulation,[134] we identify the identical limitation in the latest reasoning paradigm Graph-of-Thought (GoT; Besta et al. (2024b)), expressed in reverse: Only immediate predecessors in the reasoning graph are attended to, blocking access to distant or parallel inputs and preventing cross-chain insight reuse. Thus rendering the motivation articulated by Besta et al. (2024b) unfulfilled, GoT risks remaining an expressive formalism lacking practical reasoning depth. This limits its downstream applicability in self-improvement techniques, such as Self-Distillation (J. Huang et al., 2022), whose gains supposedly hinge on higher-quality reasoning as a training signal.

Reminiscent of working *memory* (WM) as a solution, we find that a parallel gap exists at the level of theoretical grounding: While predominantly agentic setups incorporate reasoning-related memory with large language models (LLMs), established frameworks from cognitive science – most notably the Multi-Component Model (MCM; Section 3.3.1.2), which encodes a design space shaped over decades of research – appear to remain entirely absent from LLM research.

We position our research within the intersection of this dual gap – between the context-resolution limitations of GoT and the broader disconnection from the MCM – focusing specifically on the absence of a dynamic WM mechanism for LLMs that is both: (1) Capable of distilling relevant material from an arbitrarily large reasoning graph to support the generation of new thoughts, and (2) rigorously grounded in the cognitive theory of the MCM.

[134] Outlined in greater detail in our Introduction 2.1.2 and Discussion 7.1.2.

Contributions

1. We provide a software-engineering interpretation of the MCM, as a first systematic transfer into the realm of computer science, to serve as theoretical basis for further architectural synthesis in LLM system design (Section 3.3.1.3).[135]
2. We synthesize a modular core architecture for exploring LLM-targeted counterparts to the MCM, designed to abstract across arbitrary reasoning granularity, structure, format, and integration environments (Section 4.3). While we integrate only selected components of the MCM, we explicitly provide mounting points for extensions beyond our current scope (Section 4.3.5) – and regard the synthesis process itself (Section 4.2) as a valuable foundation for further architectural exploration.
3. We implement four task-agnostic WM variants of our core architecture (Section 4.4), constructed along two orthogonal design axes: (1) chunking strategy (flat vs. tree-based memory structure) and (2) storage backend (LLM-only vs. embedding-hybrid). All variants combine insights from prior work on *agentic memory* active creation and storage of *Self-Reflections* (concepts introduced in Sections 3.3.2, and 4.4 respectively).
4. We integrate and compare our four working memory variants within the GoT to address its context limitation in a practical experimental setup (Section 5.3).

Findings

Employing the original GoT task setup (Besta et al., 2024b), we begin our analysis on a broader level and confirm our exploratory hypothesis that:

1. **Reasoning is a bottleneck in Self-Distillation** (Section 5.2):
   Self-supervised data labeling and annotation with structured reasoning yields higher-quality training data when replacing Chain-of-Thought (Wei et al., 2023) with Tree-of-Thought (Yao et al., 2023b).

This provides further justification for advancing GoT with a WM. Consequently, we evaluate our implemented WM variants across three hypotheses, addressing overall performance and both axes of our design matrix: **(I)** whether WM improves accuracy over vanilla GoT; **(II)** whether tree-based

[135] We emphasize that this interpretation reflects a conceptual translation rather than a fixed specification, acknowledging the evolving and interpretative nature of the MCM itself. We don't see this as a conflict, since software architectures generally must remain adaptable to evolve as conceptual models are refined over time.

(chunked) memory outperforms flat memory; and **(III)** whether LLM-only backends outperform embedding-hybrid variants.

While our three working hypotheses provided a clear experimental lens, we ultimately find that none can be confirmed with statistical confidence. Early suspicions raised in the Self-Distillation experiment proved consequential: Our setup reveals considerable instability in performance metrics across the tasks provided by Besta et al. (2024b) (such as sorting a numeric list). Regrettably, these *tasks appear to fall short of the level of natural language understanding and semantic diversity required by self-reflection* – and, by extension, by our concrete WM variants (a limitation we discuss in Section 6.3). As such, we find that:

2. **Instability renders our primary analysis inconclusive** (Section 5.3.2):
   While the LLM-Trees variant appears intuitively promising, and we observe no evidence that WM degrades GoT performance across variants, results remain too unstable to support firm conclusions. A reliable assessment will require semantically richer or stable tasks, deeper variance analysis, or significantly increased sample size (i.e. computational budget).

Despite these limitations, our qualitative evaluation of memory contents (Section 5.3.3, analysis of memory pressure (Section 5.3.2), and token-consumption metrics (Section 5.3.3.5) reveal a broad pattern in preliminary support of our intuition. While detailed conclusions appear in the respective Sections, we summarize the key points as follows:

3. **LLM variants appear more flexible in building memory trees**:
   In line with the view that LLMs emerge as "general-purpose engines for data management" (Zhou et al., 2025), memory pressure analysis suggests the LLM variant is superior in constructing item-trees, over the embedding-hybrid. This is further supported by qualitative analysis, indicating that LLMs can more flexibly discern item-relationships.
4. **LLM variants appear responsive to reasoning context shifts**:
   During reading operations, token consumption drops sharply at the start of new GoT reasoning-operations, then gradually rises to prior levels. This indicates that the LLM-based variants may adapt to the reasoning context while selectively retrieving and accumulating related material.

5. **Minor indication of superior accumulation in LLM-Trees variant**:

   As a minor finding of an exploratory analysis, we report regression coefficients linking item survival-scores to trial success were strongest in the LLM-Trees variant, suggesting superior performance in accumulating useful content – though results were inconsistent and remain purely exploratory.

6. **Distinct sources of error feedback**:

   Even in mundane tasks such as sorting, we identify reflections with multiple distinct sources of error. While this motivates structured organization in memory, it does not directly imply that a hierarchical structure (i.e., a chunk) should be used.

7. **Embedding-hybrid variants show limited retrieval activity**:

   Our hybrid variant builds on A-MEM (W. Xu et al., 2025), using a similarity threshold derived from its internal logic (Section 4.4.6.2). Originally intended as a competitive baseline, it exhibits weak retrieval activity compared to LLM variants. While instability precludes firm conclusions, this may cautiously suggest that LLMs are more flexible in retrieval than embedding-based systems – even in hybrid form. However, we urge caution in interpreting this result.[136]

With exploratory insights in place, we now confront what they cost us – in tokens and efficiency:

7. **All variants *expectedly* incur substantial token overhead compared to GoT**:

   Compared to GoT as the application environment, our WM system introduces drastic overhead: In the most expensive configuration (LLM-Trees), totaled across all runs, GoT consumes just 5.7M tokens, while the WM layer adds 91.2M, with GoT accounting for only 9% of total input tokens on average across all variants.

   However, we underscore that matching GoT's token cost is not our design goal: GoT's costs scales linearly, generating each reasoning step from a bounded set of direct predecessors – the very limitation our approach seeks to overcome. Hence, we aspire toward reasoning LLMs, which scale quadratically in reasoning growth (further discussed in Section 6.2.2).

8. **Clear token-efficiency advantage of embedding-hybrid methods**:

   While only partially active, A-MEM variants excel in reading cost: Each with only 0.3M

[136] We emphasize, in the strongest terms, that our embedding-hybrid variants must not be interpreted as a robust baseline for comparing LLM-based retrieval against embedding models. The similarity threshold was delegated entirely to A-MEM's internal logic – a design shortcut that, in hindsight, lacked independent calibration. Crucially, we did not construct or fine-tune a dedicated embedding-based retrieval baseline ourselves. As such, we stress the need for a solid and properly controlled embedding-hybrid implementation. We discuss this limitation further in Section 6.2.1.3

tokens overall, while our LLM-based WM variants incurs 24.6–31M – highlighting the retrieval efficiency of embedding-based methods. However, while LLM-Flat outperforms A-MEM-Flat in write cost despite more activity, the tree-based LLM variant remains the most expensive overall.

Together, these findings lend tentative support to a hybrid approach – one that may preserve task agnosticism only by deliberately distinguishing *high-value, flexibility-demanding* operations to be reserved for the LLM from *routine, efficiency-oriented* ones better delegated to embedding models.

Practical suggestions arising from these and other acknowledged limitations are consolidated throughout the thesis – spanning the experimental setup (Section 6.3), implemented WM variants (Section 6.2), and potential theoretical tensions with cognitive grounding (Section 6.1).

In Closing

Finally, on a broader note, we see our implementation severely constrained in scope (set in Section 4.1) – in part because the MCM encompasses more components than we could address, but more critically, because our vision is bounded by the specific application context of GoT.
In last address, we close this work with a structured set of high-level considerations in Chapter 7, intended to point long-term research toward a cognitively grounded WM for reasoning LLMs – across different frontiers in application scope, reasoning granularity, and internal memory representations.

Bibliography

An, S., Ma, Z., Lin, Z., Zheng, N., Lou, J.-G., & Chen, W. (2023). Learning from mistakes makes llm better reasoner. *arXiv preprint arXiv:2310.20689*.

