ChatGPT

Kimera SWM: Semantic Working Memory System – Technical White Paper

Abstract

Kimera SWM is a novel semantic working memory architecture that uses principles from thermodynamic entropy and information theory to manage knowledge storage and retrieval with engineering rigor. This white paper introduces the concept of **scars** as durable memory/event units and an **EchoForm** grammar for encoding events, ensuring that each memory is structured and semantically rich. The system's architecture is self-contained and modular, avoiding external frameworks while maintaining internal coherence. We model memory in thermodynamic terms, defining *semantic entropy* to measure information content and introducing the notion of *entropy tension fields* that drive memory compression and recall. Memory management employs entropy-based pruning (inspired by NEPENTHE and MIPP algorithms) to remove low-information content while preserving essential mutual information [1] [2]. A dedicated contradiction engine monitors and resolves inconsistencies, and memory access is achieved through non-tokenized, event-driven cues rather than linear text tokens. We provide implementation guidelines detailing data structures, code-level considerations, and computational costs to ensure the design is feasible. We also evaluate potential blind spots and opportunities for improvement, such as incorporating entropy-aware hierarchical memory layers and addressing naive design assumptions. Finally, we discuss future development scenarios including scaling to multimodal inputs, enabling autonomous semantic adaptation, and deploying Kimera SWM on edge AI systems. All proposed mechanisms are grounded in known physics and information theory, ensuring they are testable and implementable with current technology rather than speculative. This comprehensive paper is intended for system architects, engineers, and researchers interested in memory systems that unite semantic structure with thermodynamic principles.

Introduction

Artificial intelligence agents and cognitive systems increasingly require robust working memory to store and reason over accumulated knowledge in real time. However, conventional memory strategies—such as fixed-size context windows in large language models or simple vector databases for retrieval—have clear limitations. For example, today's large language models have a limited "temporal cognition" and cannot autonomously recall earlier interactions unless explicitly provided in the prompt [3]. This leads to inefficiencies and disjointed long-term behavior. There is a need for a **Semantic Working Memory (SWM)** system that can retain and manage meaningful events over time, enabling continuity and context awareness beyond short context windows.

Kimera SWM is proposed as a self-contained memory architecture that addresses these challenges by treating knowledge as a dynamic thermodynamic system. By "semantic working memory," we mean a memory store focused on the meaning of events (who did what, when, why, with what result), rather than raw token sequences. Kimera SWM operates on **event units** encoded in a structured form, allowing it to recall information based on situations or contexts rather than exact word matches. This approach contrasts

with token-based memory: instead of retrieving text by string similarity, the system recalls whole events or facts when relevant situations arise.

The design of Kimera SWM is grounded in well-established principles for realism. We incorporate **information-theoretic and thermodynamic principles** to guide memory management. In physical terms, erasing or adding information has energy and entropy implications – Landauer's principle states that erasing a single bit of information requires a minimum energy cost of $k_B T \ln 2$ at temperature T [4]. While our implementation is in software, this principle motivates an entropy-conscious approach: the system should minimize arbitrary information deletion and compression to conserve "energy" (computational resources) and preserve valuable data. By framing memory operations in terms of entropy, we ensure that every design choice (from pruning data to recalling events) can be quantified and justified rigorously, avoiding magic or speculation.

In summary, the introduction of Kimera SWM is driven by the convergence of two needs: - **Cognitive Continuity:** AI systems must remember important events and facts over long durations without human-supervised context resets. - **Efficiency and Realism:** Memory management should obey information-theoretic constraints and physical analogies, ensuring that methods like forgetting, compression, and retrieval are optimal and implementable in real engines (with finite memory and computation).

The following sections detail the theoretical foundations of Kimera SWM, its architecture, the entropy-based models at its core, strategies for memory pruning and contradiction handling, implementation considerations, and an evaluation of limitations and future directions. Throughout, we maintain a focus on engineering realism: every component could be built with today's technology and is informed by existing research or principles, not sci-fi speculation.

Motivations and Theoretical Foundations

Fundamental Memory Units – "Scars": Kimera SWM introduces **scars** as the fundamental units of memory. A "scar" represents a single event or experience encoded into the system's working memory. The terminology evokes how significant events leave lasting marks (as scars) on an entity. In psychological and physical contexts, traces of past events persist – **the present is full of traces of the past (footsteps in sand, scars on skin, craters on the moon), whereas we have no traces of future events** [5]. This asymmetry is tied to the thermodynamic arrow of time; it's natural for past events to leave information residues in a system [6]. By modeling memory events as scars, we capture this notion that each meaningful occurrence leaves an imprint on the knowledge base that can later be recalled. Just as a scar in biology is a healed wound that signifies a past incident, a memory scar is a processed, stored representation of an event that the system has "healed from" (integrated) but not forgotten.

Each scar is meant to be *atomic* at an event level – it could be a specific experience (e.g., an agent's interaction at a certain time) or a learned fact. The decision to use events as the grain of memory is supported by cognitive science: episodic memory research suggests that **events form the natural episodes of memory** [7]. By organizing memory around discrete events, we align with how humans segment experiences and how episodic memory encodes information (each episode having a who/what/when/where). This structure helps the system to index and retrieve information in a meaningful way, as opposed to dealing with undifferentiated streams of data.