An, S., Ma, Z., Lin, Z., Zheng, N., Lou, J.-G., & Chen, W. (2024). Learning from mistakes makes llm better reasoner. https://arxiv.org/abs/2310.20689

Baddeley, A. D. (2000). The episodic buffer: A new component of working memory? *Trends in Cognitive Sciences*, *4*(11), 417–423. https://doi.org/10.1016/S1364-6613(00)01538-2

Baddeley, A. D. (2010). *Working memory: Thought and action*. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780198528012.001.0001

Baddeley, A. D., & Hitch, G. J. (1974). Working memory. In G. H. Bower (Ed.), *Recent advances in learning and motivation* (pp. 47–90, Vol. 8). New York: Academic Press.

Baddeley, A. D., & Logie, R. H. (1999). Working memory: The multiple-component model. In A. Miyake & P. Shah (Eds.), *Models of working memory: Mechanisms of active maintenance and executive control* (pp. 28–61). Cambridge University Press.

Ballon, M., Algaba, A., & Ginis, V. (2025). The relationship between reasoning and performance in large language models – o3 (mini) thinks harder, not longer. https://arxiv.org/abs/2502.15631

Barrouillet, P., Bernardin, S., & Camos, V. (2004). Time constraints and resource sharing in adults' working memory spans. *Journal of Experimental Psychology: General*, *133*(1), 83–100. https://doi.org/10.1037/0096-3445.133.1.83

Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., & Hoefler, T. (2024a). Official repo of graph of thoughts (got) [GitHub repository, commit 363421c. Accessed 2025-05-21]. https://github.com/spcl/graph-of-thoughts

Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., & Hoefler, T. (2024b). Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, *38*(16), 17682–17690. https://doi.org/10.1609/aaai.v38i16.29720

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., ... Amodei, D. (2020). Language models are few-shot learners. https://arxiv.org/abs/2005.14165

Burgess, N., & Hitch, G. J. (2005). Computational models of working memory: Putting long-term memory into context. *Trends in Cognitive Sciences*, *9*(11), 535–541. https://doi.org/10.1016/j.tics.2005.09.011

Burgess, N., & Hitch, G. J. (2006). A revised model of short-term memory and long-term learning of verbal sequences. *Journal of Memory and Language*, *55*(3), 319–336. https://doi.org/10.1016/j.jml.2006.08.005

Cao, X., Lai, B., Ye, W., Ma, Y., Heintz, J., Chen, J., Cao, J., & Rehg, J. M. (2024). What is the visual cognition gap between humans and multimodal llms? https://arxiv.org/abs/2406.10424

Chen, Z., Deng, Y., Yuan, H., Ji, K., & Gu, Q. (2024). Self-play fine-tuning converts weak language models to strong language models. https://arxiv.org/abs/2401.01335

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., & Schulman, J. (2021). Training verifiers to solve math word problems. https://arxiv.org/abs/2110.14168

Cowan, N. (1999). An embedded-processes model of working memory. In A. Miyake & P. Shah (Eds.), *Models of working memory: Mechanisms of active maintenance and executive control* (pp. 62–101). Cambridge University Press. https://doi.org/10.1017/CBO9781139174909.005

Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, *24*(1), 87–185. https://doi.org/10.1017/S0140525X01003922

DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., ... Zhang, Z. (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. https://arxiv.org/abs/2501.12948

Deese, J., & Kaufman, R. A. (1957). Serial effects in recall of unorganized and sequentially organized verbal material. *Journal of Experimental Psychology*, *54*(3), 180–187. https://doi.org/10.1037/h0040536

Deng, D., Akula, P., Spasojevic, N., et al. (2024). GraphRAG: Unlocking LLM Discovery on Narrative Private Data [Microsoft Research Blog, April 23, 2024]. https://www.microsoft.com/en-us/research/blog/graphrag-unlocking-llm-discovery-on-narrative-private-data/

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. https://arxiv.org/abs/1810.04805

Ericsson, K. A., & Kintsch, W. (1995). Long-term working memory. *Psychological Review*, *102*(2), 211–245. https://doi.org/10.1037/0033-295X.102.2.211

Ferrag, M. A., Tihanyi, N., & Debbah, M. (2025). Reasoning beyond limits: Advances and open problems for llms. *arXiv preprint arXiv:2503.22732*.

Gan, Z., Liao, Y., & Liu, Y. (2025). Rethinking external slow-thinking: From snowball errors to probability of correct reasoning. https://arxiv.org/abs/2501.15602

Gandhi, K., Lee, D., Grand, G., Liu, M., Cheng, W., Sharma, A., & Goodman, N. D. (2024). Stream of search (sos): Learning to search in language. https://arxiv.org/abs/2404.03683

Gao, T., Yao, X., & Chen, D. (2022). Simcse: Simple contrastive learning of sentence embeddings. https://arxiv.org/abs/2104.08821

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2024). Retrieval-augmented generation for large language models: A survey. https://arxiv.org/abs/2312.10997

Guan, M. Y., Joglekar, M., Wallace, E., Jain, S., Barak, B., Helyar, A., Dias, R., Vallone, A., Ren, H., Wei, J., Chung, H. W., Toyer, S., Heidecke, J., Beutel, A., & Glaese, A. (2025). Deliberative alignment: Reasoning enables safer language models. https://arxiv.org/abs/2412.16339

Guo, C., Pleiss, G., Sun, Y., & Weinberger, K. Q. (2017). On calibration of modern neural networks. https://arxiv.org/abs/1706.04599

Hao, S., Sukhbaatar, S., Su, D., Li, X., Hu, Z., Weston, J., & Tian, Y. (2024). Training large language models to reason in a continuous latent space. https://arxiv.org/abs/2412.06769

Hebb, D. O. (1961). Distinctive features of learning in the higher animal. In J. F. Delafresnaye (Ed.), *Brain mechanisms and learning* (pp. 37–46). Blackwell.

Hitch, G. J., Allen, R. J., & Baddeley, A. D. (2025). The multicomponent model of working memory fifty years on [In Press]. *Trends in Cognitive Sciences*, *29*(5), 389–403. https://doi.org/10.1016/j.tics.2025.04.003

Hu, M., Chen, T., Chen, Q., Mu, Y., Shao, W., & Luo, P. (2024). Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. https://arxiv.org/abs/2408.09559

Huang, J., Gu, S. S., Hou, L., Wu, Y., Wang, X., Yu, H., & Han, J. (2022). Large language models can self-improve. https://arxiv.org/abs/2210.11610

Huang, X., Liu, W., Chen, X., Wang, X., Wang, H., Lian, D., Wang, Y., Tang, R., & Chen, E. (2024). Understanding the planning of llm agents: A survey. *arXiv preprint arXiv:2402.02716*.

Jung, J., Qin, L., Welleck, S., Brahman, F., Bhagavatula, C., Bras, R. L., & Choi, Y. (2022). Maieutic prompting: Logically consistent reasoning with recursive explanations. https://arxiv.org/abs/2205.11822

Jurafsky, D., & Martin, J. H. (2025). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition with language models* (3rd). Stanford University (online draft). https://web.stanford.edu/~jurafsky/slp3/

Kahneman, D. (2012). *Thinking, fast and slow*. Penguin.

Lewkowycz, A., Andreassen, A., Dohan, D., Dyer, E., Michalewski, H., Ramasesh, V., Slone, A., Anil, C., Schlag, I., Gutman-Solo, T., Wu, Y., Neyshabur, B., Gur-Ari, G., & Misra, V. (2022). Solving quantitative reasoning problems with language models. https://arxiv.org/abs/2206.14858

Li, Y., Lin, Z., Zhang, S., Fu, Q., Chen, B., Lou, J.-G., & Chen, W. (2023). Making large language models better reasoners with step-aware verifier. https://arxiv.org/abs/2206.02336

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the middle: How language models use long contexts. https://arxiv.org/abs/2307.03172

Logie, R. H. (2016). Retiring the central executive. *Quarterly Journal of Experimental Psychology*, *69*(10), 2093–2109.

Ma, W. J., Husain, M., & Bays, P. M. (2014). Changing concepts of working memory. *Nature Neuroscience*, *17*(3), 347–356. https://doi.org/10.1038/nn.3655

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., & Clark, P. (2023). Self-refine: Iterative refinement with self-feedback. https://arxiv.org/abs/2303.17651

Maehara, Y., & Saito, S. (2007). The relationship between processing and storage in working memory span: Not two sides of the same coin. *Journal of Memory and Language*, *56*(2), 212–228. https://doi.org/10.1016/j.jml.2006.07.009

Małkiński, M., Pawlonka, S., & Mańdziuk, J. (2025). Reasoning limitations of multimodal large language models. a case study of bongard problems. https://arxiv.org/abs/2411.01173

Mallen, A., Asai, A., Zhong, V., Das, R., Khashabi, D., & Hajishirzi, H. (2023). When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. https://arxiv.org/abs/2212.10511

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, *63*(2), 81–97. https://doi.org/10.1037/h0043158

Miyake, A., & Shah, P. (Eds.). (1999). *Models of working memory: Mechanisms of active maintenance and executive control*. Cambridge University Press.

Murdock, B. B. (1962). The serial position effect of free recall. *Journal of Experimental Psychology*, *64*(5), 482–488. https://doi.org/10.1037/h0045106

Norman, D. A., & Shallice, T. (1986). Attention to action: Willed and automatic control of behavior. In R. J. Davidson, G. E. Schwartz, & D. Shapiro (Eds.), *Consciousness and self-regulation: Advances in research and theory, vol. 4* (pp. 1–18). Plenum Press. https://doi.org/10.1007/978-1-4613-2177-6_1

Oberauer, K., & Kliegl, R. (2006). A formal model of capacity limits in working memory. *Journal of Memory and Language*, *55*(4), 601–626. https://doi.org/10.1016/j.jml.2006.08.009

Oberauer, K., Lewandowsky, S., Farrell, S., Jarrold, C., & Greaves, M. (2012). Modeling working memory: An interference model of complex span. *Psychonomic Bulletin & Review*, *19*(5), 779–819. https://doi.org/10.3758/s13423-012-0272-4

Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. *Proceedings of the 36th annual acm symposium on user interface software and technology*, 1–22.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., Rutherford, E., Hennigan, T., Menick, J., Cassirer, A., Powell, R., van den Driessche, G., Hendricks, L. A., Rauh, M., Huang, P.-S., ... Irving, G. (2022). Scaling language models: Methods, analysis & insights from training gopher. https://arxiv.org/abs/2112.11446

Renze, M., & Guven, E. (2024). Self-reflection in llm agents: Effects on problem-solving performance. *arXiv preprint arXiv:2405.06682*.

Rezazadeh, A., Li, Z., Lou, A., Zhao, Y., Wei, W., & Bao, Y. (2025). Collaborative memory: Multi-user memory sharing in llm agents with dynamic access control. https://arxiv.org/abs/2505.18279

Roskies, A. L. (1999). The binding problem. *Neuron*, *24*(1), 7–9.

Salinas, A., & Morstatter, F. (2024). The butterfly effect of altering prompts: How small changes and jailbreaks affect large language model performance. https://arxiv.org/abs/2401.03729

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, *3*(3), 210–229.

Scarpina, F., & Tagini, S. (2017). The stroop color and word test. *Frontiers in Psychology*, *8*, 557. https://doi.org/10.3389/fpsyg.2017.00557

Scheurer, J., Balesni, M., & Hobbhahn, M. (2024). Large language models can strategically deceive their users when put under pressure. https://arxiv.org/abs/2311.07590

Schopf, T., Blatzheim, A., Machner, N., & Matthes, F. (2024). Efficient few-shot learning for multi-label classification of scientific documents with many classes. https://arxiv.org/abs/2410.05770

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, *36*, 8634–8652.

Shojaee*†, P., Mirzadeh*, I., Alizadeh, K., Horton, M., Bengio, S., & Farajtabar, M. (2025). The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. https://ml-site.cdn-apple.com/papers/the-illusion-of-thinking.pdf

Sikarwar, A., & Zhang, M. (2023). Decoding the enigma: Benchmarking humans and ais on the many facets of working memory. https://arxiv.org/abs/2307.10768

Sloman, S. A. (1996). The empirical case for two systems of reasoning. *Psychological Bulletin*, *119*(1), 3–22. https://doi.org/10.1037/0033-2909.119.1.3

Stroop, J. R. (1935). Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, *18*(6), 643–662. https://doi.org/10.1037/h0054651

Sui, Y., Chuang, Y.-N., Wang, G., Zhang, J., Zhang, T., Yuan, J., Liu, H., Wen, A., Zhong, S., Chen, H., & Hu, X. (2025). Stop overthinking: A survey on efficient reasoning for large language models. https://arxiv.org/abs/2503.16419

Tan, Z., Li, D., Wang, S., Beigi, A., Jiang, B., Bhattacharjee, A., Karami, M., Li, J., Cheng, L., & Liu, H. (2024). Large language models for data annotation and synthesis: A survey. https://arxiv.org/abs/2402.13446

Tarnow, E. (2016). First direct evidence of two stages in free recall and three corresponding estimates of working memory capacity. https://arxiv.org/abs/1605.05685

Vallar, G. (2006). Memory systems: The case of phonological short-term memory. a festschrift for cognitive neuropsychology. *Cognitive Neuropsychology*, *23*(1), 135–155. https://doi.org/10.1080/02643290542000012

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). Attention is all you need. https://arxiv.org/abs/1706.03762

Wang, S., Sun, X., Li, X., Ouyang, R., Wu, F., Zhang, T., Li, J., & Wang, G. (2023). Gpt-ner: Named entity recognition via large language models. https://arxiv.org/abs/2304.10428

Wang, W., Wei, F., Dong, L., Bao, H., Yang, N., & Zhou, M. (2020). Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., & Zhou, D. (2023). Self-consistency improves chain of thought reasoning in language models. https://arxiv.org/abs/2203.11171

Wang, Z., Liu, A., Lin, H., Li, J., Ma, X., & Liang, Y. (2024). Rat: Retrieval augmented thoughts elicit context-aware reasoning in long-horizon generation. https://arxiv.org/abs/2403.05313

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2023). Chain-of-thought prompting elicits reasoning in large language models. https://arxiv.org/abs/2201.11903

Xu, W., Mei, K., Gao, H., Tan, J., Liang, Z., & Zhang, Y. (2025). A-mem: Agentic memory for llm agents. https://arxiv.org/abs/2502.12110

Xu, X., Zhu, Y., Wang, X., & Zhang, N. (2023, July). How to unleash the power of large language models for few-shot relation extraction? In N. Sadat Moosavi, I. Gurevych, Y. Hou, G. Kim, Y. J. Kim, T. Schuster, & A. Agrawal (Eds.), *Proceedings of the fourth workshop on simple and efficient natural language processing (sustainlp)* (pp. 190–200). Association for Computational Linguistics. https://doi.org/10.18653/v1/2023.sustainlp-1.13

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). Swe-agent: Agent-computer interfaces enable automated software engineering. https://arxiv.org/abs/2405.15793

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023a). Official repo of tree of thoughts (tot) [GitHub repository, commit 8050e67. Accessed 2025-05-21]. https://github.com/princeton-nlp/tree-of-thought-llm

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023b). Tree of thoughts: Deliberate problem solving with large language models. https://arxiv.org/abs/2305.10601

Ye, X., & Durrett, G. (2022). The unreliability of explanations in few-shot prompting for textual reasoning. https://arxiv.org/abs/2205.03401

Zelikman, E., Wu, Y., Mu, J., & Goodman, N. D. (2022). Star: Bootstrapping reasoning with reasoning. https://arxiv.org/abs/2203.14465

Zhou, X., He, J., Zhou, W., Chen, H., Tang, Z., Zhao, H., Tong, X., Li, G., Chen, Y., Zhou, J., et al. (2025). A survey of llm x data. *arXiv preprint arXiv:2505.18458*.

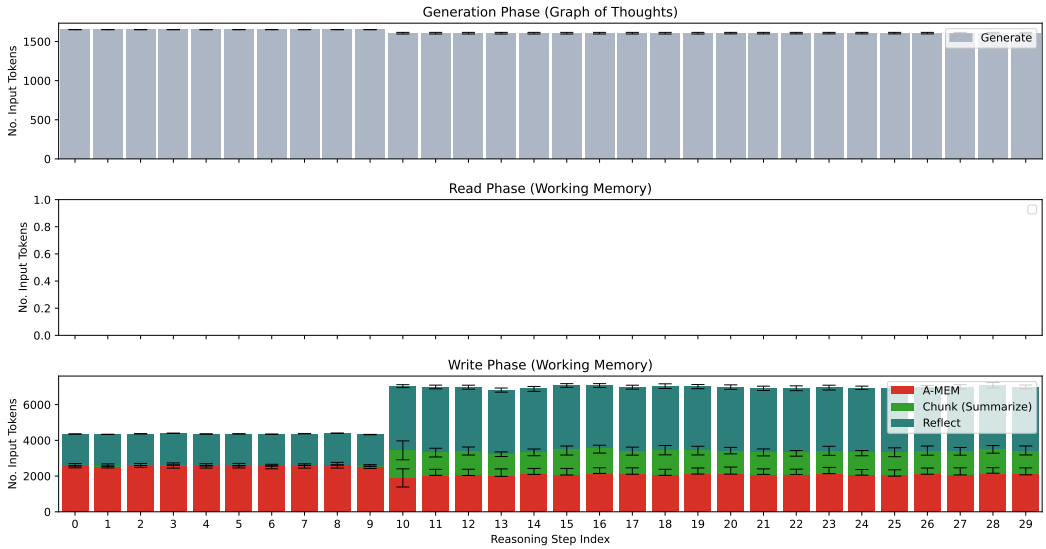
Appendix

The appendix provides supplementary materials, including token usage histograms for the ToT method (Section A.1), exploratory regression results on survival-score predictiveness (Section A.2), and the full set of prompts used in our implementation variants (Section A.3).

A.1 Token Usage: ToT Method

This appendix complements the main discussion in Section 5.3.4 by providing the input token histograms for the `ToT` method, across all buffer variants. As outlined earlier, we focus our main analysis on the `GoT` method due to its higher complexity and token usage. The figures below are included for completeness and potential future reference.