EchoForm – Grammar for Event Encoding: To store events as scars, we require a consistent representation format. Kimera SWM employs a formal event-encoding grammar called **EchoForm**. EchoForm serves as the schema or "language" in which all events are recorded. It is designed to encapsulate the key semantic components of an experience in a standardized way. For example, an event encoded in EchoForm might include fields such as: **Actors** (entities involved), **Action** (the verb or change that occurred), **Objects/ Targets** (entities affected), **Time/Context** (temporal and situational context), **Outcome** (result of the action), and **Causal Links** (what prior events led to this event, and what future events might be triggered). This is analogous to defining a grammar for sentences, but here it's a grammar for multi-faceted events.

By using a grammar, we ensure that every scar has an *internal structure* that can be parsed and reasoned about. EchoForm is essentially a domain-specific data model for events; it can be thought of as a combination of a semantic frame (from linguistics) and a knowledge graph entry. For instance, an event like *"Robot R1 [Actor] delivered package P123 [Object] to Location L5 at 10:00 AM [Time]"* would be stored in a structured form with those labeled slots. The **grammar enforces consistency** – every event record must specify certain fields – and this yields several benefits: - *Semantic Integrity:* Important attributes (actors, time, etc.) are never missing, so the system can always understand the basic who/what/when of an event. - *Comparability:* Because events follow the same schema, the system can compare two events field by field (e.g. check if they share an actor or location) without needing complex NLP on raw text. - *Compressibility:* A structured format is more amenable to compression and summarization. Repeated values or patterns in the same field across multiple events can be identified and utilized to reduce storage or to abstract common patterns.

The term "EchoForm" highlights that each event encoding is like an **echo of the actual occurrence** – it's a concise reverberation of what happened, stripped of extraneous detail but preserving core meaning. It provides a grammar in the same sense that **Backus-Naur Form (BNF)** provides a grammar for programming languages, but here it is for real-world events. While the exact specification of EchoForm may vary by application domain (e.g., different fields for a dialogue event versus a robotic sensor event), the principle is that a strict schema exists and is adhered to by the system's memory writer.

Semantic Thermodynamics Basis: Underlying the concept of scars and EchoForm is a theoretical stance that memory can be treated with *thermodynamic analogies*. Each stored event carries a certain **information entropy** reflecting how much uncertainty or surprise it contains. We posit that semantic information can be quantified analogous to Shannon entropy: if an event's outcome was highly unpredictable given prior knowledge, encoding that event provides a large information gain (high entropy reduction). Conversely, a very expected event (one that the system could have predicted) carries less new information. We leverage this intuition extensively in later sections, but it's a core foundation: **memory management should prioritize high-information (high entropy) content while trimming low-information (redundant) content**.

A key theoretical principle guiding us is the **conservation of information**. In information theory, mutual information measures how much knowing one variable tells us about another. We aim to store events such that the *meaningful mutual information* between events (like cause-effect relationships, or recurring actors) is preserved even if we compress or prune some details. This is inspired by pruning techniques like MIPP (Mutual Information Preserving Pruning), which ensure that when simplifying a system, the mutual information between layers (or, in our case, between memory elements) remains intact [8] [9]. Thus, one theoretical goal is to never prune away an event that serves as a critical link (information conduit) between other events.

Finally, it is worth noting that our approach aligns with physical reality: systems create records of the past by increasing local order at the expense of increased entropy elsewhere. By storing a memory (creating order), the system may incur a cost (computational work or energy usage) that manifests as entropy expelled to the environment (for example, CPU heat). We keep this balance in mind so that Kimera SWM will manage memory in a way analogous to a thermodynamic system seeking equilibrium.

In summary, the motivations for Kimera SWM are rooted in: - Treating events as primary memory units (scars) to mirror how real experiences leave lasting, retrievable marks. - Establishing a strict but flexible event encoding (EchoForm) to ensure semantic clarity and consistency. - Grounding memory operations in entropy and information-theoretic principles, providing a physics-inspired backbone that informs how we decide what to keep, compress, or discard.

With these foundations, we can now describe the overall system architecture and how these ideas come together in implementation.

System Architecture (Modular & Self-Contained)

Kimera SWM is designed as an **internally coherent, modular architecture** with no reliance on external frameworks or proprietary cloud services. This section outlines its main components and how they interact, ensuring the system is both self-sufficient and adaptable.

- **1. Event Ingestion & Encoding Module:** This front-end module receives raw inputs (which could be sensor data, natural language statements, logs of actions, etc., depending on the AI agent's context) and converts them into the structured EchoForm format. Think of this as a parser or translator that takes unstructured or semi-structured data from the agent's experiences and produces a well-formed scar. For example, if the agent just completed a task or the user provided a new piece of information, this module identifies the key semantic elements and populates an EchoForm record. Its design is analogous to an ETL (Extract-Transform-Load) pipeline in data engineering: extract key facts, transform into standardized schema, load into memory.