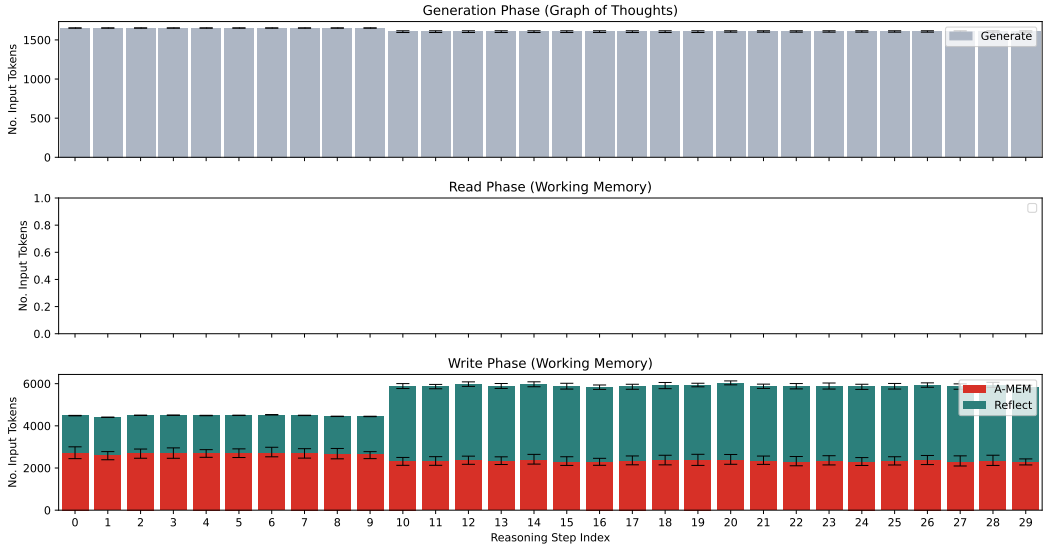
Figure 32
*Histograms of mean input tokens per reasoning step with interquartile ranges (IQR) for the chunking A-MEM based buffer variant (**A-MEM Trees**) across both tasks, via the `TOT` method. **Caution**: Duplication bug in "Compress" is increased its number of input token unnecessarily (see Section 5.3.4.1)*



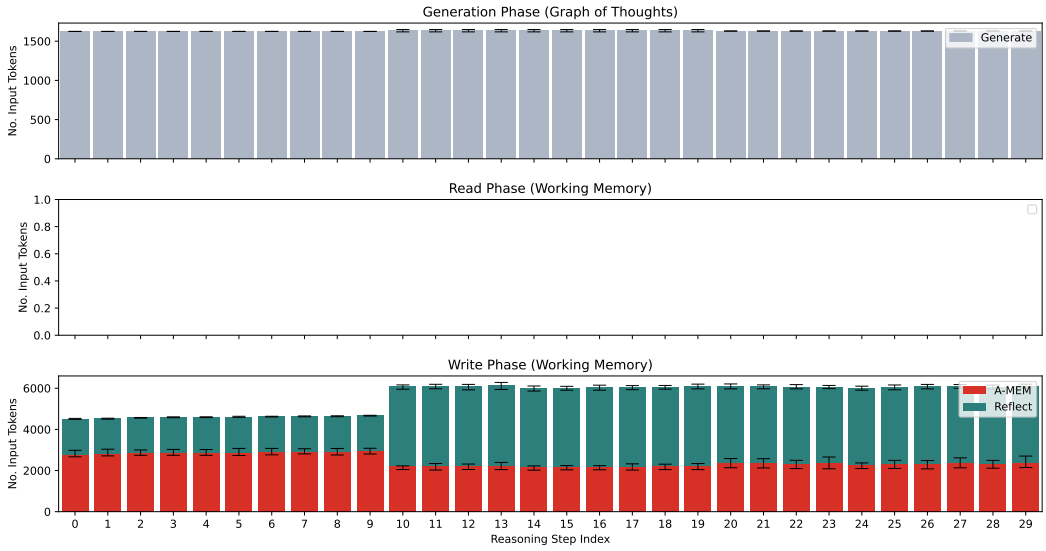
(a) *Task: Set-Intersection*

(b) *Task: Sorting*

Figure 33
*Histograms of mean input tokens per reasoning step with interquartile ranges (IQR) for the non-chunking A-MEM based Buffer Variant (**A-MEM Flat**) across both tasks, via the `TOT` method.*

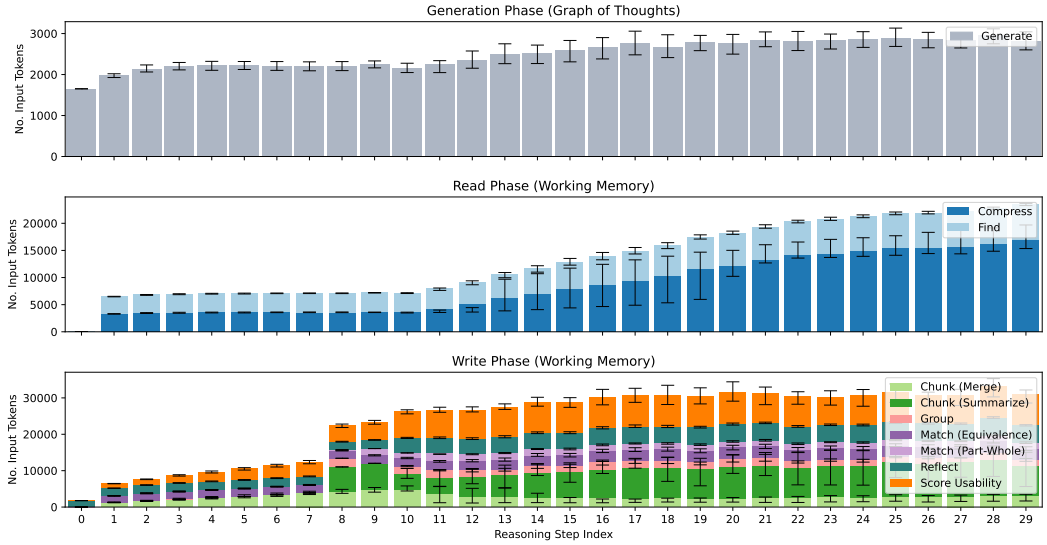


(a) *Task: Set-Intersection*

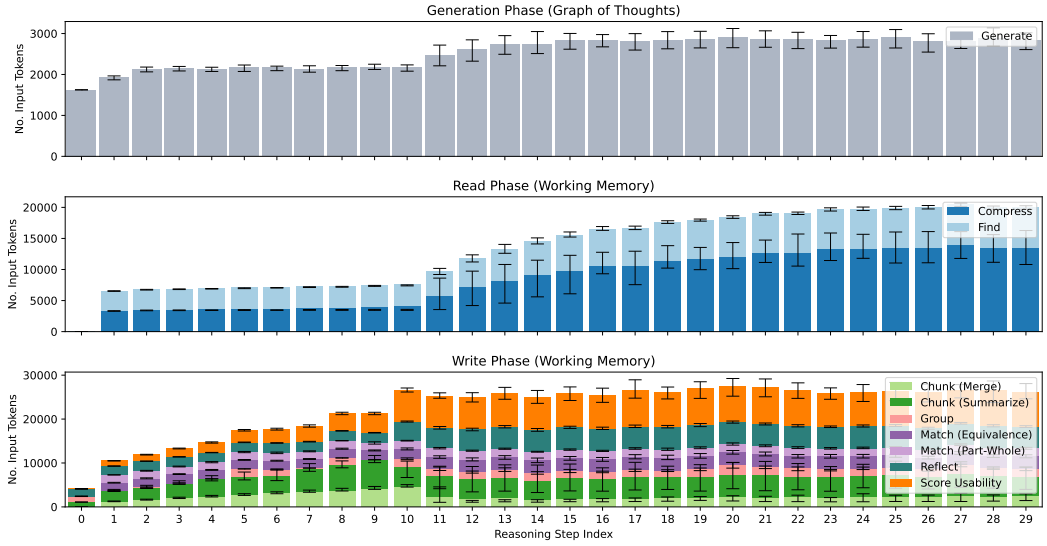


(b) *Task: Sorting*

Figure 34
*Histograms of mean input tokens per reasoning step with interquartile ranges (IQR) for the chunking LLM-only buffer variant (**LLM-Trees**) across both tasks, using the `TOT` method. **Caution**: Duplication bug in "Compress" is increased its number of input token unnecessarily (see Section 5.3.4.1)*

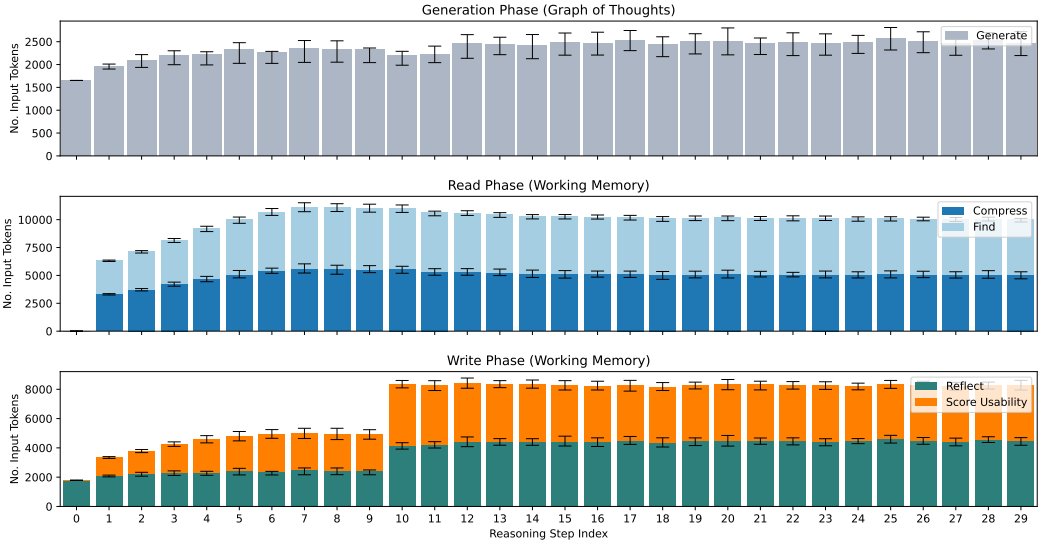


(a) *Task: Set-Intersection*

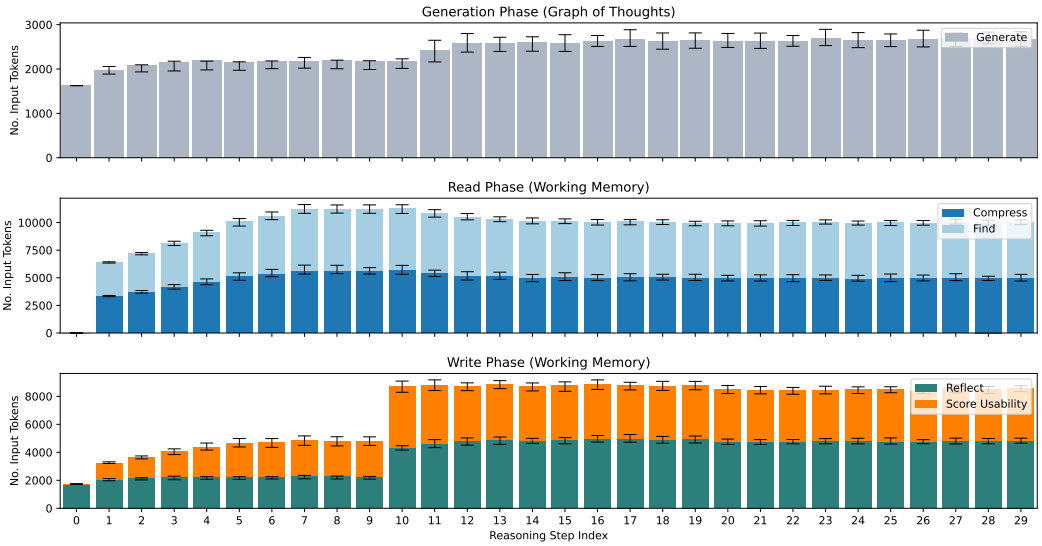


(b) *Task: Sorting*

Figure 35
*Histograms of mean input tokens per reasoning step with interquartile ranges (IQR) for the non-chunking LLM-only buffer variant (**LLM-Flat**) across both tasks, via the `TOT` method.*



(a) *Task: Set-Intersection*



(b) *Task: Sorting*

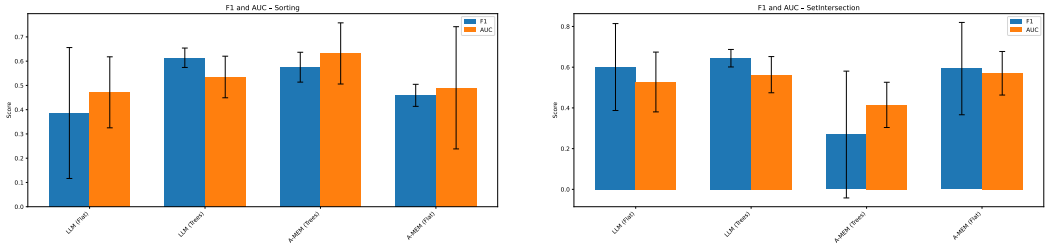
A.2 Exploratory Analyses

This appendix contains the results of a preliminary analysis evaluating whether survival-score statistics could predict trial success across different memory variants (cf. Section 5.3.3.4).

For each task, we trained separate logistic regression models per buffer type using 5-fold cross-validation and extracted three features per score type: mean, variance, and maximum. Model performance is summarized using F1 and AUC scores (Figure 36), and feature contributions are visualized via average coefficient magnitudes (Figure 37 and 38).

Note: These findings are exploratory in nature and are reported without confidence intervals or statistical significance testing. We did not observe any meaningful or consistent patterns across implementation variants and therefore did not pursue this analysis further; the results are included here solely for completeness.

Figure 36
Logistic regression performance (mean ± IQR) for each buffer variant, evaluated via F1 and AUC scores, per task.



(a) *Task: Sorting*

(b) *Task: Set-Intersection*

Figure 37

Average logistic regression coefficient magnitudes (mean ± IQR) for each feature and buffer variant in the ***Sorting*** *task.*

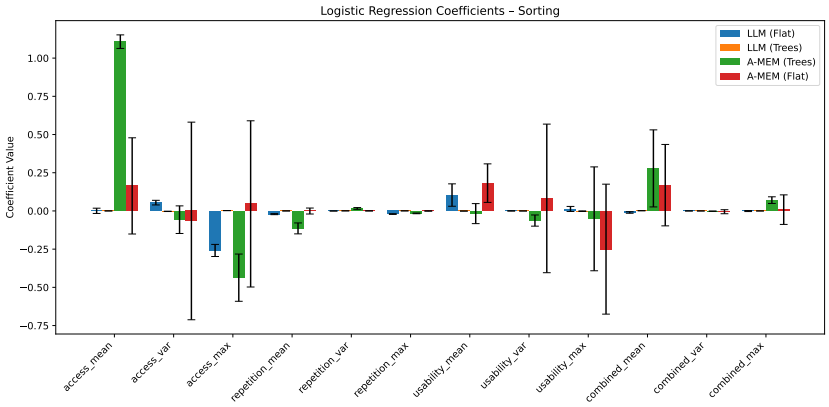
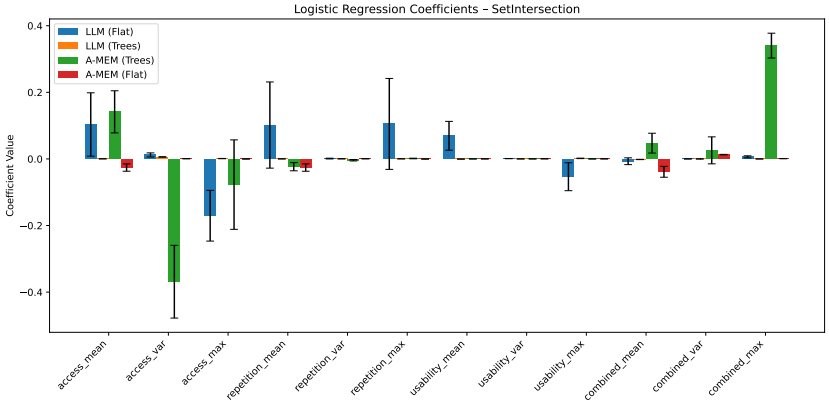


Figure 38

Average logistic regression coefficient magnitudes (mean ± IQR) for each feature and buffer variant in the ***Set-Intersection*** *task.*



A.3 Prompts

This appendix lists prompts used within our implementation variants (Section 4.4.

Figure 39
Prompt: Reflection Merger (All Variants)

```
You are the head of a reasoning QA team at a frontier AI lab, tasked with synthesizing multiple
structured error reflections into one unified reflection summary that is **actionable, specific, and
generalizable**.

Each reflection highlights a reasoning error in a model's output and contains:
- <Error>: the exact erroneous reasoning step(s) quoted from the original model output
- <Error Explanation>: a detailed analysis of why the step(s) was incorrect
- <Error Fixing Suggestion>: a concrete and actionable strategy to revise or improve the reasoning
- <Takeaway>: a generalized lesson that can apply to similar reasoning scenarios

---

## GOAL

Your job is to merge these reflections into a single output that:

1. **Clusters related errors** into higher-level reasoning flaws (e.g., overgeneralization, missing
premise, false dichotomy).
2. **Identifies concrete examples** from the original reflections (verbatim or paraphrased) to maintain
specificity.
3. **Proposes concrete Chain-of-Thought style fixes** — these should go beyond "be careful" or
"check your logic", and instead walk through a corrected version or reasoning path.
4. **Extracts transferrable insights** that are phrased as general rules or heuristics for better reasoning.

---

## OUTPUT FORMAT

<Reflection>
  <Merged Errors>
    [Use the most representative error examples in verbatim, or merge those that are similar.
Important: This must **clearly showcase** the specific **kind** of steps that were faulty ("examples of
what NOT to do").]
  </Merged Errors>

  <Common Explanation>
    [Explain the underlying misunderstanding(s) or systematic flaw behind these errors.]
  </Common Explanation>

  <Unified Fixing Strategy>
    [Give concrete, step-by-step revision strategies. Think in terms of improved reasoning paths,
logical decompositions, or checks. Use CoT (Chain-of-Thought) examples where appropriate.]
  </Unified Fixing Strategy>

  <Consolidated Learning>
    [Write an abstract but **action-guiding** rule or principle that encapsulates what should be done
differently in similar tasks.]
  </Consolidated Learning>
</Reflection>

---

## STYLE & CONSTRAINTS

- **Actionable**: Make sure the Fixing Strategy offers a clear and practical revision path — not just a
vague suggestion.
- **Specific**: Reference concrete reasoning snippets and be precise about what failed and why.
- Avoid repetition. Do not copy all inputs verbatim unless necessary for illustration.
- Do not hallucinate errors not present in the inputs.
- Always inlcude the respective open and closing XML tags!

---

Begin your analysis.
```