- **2. Semantic Memory Store (Scars Database):** At the heart of Kimera SWM is the **memory store** that holds all the scars (encoded events). This can be conceptualized as a **knowledge base** optimized for event records. Implementation-wise, it could be as simple as an in-memory table or list of event objects, or more sophisticated like a graph structure where nodes represent entities and events form links (or vice versa). The key is that it's optimized for *semantic queries* rather than textual ones. Each entry in this store is an EchoForm-structured event with fields that can be indexed. The memory store does not depend on an external database; instead, it might utilize efficient data structures like hash maps for quick lookup (e.g., an index mapping from an entity ID to all scars involving that entity) and possibly a lightweight custom file format for persistence if needed. The store is modular in that it doesn't need to know *why* an event is being stored or retrieved – it just reliably holds and returns scars.

- **3. Entropy Monitor & Thermodynamic Manager:** This component continuously assesses the "entropy state" of the memory store. It assigns entropy values to individual events and to subsets of memory, capturing how much uncertainty or information each contains. For instance, it may compute a running measure of how frequently certain event patterns appear, which affects their surprise level. This module functions analogously to a thermostat or load balancer: if the memory

store grows too chaotic (high entropy) or too complacent (very low entropy), the manager takes action (triggers compression, summarization, or prompts for novel data injection). We can imagine it maintaining an **entropy tension map** across the memory—areas (clusters of events) with high entropy vs low entropy. An imbalance (a high gradient between a chaotic cluster and an overly orderly cluster) is treated like a pressure differential that should be evened out by some operation (like recalling related info to clarify, or pruning redundant data to reduce noise). We will detail these operations in the next section on entropy modeling, but architecturally, this manager is the logic that decides when to apply those operations.

- **4. Pruning & Consolidation Module:** Working closely with the entropy monitor is the pruning module, responsible for implementing **memory cleanup strategies**. This module is inspired by algorithms such as NEPENTHE and MIPP from neural network pruning. For example, if the entropy monitor flags that a subset of memory events is low entropy (i.e., they are highly predictable or repetitive), the pruning module may remove or consolidate them. Consolidation could mean summarizing several low-entropy events into one higher-level event (thus reducing total count while preserving info). The module ensures this is done modularly: it doesn't break the system if removed, and it doesn't require an outside service to decide what to prune—the logic is encapsulated (e.g., thresholds and rules set by engineers or learned over time). We ensure that *mutual information is preserved* during pruning – akin to how MIPP prunes neurons that don't transfer entropy [2], we prune memory that doesn't transfer information to other memory (for instance, an isolated trivial fact that has never been referenced and has no inferred links can be safely dropped). By design, this module runs asynchronously to avoid halting the main operations; it can operate during idle times or background threads.

- **5. Contradiction Detection & Resolution Engine:** Another critical module is the **contradiction engine**, which scans and maintains the logical consistency of the stored events. Each time a new scar is added, or periodically in batch, this engine checks for conflicts: cases where two scars represent mutually exclusive states of the world. For example, if one memory scar says "Device X's status = ON at time 12:00" and another says "Device X's status = OFF at 12:00" (with overlapping context), we have a contradiction. The engine uses rules or an inconsistency logic to flag these. This design echoes a **Truth Maintenance System (TMS)** in classical AI, but here it's informed by our semantic-entropy approach: contradictions are effectively *high entropy regions* (because they introduce maximum uncertainty – the system doesn't know which fact is true). The contradiction engine may resolve issues by either reconciling (e.g., adding context: both could be true if they refer to different conditions), or by prioritizing one memory over another (based on credibility or recency), or by marking the information as uncertain. Notably, a *recursive contradiction engine* approach can be taken: similar to the Aletheion Agent, which recursively probes a belief system for stability [10], our engine can simulate the consequences of a set of beliefs to see if they lead to any contradiction down the line. Architecturally, this is an internal service that can query the memory store and update a "belief state" labeling (each scar could have a flag: consistent, contradicted, resolved, etc.). By modularizing it, we could disable or tweak contradiction handling rules without affecting the rest of memory functions.

- **6. Memory Access Interface (Event-Driven Recall):** This module handles queries and retrievals from the memory store. It is designed to allow **event-driven access** as opposed to raw text or key-based lookup. In practice, this means the interface accepts rich queries like *"What events involve Entity Y in Context Z?"* or triggers recall automatically when a new input resembles a stored event's context.

Internally, it uses the indices and structure of the memory store to fetch relevant scars. For example, if the agent's current situation (context) is similar to a past context stored in memory, the entropy monitor might note a rising prediction error (entropy) and signal this module to recall similar past events to guide the agent. Because the memory entries are structured, this retrieval can be done efficiently by matching fields rather than scanning raw text. For instance, if the query is an incomplete event (some fields known, others blank), the system finds stored events that match the known fields. This is analogous to a database query or an associative recall in cognitive architectures (similar to how ACT-R can retrieve a chunk by partial matching). **Non-tokenized** means that we are not relying on word embeddings or large text vectors alone; however, we might still use vector representations for efficiency (e.g., each event could have an embedding representing its semantic content for quick nearest-neighbor search). Importantly, the module can proactively push memories: if a certain pattern is detected (like a repeat of a prior scenario), it can alert the agent or other modules by providing the relevant memory, without being explicitly asked. This event-driven paradigm ensures that memory recall is situationally appropriate and not just a simple key-value store lookup.