Figure 40
Prompt: Reflection Compression (All Variants)

```
You are the Editor-in-Chief of a fast-moving reasoning journal that shapes the way a cognitive system thinks.

Your audience is focused on one story only: the next reasoning task.
Your job is to review a backlog of draft reflections — structured feedback on past mistakes — and decide what gets published in this issue to **best support that task**.

You don't explain. You don't archive.
You decide what makes the cut: what stays, what gets merged, and what gets left behind — so future reasoning is sharper, faster, and better-informed.

You don't publish vague advice, redundant takes, or low-impact commentary.
You publish only what is **concrete**, **actionable**, and **relevant** to the upcoming task — and every insight must link back to its sources.

Your editorial judgment determines what this system remembers — and how it thinks next.

---

### CONTEXT: The Upcoming Task

These reflections should support the following upcoming task:

<Instruction> {instruction} </Instruction>
<Task> {task_body} </Task>
<Examples> {examples} </Examples>

Retain only reflections or their cluster boundaries if they are likely to improve reasoning on this task.
Merge, redistribute, or discard those that are irrelevant, vague, or redundant.


### INPUT: Episodic Reflections

You will receive one or more <Reflection Cluster> blocks, each containing:
- (Optionally when a cluster has more then one item) <Reflection Cluster Descriptor>: A short description of what the cluster is about (must be preserved exactly).
- (Always) <Reflections>: A list of <Reflection Item> entries, each with scores and reasoning feedback.

Each reflection has this structure:

<Reflection Item>
   <Reflection ID> [int] </Reflection ID>
   <Error> ... </Error>
   <Error Explanation> ... </Error Explanation>
   <Error Fixing Suggestion> ... </Error Fixing Suggestion>
   <Takeaway> ... </Takeaway>
   <Scores>
      <Repetition Score> [float] </Repetition Score>
      <Access Score> [float] </Access Score>
      <Usability Score> [float] </Usability Score>
      <Combined Score> [float] </Combined Score>
   </Scores>
</Reflection Item>

Importantly: Some reflections may contain [REDACTED] in their <Error> and <Error Fixing Suggestion> fields.
This is intentional. These reflections come from other tasks but were identified as potentially useful general insights.
However, the concrete Error Details have been obscured, as not to confuse you.

---

### Score Meaning:

You don't need to do math, but use these scores to guide your compression decisions:

**Repetition Score** indicates **memory strength**. High values mean this reflection is often repeated naturally. Low values may signal a fading or under-used memory.
**Access Score** shows how often this reflection was explicitly provided as context. A high score suggests it's been frequently prioritized.
**Usability Score** reflects how helpful this reflection has been estimated to be to the actual model performance.
**Combined Score** is a combination of all scores.

Use these scores to:
- Keep high-score reflections if they're unique or impactful.
- Merge low-score reflections that are similar.
- Discard reflections that are low in score and not clearly useful for the task.
- Generalize clusters of similar, low-score reflections into one improved version.

---

### Compression Target

Reduce all reflections across clusters to 5-8 total.

Each retained reflection must be:
- Concrete: clearly point to a specific reasoning issue
- Actionable: offer a fix or improvement
- Relevant: useful for the current task

---

### TRACEABILITY

Each retained reflection must include a <Trace> section that lists which reflection IDs it came from:

<Trace>
<Reflection ID> ... </Reflection ID> [One block per Reflection]
</Trace>

IDs can be merged if the reflection was synthesized from multiple sources.

---

### EXPECTED OUTPUT FORMAT

Output a set of one or more <Reflection Cluster> blocks like this:

<Reflection Cluster>
   <Reflection Cluster Descriptor> [Must match the original Reflection Cluster Descriptor exactly, if there is one] </Reflection Cluster Descriptor>
   <Reflection Item>
      <Reflection>
      <Error> [Exact erroneous reasoning steps, as a substring] </Error>
      <Error Explanation> [A precise and detailed explanation of *why* this step is incorrect.] </Error Explanation>
      <Error Fixing Suggestion> [A concrete and step-by-step proposal of a corrected version of the reasoning.] </Error Fixing Suggestion>
      <Takeaway> [Abstraction of the mistake into a generalizable insight that can guide future reasoning] </Takeaway>
      <Trace>
         [List of Reflection Ids]
         <Reflection ID> ... </Reflection ID>
         ...
      </Trace>
      </Reflection>

      (...total: 5-8 reflections max...)

   </Reflection Item>
</Reflection Cluster>

---

### Final Notes

- Don't change <Reflection Cluster Descriptor> text.
- Every reflection must be concrete (target specific reasoning output) and actionable (offer clear ways to improve).
- Prefer high-score items unless merging improves clarity.
- Don't include vague suggestions or general advice.
- Don't output boilerplate or explanation. Only the final XML structure.
- Always populate all fields of a summarized reflection; never use the [REDACTED] placeholder, expand them, or guess what was removed.
- If no given reflection matches, simply output nothing.

Now read the input clusters and produce a compressed, traceable output with 5-8 total high-impact reflections:
```

Figure 41
Prompt: Reflection Usability Evaluation (All Variants)

```
You are a post-hoc evaluation module for a memory-augmented reasoning system.
Your task is to score how **useful** each provided reflection was for the model's most recent output.
These reflections were injected into working memory **before** the model generated its response.

You will receive:
- the original instruction used to guide the model (<Instruction>)
- the task the model was asked to solve (<Task>)
- the model's final output (<Reasoning Output>)
- the list of reflections that were made available at generation time (<Reflections>)

---

### What You Are Scoring

You must determine, for **each reflection**, how much it *actually influenced or improved* the
reasoning output.
This score is called the **Usability Score**. It should reflect:
- Whether the reflection appears to have changed or guided how the model reasoned
- Whether any of its advice or learnings are **visible** in the output
- Whether it helped avoid errors the reflection was meant to prevent
- Whether the output would likely have been worse *without* it

Do not evaluate whether the reflection is *good in general*, only how useful it was for *this specific
output*.

---

### Score Scale (1 to 10):

Use the following rubric:

**1–3**: The reflection was irrelevant or had no detectable influence.

**4–6**: The reflection may have helped somewhat or indirectly, but its impact is unclear or partial.

**7–9**: The reflection clearly shaped the reasoning or helped avoid likely mistakes.

**10**: The reflection was directly and significantly responsible for the structure or correctness of the
output.

---

### Output Format

For each reflection, **in order of appearance**, return an evaluation of the following format:

<Usability Evaluation>
  <Justification> [Brief reason why the score was chosen] </Justification>
  <Usability Score> [1–10] </Usability Score>
</Usability Evaluation>

---

### Input

<Original Instruction> ... </Original Instruction>
<Reflections> [Each reflection is structured as:
  <Reflection Item>
    <Reflection Summary> ... </Reflection Summary>
    <Error> ... </Error>
    <Error Explanation> ... </Error Explanation>
    <Error Fixing Suggestion> ... </Error Fixing Suggestion>
    <Takeaway> ... </Takeaway>
  </Reflection Item>
]
</Reflections>
<Original Task> ... </Original Task>
<Original Reasoning> ... </Original Reasoning>

---

Evaluate the reflections **one by one, in order of appearance**, score them according to their **actual
influence**, and return your results using the specified output format.
```