All these modules are designed to work in concert within Kimera SWM. The architecture is **modular** in the sense that each component has a clear role and interface: - The ingestion module writes to the memory store. - The entropy monitor oversees the state and triggers pruning or recall actions. - The pruning module modifies the memory store (under guardrails to not lose important info). - The contradiction engine reads from the store and may annotate or adjust entries (but typically doesn't remove – that's left to pruning unless a piece of info is deemed entirely false). - The access interface provides read (and possibly complex query) capabilities to the rest of the AI agent or system.

Furthermore, the entire system is self-contained: it can be implemented with basic data structures and algorithms. We do not assume any proprietary technology (no reliance on cloud APIs or heavy databases). This means an implementation could be done in a systems programming language (C++/Rust) or a high-level one (Python/Java) using built-in libraries. The modularity also aids testability – e.g., one can unit test the contradiction engine with simulated memory inputs, or test the entropy monitor's calculation on a known set of events.

In the next sections, we dive deeper into how semantic entropy is modeled in this architecture and how it drives the behavior of components like the entropy monitor and pruning module.

Semantic Thermodynamic Entropy Modeling

A distinguishing feature of Kimera SWM is its explicit modeling of **semantic entropy** in the working memory. We borrow concepts from thermodynamics and information theory to quantify and manage the "disorder" and "information content" within the memory system. This section explains how we define entropy in a semantic context and how *entropy tension fields*, compression, and recall mechanisms work in practice.

Defining Semantic Entropy: In classical information theory, Shannon entropy $H = -\sum p_i \log_2 p_i$ measures the uncertainty of a random variable (in bits). In our memory system, we define **semantic entropy** H_s as a measure of uncertainty or unpredictability in the content of memory events. One way to operationalize this is to consider the probability distribution of certain attributes or outcomes in events. For example, imagine the memory has a field "Outcome" for events. If across all stored events the outcome is

almost always “success” with probability 0.99, then that field has low entropy (it’s very predictable), whereas if the outcomes are diverse (each with p ~ 0.1), entropy is high. More generally, each field of the EchoForm could have an entropy (e.g., entropy of distribution of actions, entropy of distribution of involved entities, etc.), and an individual event could have an entropy value reflecting how surprising that event is relative to prior expectations.

Mathematically, suppose we have a model that can assign a prior probability to an event (based on past data or an internal predictive model). The **surprisal** of that event is $-\log_2 P(\text{event})$. We can interpret that as the information content (in bits) of the event. The semantic entropy of the memory can then be thought of as the expected surprisal of events the system encounters or stores. We ensure that **semantic entropy is bounded by syntactic entropy** – meaning the entropy in the abstract meaning of events is not more than the entropy in their raw representation (this aligns with the idea that structure and semantics constrain possibilities, reducing uncertainty compared to raw data). While we won’t dwell on a formal proof, intuitively, having a structured grammar (EchoForm) limits the space of possible events compared to arbitrary data, so it imposes a limit $H_s \le H$.

Entropy Tension Fields: We introduce the concept of an *entropy tension field* to describe how entropy is spatially (or rather, contextually) distributed in the memory system. Consider plotting the memory as a network graph: nodes could be entities or states, and edges could be events linking them. Each part of this graph can have a local entropy measure (how unpredictable are the events in that region of the graph?). For instance, events involving a particular device might be very routine (low entropy), whereas events involving a human user might be more varied (higher entropy). This creates a gradient or tension between different parts of memory. Just as in physics a difference in pressure or temperature creates a *field* that causes movement (air flows from high pressure to low, heat flows from hot to cold), here an entropy gradient prompts information flow or reorganization.

What does that mean concretely? If one region of memory is highly disordered (lots of conflicting or highly varied information) and another is very orderly, the system will attempt to reduce that tension: - It might *recall and compare events* between the regions to see if the disorder can be explained or resolved. This is akin to mixing to equilibrate knowledge – e.g., if subsystem A is very uncertain and subsystem B has stable patterns, maybe pulling some patterns from B to interpret A will reduce uncertainty in A. - It could *trigger learning or querying for missing information*: a high entropy region indicates the system is very unsure about something (many possibilities remain). The system might flag this as an area to gather more data or to ask a user for clarification, thus importing some low-entropy (clarifying) data to reduce uncertainty.

In practice, we implement entropy tension monitoring by computing entropy over sliding windows or clusters of events. For example, the entropy monitor might calculate entropy for events grouped by time window (e.g., entropy of events this week vs last week) or by topic (entropy of events related to “navigation” vs “communication”). If a significant difference is found, that indicates a tension. This field concept is a guide for **memory consolidation**: the system might decide to merge a high-entropy cluster with a low-entropy cluster by finding commonalities (thus raising the low side’s entropy slightly and lowering the high side’s entropy, moving toward equilibrium). In essence, *entropy tension fields drive the dynamic balancing of memory*, ensuring no part of the memory becomes too chaotic or too stagnant without intervention.