Figure 42
Prompt: `find_useful_items_for` *(Filter and Ranking), LLM-Buffer (Trees / Chunking-Enabled)*

```
You are a teacher working closely with a student to help them reason through an upcoming task.
Together, you're reviewing the student's personal archive of past material — including both individual reflections and summarized clusters of such.

Your goal is not to summarize these, but to **decide which items are even worth considering** for the upcoming task.
Some learnings may be unrelated, or based on reasoning patterns that don't match.
Others — whether individual reflections or higher-level summaries — may offer transferrable strategic themes or directly relevant advice.

This step comes **before** summarization of relevant reflections and/or clusters of such. It is a collaborative filter:
→ **Which items might be helpful for this task?**
→ **Which ones can we safely ignore?**

Each item includes performance scores from previous tasks. These scores are useful signals — but your shared judgment decides what's worth ranking.

---

## CANDIDATE MATERIAL

You will receive a list of candidate items. Each one may be:

- A **reflection**: An individual past mistake, its explanation, a fixing suggestion, and a takeaway.
- A **summary**: A cluster-level descriptor of recurring reasoning failures and their strategic characteristics, compiled from multiple related reflections that will be considered, if
selected.

Each item includes:

- <ID>: A unique numeric identifier
- <Type>: Either "Reflection" or "Summary"

If the <Type> is "Reflection", it also contains:
- <Error>: The specific reasoning flaw or theme
- <Error Explanation>: Why it matters
- <Error Fixing Suggestion>: A concrete improvement strategy
- <Takeaway>: A transferrable principle or insight

All items may include:

- <Scores>:
    - Repetition Score
    - Access Score
    - Usability Score
    - Combined Score

**CAUTION**: Some items originate from other tasks.
Do **not** assume that an item is relevant just because it shares a task structure — or irrelevant just because it doesn't.
What matters is whether it *actually helps* for this reasoning challenge.

---

## SCORE MEANING

You don't need to do any calculations. Use provided scores, if any, to support your judgment — but be cautious:

**Scores might be misleading** — sometimes a great insight hasn't had enough time to shine.

- **Repetition Score** — How often this item (or its pattern) reappeared
- **Access Score** — How often it was chosen explicitly
- **Usability Score** — How helpful it has been estimated to be in prior contexts
- **Combined Score** — A rough summary of the above

Use these scores to:
- Support your ranking — but never rely on them blindly
- Elevate high-score items *only if they're relevant*
- Keep low-score items *if* they clearly help with this task

---

## TASK CONTEXT

**This** is the task you need to prepare your student for:

<Student Task>
    {student_task_context}
</Student Task>

---

## WHAT TO DO

1. Evaluate each item for how **useful it would be in helping the student complete this task**.
2. Keep any item that is clearly relevant.
3. Rank all retained items in **descending order of usefulness**.
4. For each ranked item, provide a short reason that explains why it might help.

You may stop ranking once it's clear that no remaining items are useful.
Do not rank marginal items just to be exhaustive.

---

### For each item, ask:

#### If the item is a **Reflection**:
- "Would this specific mistake and its fix directly support this task?"
- "Is the reasoning style, pattern, or lesson reusable here?"

#### If the item is a **Summary**:
- "Does this summary describe a type of reasoning failure that's likely to occur in this task?"
- "Does its descriptor suggest useful themes or failure modes to be aware of?"
- "Is the level of abstraction too general — or just right — for the current task?"

#### For all items:
- "How helpful would this item be compared to the others?"
- "Does this item risk misleading us if it doesn't match the task structure or reasoning demands?"

---

## OUTPUT FORMAT

Output a ranked list of items, in the following XML-style format:

<Task-Relevant Items>
    <Ranked Item>
        <ID> [int] </ID>
        <Rank> [int] </Rank>
        <Reason> [Why does this item receive this rank?] </Reason>
    </Ranked Item>

    ...

</Task-Relevant Items>

Only include items that you believe would actually help with this task.
Never assume relevance. Never rank something unless you have a reason.
Do not hallucinate missing fields or fill in what isn't provided.

---

Begin your filtering and ranking.
```

Figure 43
Prompt: `find_useful_items_for` *(Filter and Ranking), LLM-Buffer (Flat / Non-Chunking)*

```
You are a teacher working closely with a student to help them reason through a new task.
   Together, you're reviewing the student's personal archive of past reflections — reflections on previous reasoning mistakes, improvements, and general insights.

   Your goal is not to summarize these lessons, but to **decide which ones are even worth considering** for this task. Some learnings may be unrelated, or based on reasoning patterns that don't match. Others may offer transferrable strategies or directly relevant advice.

   This step comes **before** summarization. It is a collaborative filter:
   → **Which lessons might be helpful for this task?**
   → **Which ones can we safely ignore?**

   Each reflection includes scores that reflect how often it was repeated, reused, or helped in previous tasks. These scores are useful signals — but your shared judgment decides what's worth ranking.

   ---

   ## CANDIDATE REFLECTIONS

   You will receive a list of candidate reflections. Each one includes:

   - <Reflection ID>: A unique numeric identifier
   - <Error>: The exact reasoning step that was incorrect
   - <Error Explanation>: A detailed explanation of the mistake
   - <Error Fixing Suggestion>: A concrete step-by-step improvement strategy
   - <Takeaway>: A generalized insight or transferrable principle
   - <Scores>:
      - Repetition Score
      - Access Score
      - Usability Score
      - Combined Score

   **CAUTION**: Some reflections originate from other tasks — their formats may differ.
   Do **not** assume that a reflection is relevant just because it shares a task structure — or irrelevant just because it doesn't.
   What matters is whether it *actually helps* for this reasoning challenge.

   ---

   ## SCORE MEANING

   You don't need to do any calculations. Use the scores to support your judgment, but be cautious:
   **Scores might be misleading, simply because a good reflection takes time to stand out!**

   - **Repetition Score** — How often this reflection reappeared previously
   - **Access Score** — How often it was chosen explicitly for reuse
   - **Usability Score** — How helpful it has been in prior reasoning contexts
   - **Combined Score** — A rough blend of the above; not always perfect

   Use these scores to:

   - Support your ranking — but never rely on them blindly
   - Elevate high-score items *only if they're relevant*
   - Keep low-score items *if* they clearly help with this task

   ---

   ## TASK CONTEXT

   **This** is the task you need to prepare your student for:

   <Student Task>
      {query_llm_input}
   </Student Task>

   ---

   ## WHAT TO DO

   1. Evaluate each reflection for how **useful it would be in helping the student complete this task**.
   2. Keep any reflection that is clearly relevant or reusable in some way.
   3. Rank all retained reflections in **descending order of usefulness**.
   4. For each ranked reflection, provide a short reason that explains why it might help.

   You may stop ranking once it's clear that no remaining reflections are useful for this task.
   Do not rank marginal items just to be exhaustive.

   ---

   ### For each reflection, ask:

   - *Would this genuinely help my student on this task?*
   - *How helpful would it be compared to the others?*
   - *Is the reasoning style or fix reusable here?*
   - *Does this reflection risk misleading us if it doesn't match the task structure?*

   ---

   ## OUTPUT FORMAT

   Output a ranked list of reflections, in the following xml style format:

   <Task-Relevant Refelections>
      <Ranked Item>
         <Reflection ID> [int] </Reflection ID>
         <Rank> [int] </Rank>
         <Reason> [Why does this reflection recieve this rank?] </Reason>
      </Ranked Item>

      ...

   </Task-Relevant Refelections>

   Only include reflections that you believe would actually help with this task.
   Never assume relevance. Never rank something unless you have a reason.
   Do not hallucinate missing fields or fill in what isn't provided.

   ---

   Begin your filtering and ranking.
```

Figure 44
Prompt: `match_items` *(Part-Whole Relationship), LLM-Buffer (Both Variants)*

```
You are a teacher guiding your students through a long-term project on **learning to reason better**.

Every week, students submit new reflections on the mistakes they encounter when solving problems.
Over time, common patterns of mistakes and corrections are organized into **clusters** —
each cluster captures a recurring type of reasoning error or learning theme.

As the year progresses, **new types of mistakes** may emerge,
and **existing clusters may need to be expanded or updated**.

Your goal is to structure this archive properly, for your lessons.

---

## YOUR TASK

You are reviewing a new batch of individual student reflections.
For each new reflection, you must decide:

- Does it naturally fit into one of the **existing clusters**?
- Or is it a **new category of mistake** not yet captured by any cluster?

Your goal is to prepare for tomorrow's class discussion,
where you will talk about **what's new** and **how the existing clusters are evolving**.

You must group only what belongs — not force matches.

---

## INPUT FORMAT

You will receive:

1. A list of **trusted clusters** inside a `<Trusted Clusters>` block.
2. A list of **new candidate reflections** inside a `<Candidate Reflections>` block.

Each trusted cluster is structured as:

<Cluster>
  <Reflection Cluster ID> [char] </Reflection Cluster ID>
  <Reflection Cluster Descriptor> ... </Reflection Cluster Descriptor>
</Cluster>

Each candidate reflection is structured as:

<Reflection>
  <Reflection ID> [int] </Reflection ID>
  <Error> ... </Error>
  <Error Explanation> ... </Error Explanation>
  <Error Fixing Suggestion> ... </Error Fixing Suggestion>
  <Takeaway> ... </Takeaway>
</Reflection>

Notes:
- Cluster IDs are **characters** (A, B, C, ...).
- Candidate Reflection IDs are **integers** (1, 2, 3, ...).
- Each trusted cluster summarizes an existing theme.
- Each candidate reflection is a newly submitted learning.
- **Trusted clusters represent previously established and meaningful learning patterns.
Assume their descriptions are reliable — your task is to assign reflections, not to revise or correct the clusters.**

---

## WHAT COUNTS AS A PART-WHOLE MATCH

A reflection belongs to a cluster if:

- The mistake described fits naturally under the broader pattern captured by the cluster descriptor.
- The type of reasoning error aligns meaningfully.
- The fixing strategy would reinforce the broader lesson.
- The takeaway supports the overall learning direction.

**Important:**
- Do **not** match a reflection unless it truly fits conceptually.
- If no fitting cluster exists, leave the reflection unmatched — this signals a **new type of mistake**.
- Precision matters: it's better to identify something new than wrongly categorize something existing.

---

## HOW TO APPROACH MATCHING

- For each candidate reflection:
  - Read all cluster descriptors carefully.
  - Compare the reflection's mistake, explanation, fixing strategy, and takeaway to the cluster's theme.
  - Ask yourself:
    - Would this reflection be a natural example to discuss under this cluster?
    - **Would it help illustrate or expand the cluster's main idea?**
- If multiple clusters seem possible:
  - Select the **best-fitting, most natural** cluster.

---

## CROSS-TASK REASONING REMINDER

Sometimes reflections may fit a cluster even if they originated from different tasks.

- Only match across tasks if the reflection clearly **belongs under the cluster's core idea** and **supports concrete reasoning improvement**.
- Avoid clustering loosely based on surface similarity.
- Prefer strong conceptual fit over task diversity.
- If no cluster matches cleanly, leave the reflection unmatched — it may represent a new, valuable learning theme.

---

## OUTPUT FORMAT

For each candidate, output:

<Part-Whole Matching>
  <Candidate Reflection ID> [int] </Candidate Reflection ID>
  <Best Matching Cluster ID> [char] </Best Matching Cluster ID>
</Part-Whole Matching>

If no suitable cluster is found:

<Part-Whole Matching>
  <Candidate Reflection ID> [int] </Candidate Reflection ID>
  <Best Matching Cluster ID> None </Best Matching Cluster ID>
</Part-Whole Matching>

---

## IMPORTANT REMINDERS

- Be deliberate: Only assign a candidate to a cluster if it **clearly fits** conceptually.
- New mistakes **may emerge** — it's normal that not every reflection matches an existing cluster.
- Choose **the best single cluster** if several fit reasonably well.
- Trust the integrity of the existing clusters — they represent meaningful learning patterns from past weeks.
- Your judgment helps decide whether the student's new reflection **extends the known map** or **opens a new path**.

---

Begin reviewing the reflections.
```