Compression Mechanisms: Compression in Kimera SWM is the process of reducing entropy (and data size) while preserving semantic content. Drawing a thermodynamic analogy, compression corresponds to the system doing work to reduce disorder (like compressing a gas into a smaller volume reduces entropy but

requires work/energy). We treat unnecessary details and redundancies in memory as entropy that can be purged or condensed. For example: - If multiple scars record very similar events (e.g., a robot delivers items to various locations daily with only minor differences), the system can compress these into a summary or a prototype event. Instead of storing 100 nearly identical delivery events, it might store one template event "Robot delivered a package to location X (various times)" plus some statistical info. This is like finding regularity and thereby encoding it with fewer bits. - If an event's description is very detailed, but not all details are needed for future reasoning, the system can encode it in a more abstract form (like dropping or generalizing certain fields). This is analogous to data compression by removing irrelevant or low-information content.

The EchoForm grammar assists compression because events are structured; the system can systematically remove or merge specific fields. For instance, it could compress out the exact timestamp of an event if only the date is important, reducing resolution (and entropy). Or combine two sequential events into one if they logically form a single higher-level event (like two sub-actions that always happen together could be stored as one action).

A key guide for compression is the **entropy threshold**: if removing or merging information does not significantly increase the unpredictability of the system's knowledge (i.e., doesn't increase entropy beyond a threshold), then it's deemed safe. We take inspiration from NEPENTHE here: in neural nets, entire layers with low entropy contribution can be removed with little performance loss [11]. Similarly, in memory, if a collection of facts yields little new information, we can remove or simplify them with little loss to the system's overall capability. By quantifying "little loss" as negligible change in entropy or mutual information, we keep this rigorous. The **NEPENTHE principle** in our context becomes: *remove memory elements that contribute minimal entropy to the knowledge state, so long as their removal doesn't affect the outcome of queries or reasoning*. Empirical design might define "minimal entropy contribution" as, say, an event that is almost entirely predictable from other events.

One practical compression strategy is **entropy-based clustering**: cluster events such that each cluster's internal entropy is low (members are similar). Then represent that cluster by one prototype event plus some variance description. This way, memory storage is reduced and when recalling, we can instantiate variations as needed. This directly reduces memory size and ensures the highest-level summary has the most salient info (high entropy events stand out, routine ones collapse together).

Memory Recall and Expansion: In thermodynamics, if compression is analogous to reducing entropy, expansion is increasing entropy. Memory recall can be seen as a controlled expansion of compressed information. When the system needs details that were abstracted away, it must "decompress" something. For instance, if a summary event was stored for routine deliveries, and now the agent needs to remember a specific delivery (because an anomaly occurred that day), the memory system may retrieve the summary and then fetch the particular details from an archive or by inference. If the exact details were pruned, the system might at least know that it's missing detail and could flag that external input is needed (or that it should have retained it – a learning experience to adjust thresholds).

In our design, whenever compression happens, we consider storing the discarded details in a secondary storage (like a long-term archive or log) that's not actively used but can be consulted if needed. This is akin to a **two-tier memory**: an active working memory (compressed, low entropy, minimal but sufficient) and a passive long-term memory or archive (higher entropy, full detail, maybe slower to access). If a query cannot be satisfied by the active memory (because details were compressed out), the system can retrieve from the

archive and in doing so, effectively expand the memory representation temporarily. This ensures no permanent loss; it's like reversible compression where possible.

Even without an archive, memory recall itself can increase entropy in the working set temporarily: when the agent recalls multiple past events to compare, it is bringing a lot of information into focus at once. We ensure this doesn't overwhelm the system by controlling recall through cues and relevance. The design borrows from human memory models where *cues trigger recall of related memories* [12]. For example, a context cue (the current situation) will raise the recall probability of certain scars if their content is relevant and not too old [13]. We model recall probability with factors like contextual similarity and time elapsed, as others have proposed [13]. Only if the probability (or a relevancy score) exceeds a threshold will the memory be actively recalled into working context. This prevents random high-entropy bits from flooding the agent's mind; recall is constrained to what likely matters now.

Lastly, we note an intriguing balance observed in an **Entropic Associative Memory model**: there exists an optimal entropy range for memory where both precision and recall are high [14]. If memory entropy is too low (over-compressed, over-pruned), precision in matching queries is high (because few ambiguous data) but recall capability is low (many items were discarded). If entropy is too high (memory is too noisy/disordered), recall might retrieve a lot but precision suffers (many irrelevant or wrong associations). An intermediate entropy yields the best of both [14]. Kimera SWM's entropy modeling explicitly aims to keep the memory in that optimal band. The entropy tension field logic and compression/expansion mechanisms all serve to avoid extremes of chaos or blank order. By treating entropy as a tunable parameter, we can dynamically adjust memory content to maintain this sweet spot. This is a realistic approach: it mimics how the brain forgets some details but not all, maintaining a balance where it has sufficient info to function without being overwhelmed.

In conclusion, semantic thermodynamic entropy modeling in Kimera SWM provides: - A **measurement framework** (entropy of events, clusters, whole memory) to guide decisions. - A **dynamic mechanism** (entropy tension fields) that triggers balancing operations. - Tools for **compression** (entropy-based pruning and summarization) to reduce memory load logically. - A strategy for **recall** (entropy-driven retrieval cues and possible expansion from archives) to ensure needed details can resurface. - The assurance that these are done with quantifiable effects, not arbitrarily – consistent with known theories and analogies to physical processes.

With this model in mind, we can now detail how pruning and memory management concretely occur, referencing some of the inspiration from NEPENTHE, MIPP, and thermodynamic constraints in more depth.

Pruning and Memory Management Strategies

To keep the memory system efficient and consistent, Kimera SWM employs rigorous **pruning and memory management strategies**. These strategies are heavily inspired by recent advances in entropy-based model compression (like NEPENTHE and MIPP) and are constrained by thermodynamic considerations as discussed. We describe how the system decides what to trim, what to retain, and how it manages memory growth over time.

Entropy-Guided Pruning: The core principle is *entropy-guided selection* of what to prune. Rather than pruning memory arbitrarily or purely by recency, Kimera SWM looks at the information-theoretic value of each memory item. In practice, each scar (event) or group of scars gets an **entropy contribution score**.

Low entropy contribution means the event is either highly predictable from other events or doesn't add new distinctions. Such events are prime candidates for pruning. This aligns with NEPENTHE's strategy of removing low-entropy layers in neural networks to reduce depth [1] – by analogy, we remove or consolidate low-entropy segments of the memory graph to reduce its "depth" or size. For example, if the system has logged "sensor heartbeat OK" every minute for a year, those events are extremely predictable and redundant. The entropy monitor would flag this pattern as low information. The pruning module could then compress these heartbeats into a single summary (e.g. "sensor was OK for 99% of time in 2024") and delete the individual minute-by-minute records. The summary has far lower entropy than the collection of raw events, but it preserves the essential info (the sensor's general reliability record).

Mutual Information Preservation: A critical check during pruning is to preserve **semantic integrity** – we do not want to accidentally forget something that is a linchpin for understanding other events. This is where inspiration from MIPP comes in. MIPP prunes neurons only if they fail to transfer entropy to the next layer [2], meaning if a neuron's removal doesn't reduce the mutual information between layers, it's safe to remove. In our context, before pruning an event, we check its *connectivity* and *influence* in the memory graph. If that event is the sole witness to some important piece of information (for instance, only one event mentions a particular entity or a causal link), removing it would break a chain of understanding. That indicates it has a high mutual information with at least one other item (since it carries unique info that no other memory carries). Such an event should be kept. On the other hand, if an event's facts can be inferred from combination of other events, or if it is one of many similar events, it likely has low unique mutual information. We can prune it without losing overall knowledge, as other events collectively cover its information.

To systematically do this, the memory system can maintain an **influence map**: for each event, track references or links (which other events reference this event's actors, outcomes, etc.). If pruning a candidate event would remove an explanation for something in another event, that's a red flag. For example, suppose event A (prune-candidate) and event B (to keep) both describe related parts of a story. If A explains why B happened, then even if A is low entropy by itself, it's providing needed context to B – removing A would spike the entropy (uncertainty) about why B occurred. Therefore, A must be retained or at least partially retained (maybe compressed but not eliminated). In contrast, if event C is nearly identical to event D in context and outcome, C might be removed if D is kept (or vice versa) with minimal impact on overall information – this is like having redundant backups and deciding one copy suffices.

Temporal and Thermodynamic Constraints: Another aspect of memory management is the *temporal dimension*. Thermodynamic analogies remind us that time and irreversibility matter. Memory in an AI agent typically grows without bound unless managed; however, real physical systems (and brains) have finite capacity and tend to *gradually forget* or overwrite old information, especially if it's not used (second law of thermodynamics in a sense mandates that unused ordered information will decay unless energy is spent to maintain it). We implement a form of this by incorporating *decay or aging functions*. An event's entropy contribution might slowly decrease over time if it hasn't been accessed or proven useful (akin to how recall probability decays in human memory unless refreshed). This doesn't mean we arbitrarily delete old events, but we bias the pruning towards older, seldom-used information – unless that information was highly significant. This is comparable to a *cache eviction policy* or the AI memory analog of forgetting rarely accessed items to make room for new ones.

We also consider an energy budget: If the agent is running on an edge device with limited memory, we may set a hard cap on memory size (or an entropy cap). The thermodynamic manager ensures the system

doesn't exceed this capacity by pruning as needed. This can be viewed as maintaining a constant "entropy load" – if new events come in, some old low-value entropy must go out, similar to how an organism maintains homeostasis by shedding excess heat or waste. We might even tie this to Landauer's principle conceptually: each bit erased frees some energy (which might be negligible physically per bit, but conceptually it's an accounting of resources). So the system might track how many bits it has erased and how that relates to computational effort. In an engineering sense, this is just an accounting mechanism to ensure we don't overload memory; it encourages efficient encoding (so that the "energy" spent on memory is used for valuable info only).