Figure 45
Prompt: `match_items` *(Equivalence Relationship), LLM-Buffer (Both Variants)*

```
You are a teacher carefully reviewing reasoning assignments submitted by your students.
Each student was tasked with identifying a reasoning mistake they made, explaining it, proposing a fix, and reflecting on the general lesson they learned.

You suspect that some students may have **copied ideas** from existing submissions — not just rewording, but reproducing the same mistake, explanation, fix, and takeaway.

Your task is to check each new reflection carefully against a trusted set of existing reflections,
and determine if it **semantically duplicates** any of them — and if so, **identify which one**.

---

## TRUSTED SEARCH SPACE

The existing reflections you compare against are trusted:
They may differ slightly internally, but are considered **a consistent group** for this check.

You do not need to validate or second-guess the internal consistency of the group.
You only need to find **the most clearly matching reflection** among them.

---

## INPUT FORMAT

You will receive:

1. A list of **trusted reflections** inside a `<Trusted Reflections>` block.
2. A list of **new candidate reflections** inside a `<Candidate Reflections>` block.

Each reflection (both trusted and candidate) is structured as:

<Reflection>
  <Reflection ID> ... </Reflection ID>
  <Error> ... </Error>
  <Error Explanation> ... </Error Explanation>
  <Error Fixing Suggestion> ... </Error Fixing Suggestion>
  <Takeaway> ... </Takeaway>
</Reflection>

- **<Reflection ID>** is unique across both trusted and candidate reflections.
- **Trusted reflections** represent previous submissions and have capital letters as IDs.
- **Candidate reflections** represent new assignments you must evaluate and have integer IDs.

**Important:**
- The two lists are separate: never match candidates to candidates, or trusted to trusted.
- That is: Never match an integer ID with another integer ID and never match a character ID with another character ID.
- You only match candidate reflections **against the trusted reflections**.

---

## WHAT COUNTS AS A MATCH

- The <Error> must describe the **same faulty reasoning step**.
- The <Error Explanation> must identify **the same flaw or misunderstanding**.
- The <Fixing Suggestion> must propose **the same core correction strategy**.
- The <Takeaway> must express **the same general principle**.

Minor differences in phrasing are acceptable.
**Substantial differences in meaning or scope are not acceptable.**

---

## CROSS-TASK CAUTION

Sometimes reflections may come from **different tasks** but still seem superficially similar.

Be cautious:

- A reflection from a different task can match **only if** it is **truly semantically identical** across Error, Explanation, Fixing Suggestion, and Takeaway.
- Minor rewording is acceptable; deeper differences in reasoning, context, or intended correction are **not** acceptable.
- **Differences in the underlying task may signal important meaning shifts**, even if the surface reasoning appears close.
- Always prioritize precision:
  - If there is any doubt that the tasks, contexts, or fixes diverge meaningfully, output ***None***.
  - Strict matching is required: better to miss a match than falsely claim equivalence.

Your goal is to catch true duplicates — not loose thematic overlap.

---

## HOW TO APPROACH MATCHING

- For each candidate reflection:
  - Compare it carefully to each trusted reflection.
  - **Reason step-by-step**:
    - Compare meaning across all four fields (Error, Explanation, Fixing Suggestion, Takeaway).
    - Identify if any trusted reflection matches exactly in meaning.
- If multiple reflections match:
  - Select **the most complete and clearest** matching trusted reflection.
- If no clear match is found:
  - Assume **no match** (prioritize precision).
- Ignore surface phrasing differences — focus purely on semantic content.

---

## OUTPUT FORMAT

For each candidate, output:

<Matching Reflection>
  <Candidate Reflection ID> 7 </Candidate Reflection ID>
  <Equivalent Trusted Reflection ID> A </Equivalent Trusted Reflection ID>
</Matching Reflection>

If no match is found:

<Matching Reflection>
  <Candidate Reflection ID> 7 </Candidate Reflection ID>
  <Equivalent Trusted Reflection ID> None </Equivalent Trusted Reflection ID>
</Matching Reflection>

---

## IMPORTANT REMINDERS

- You must only declare a match if you are highly confident that all parts of the reflections are **semantically identical**.
- **If in doubt, output "None".**
- Matching must be strict — recognizing exact duplication of reasoning content, not loose thematic similarity.
- Trust the integrity of the search space; your role is to **find the clearest match**, not to question grouping.

---

Begin reviewing the reflections.
```

Figure 46
Prompt: `group_items`, LLM-Buffer (Both Variants)

```
You are a teacher guiding your student in learning how to reason more effectively.
Today, you are reviewing a collection of homework submissions — each one reflecting a reasoning mistake the student identified, explained, and proposed a fix for.
Your goal is to **organize these reflections into meaningful groups**.

Tomorrow, you and your student will review each group together — discussing common types of mistakes, and how to improve reasoning strategies systematically.

For today, your task is to:
- Group similar reflections into coherent categories.
- Choose a short, clear label for each group (like naming a lesson or workshop topic).
- Make sure each group covers **the same kind of mistake, fix, or task** — not just surface similarities.
- If a reflection doesn't fit well with others, give it its own small group.

This is not about summarizing the content — that will happen later.
But your grouping should be **clear enough** that future summarization will be easy and natural.

---

## CANDIDATE REFLECTIONS

Each reflections includes:
- <Reflection ID>
- <Error>: A specific reasoning step that was wrong
- <Error Explanation>: Why it was wrong
- <Fixing Suggestion>: How to reason better
- <Takeaway>: A general principle or lesson

You should focus on the **conceptual similarity** between learnings:
- Are these failures of the same kind?
- Would you teach them together in a classroom setting?
- Do they share the same kind of fix or correction strategy?

---

## OUTPUT FORMAT

Output your clusterin in the following format:

<Clustered Reflections>
  <Cluster>
    <Cluster Descriptor> Jumping to Conclusions </Cluster Descriptor>
    <Reflection IDs>
      <Reflection ID> 7 </Reflection ID>
      <Reflection ID> 12 </Reflection ID>
    </Reflection IDs>
    <Reason for Clustering> [What makes these items similar? Justify!] </Reason for Clustering>
  </Cluster>

  ...

</Clustered Reflections>

---

## GUIDELINES

- You may create as many clusters as needed.
- Each cluster should be semantically coherent — same type of error, fix, lesson, or task.
- Cluster descriptors should be concise but specific (think of naming a short workshop or lesson).
- If a reflection clearly stands alone, place it in a singleton cluster.

---

## FINAL INSTRUCTIONS

Before producing the <Clustered Reflections> output:

1. Reason step-by-step:
  - Which reflections belong together?
  - What concept, error type, or fixing strategy connects them?
  - Why do they belong in the same group?

2. Then, output the clusters in the specified XML format.

---

Begin organizing the following reflections.
```

Declaration of Academic Integrity

I confirm that this thesis is my own work and I have documented all sources and material used.

I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here.

This thesis was not previously presented to another examination board and has not been published.

Garching, 29.07.2025 Alexander Blatzheim

Note on Employed Tools

To improve the readability of this thesis, I, Alexander Blatzheim, made limited use of AI-based writing assistants. These tools were exclusively employed to enhance the flow and clarity of the text. All intellectual contributions, including research, design, and analysis, are my own.