Practical Pruning Algorithms: We can outline a concrete pruning algorithm in Kimera SWM: 1. **Calculate Scores:** Periodically (or when memory is at capacity), calculate an entropy score for each event or for each candidate group of events. Also compute a *reference count* or influence score (how many other events depend on this). 2. **Rank Candidates:** Create a list of events that are lowest in entropy contribution and low in influence (no critical links). Events that are old and never accessed in a long time get a further push up this list. 3. **Apply Threshold:** Determine a threshold such that removing candidates above it will free enough space or reduce entropy by a target amount. This threshold could be adaptive – e.g., prune until total memory entropy is under a desired value, or memory size under X MB. 4. **Preservation Check:** Before removal, double-check constraints. For each candidate, simulate removal and see if it would increase the entropy of any remaining memory item beyond acceptable levels (like if it was masking a contradiction or providing a needed clue). This simulation can be simplified by our influence map or by recalculating mutual information without that event. If removal is safe, proceed; if not, skip the event (it might be fundamentally required). 5. **Compress or Remove:** For each approved candidate, decide whether to compress or completely remove. Important but repetitive events might be replaced with a summary scar (so not full removal). Trivial redundant ones can be removed entirely. If removed, and if an archive is maintained, move it to long-term archive with a tombstone record. 6. **Re-index/Update:** After pruning, update indexes, and also update entropy fields of related events because removing data can slightly increase uncertainty in others (though we aimed to minimize that). The contradiction engine might also be notified to re-evaluate consistency if, say, an event that contradicted another was removed (perhaps resolving a contradiction by elimination).

This algorithm ensures *no external dependency* because all it needs are internal data and computations on them. It's effectively an **in-memory garbage collector for knowledge**, but guided by information content rather than just reference count.

NEPENTHE and MIPP Inspirations in Action: To explicitly draw parallels: - Our method of removing low-entropy events is just like NEPENTHE's removal of low-entropy layers [1]. In a DNN, those layers were "linearizable" (not adding complexity) [11], so they could go. In memory, if a set of events doesn't add complexity to what the agent knows, they can go. - Our mutual information check mirrors MIPP's focus on retaining information flow [8]. Just as MIPP ensures the pruned network can be retrained or still function by keeping info flow, we ensure the memory can still answer queries and maintain narrative coherence after pruning. - We also include what could be called "energy-aware pruning" – recognizing that pruning should also optimize computational energy (less memory = less data to search through, faster responses, maybe lower power for embedded devices). In practice, this means if two pruning options drop the same bits but one saves more CPU cycles (like removing many small events vs one big one), we might choose the path that yields computational efficiency. This aligns with *thermodynamic optimization* – akin to releasing heat in the easiest way possible.

Memory Management Beyond Pruning: Pruning is one side of the coin (removing old/unneeded info). The other side is *reinforcement of important memory*. Kimera SWM should also manage memory by strengthening what is frequently used or clearly important. While not explicitly asked for in the prompt, it's worth noting: if certain scars are repeatedly accessed or have high entropy (very informative), the system could allocate more resources to them (like keep them in a fast cache, or prevent them from ever being pruned, or create multiple indexes for them). This is reminiscent of how caching works or how the brain rehearses important memories to reinforce them. We incorporate this indirectly by the influence scores – high influence or high surprise events will naturally not be pruned and remain central. But an implementation might also include periodic re-indexing of popular memories for faster access.

Moreover, memory management includes **handling concept drift** or changes in context. If the agent's environment changes, the definition of what's redundant might too. For instance, a fact that was once low entropy might become high entropy if the world changes (e.g., "system stable" was redundant until a failure starts happening often, then those events become crucial to analyze). Our strategies remain adaptive: the entropy monitor continuously recalculates values, so the system can notice if a previously pruned detail suddenly becomes relevant (if it's archived, it can be pulled back). Strict realism means we anticipate such scenarios and ensure the design can handle them (perhaps via alerts that something unpredictable happened, prompting more data retention around it).

In summary, Kimera SWM's pruning and memory management ensure the working memory: - **Stays compact and efficient**, never bloating with useless or repetitive data. - **Preserves critical information** and context, so the system's effectiveness is not harmed. - **Operates within thermodynamic-like limits**, mimicking how physical systems manage limited resources. - **Adapts over time** as information value changes, thanks to continuous entropy assessment.

Next, we address how the system handles contradictions in memory and how the memory is accessed in an event-driven manner, which complements these pruning strategies to maintain a coherent and usable knowledge base.

Contradiction Engine and Memory Access Patterns

A robust semantic memory must not only store and prune information, but also ensure that the information remains logically consistent and accessible in useful ways. In Kimera SWM, the **Contradiction Engine** serves as the guardian of consistency, while the memory access patterns dictate how information is retrieved without relying on brittle token-based methods. This section details both.

Contradiction Detection and Resolution: Contradictions are essentially the cognitive equivalent of matter-antimatter collisions – two pieces of information that cannot both be true will annihilate the system's integrity if not handled. Our Contradiction Engine continuously scans the memory for such conflicts. It leverages the structured nature of EchoForm events: since events have defined fields like actors, state, outcomes, and time, it's easier to apply logical rules. For example: - If two scars describe the state of the same entity at overlapping times with different values, that's a direct contradiction (e.g., one event says a patient's status is "healthy" on Jan 1 and another says "critical" on Jan 1 with no further context – something is off unless they refer to different aspects or a time sequence). - If an event claims causal link that violates known physics or constraints (say Event A causes Event B, but Event B's timestamp is before Event A's – time paradox), that's a contradiction. - If the system has a rule set or ontology (not external, but internally

encoded basic constraints), it can catch violations (like an object can't be in two places at once, or a user cannot be older than their parent, etc.).

The engine uses a combination of methods: 1. **Rule-based checks:** A library of consistency rules (which could be domain-specific) is run. Each rule is a pattern to detect impossible or conflicting configurations among events. 2. **Graph consistency algorithms:** Viewing memory as a graph, certain contradictions show up as cycles or cliques that shouldn't exist. A simple example: if A->B (A implies B) and B->not A (B implies not A) are both present, that's a logical inconsistency cycle. The engine can use algorithms akin to SAT solvers or constraint satisfaction to identify unsatisfiable subsets of memory. 3. **Probabilistic inconsistency measure:** Even if hard rules aren't broken, the engine can note when two pieces of info strongly *disagree* statistically. For instance, one knowledge scar might say "component X usually fails when temperature > 80°C" and another might say "component X never fails above 80°C". They're phrased as tendencies, not absolutes, but they conflict. This raises entropy – an unresolved contradiction means the system has high uncertainty about truth. The contradiction engine flags these as well (though they might not be as clear cut as true/false conflicts, they are tensions that need resolution).

Once a contradiction is found, how does Kimera SWM resolve it? We want to avoid speculative or magical resolution; we rely on principled approaches: - **Truth Maintenance:** Similar to classical AI truth maintenance systems, we mark one or both pieces of information with their consistency status. For example, label one "RETRACTED" or "DISPUTED" if we have reason to disbelieve it. The system might have a confidence score for each event (based on source credibility or sensory input confidence). If a high-confidence and a low-confidence piece conflict, the resolution is to keep the high-confidence as truth and either drop the low-confidence one or mark it as an anomaly. If both are equally credible, the system can either: - Attempt to find a *contextual differentiation*: Perhaps they refer to different conditions (this is where the engine might augment the events with additional context to eliminate the direct contradiction – e.g., add a hidden assumption that resolves it, like "if sensor was calibrated differently in case B, then no contradiction"). - Or it can simply flag the conflict and route it to a higher-level decision (maybe a developer or a learning module to figure out). - **Recursive Contradiction Analysis:** The engine can simulate the impact of each assumption. This is where it acts like a *recursive contradiction engine* as mentioned with Aletheion [10]. It might temporarily assume one side of a contradiction and see if any further contradictions arise down the line (propagating that assumption through the event network). For example, assume event X is true and Y is false, mark Y as pruned/ignored, then check if everything else is consistent. If yes, that's one coherent worldview. Do the opposite (X false, Y true). If both worldviews can exist, then the system knows it has an ambiguity that it might keep separate (like two possible interpretations of events). If one leads to lots more contradictions, it picks the other as the likely truth. - **User or Sensor Queries:** In a realistic engineered system, if an autonomous resolution isn't clear, the contradiction engine can output a query – asking either the human operator or performing an active test in the environment to resolve the discrepancy. This is analogous to how scientists run experiments when theories conflict. For example, if memory says two different coordinates for the same object, the agent can physically check or ask for an update from sensors, then update memory accordingly.

Crucially, the contradiction engine ensures the memory remains *usable*. It's acceptable to have some disputed knowledge in memory (real humans do too) as long as the system knows it's disputed and doesn't unwittingly use two contradictory pieces together in reasoning. By labeling and segregating contradictory scars, the system can avoid combining them in a query response or plan. In effect, the contradiction engine upholds a **consistency contract**: any answer drawn from memory should come from a self-consistent subset of the knowledge base.

Memory Access Patterns – Event-Driven and Non-Tokenized: Traditional memory retrieval (like in databases or text corpora) often relies on tokens or keywords. For example, a chatbot might do a keyword search over past conversation logs. Kimera SWM deliberately avoids purely token-based access because token matches can fail when wording changes or when context is needed to know what to recall. Instead, our retrieval is **event-driven**: - **Contextual Cue Matching:** When a new input or situation arises, the system forms a partial EchoForm representation of it (just like for storing events). This partial event acts as a query key. For example, if the agent is currently at location L and interacting with person P about topic T, the query might be "<Actor=P or OtherRole=P, Location=L, Topic=T, >" – find past events that involve this person, place, or topic. Because we can query by structured fields, we aren't limited to exact word matches; even if the wording differs (e.g., "John went to Paris" vs "Trip by John to France"), as long as the entity "John" and location "France" are properly represented in the structure, the query will catch it. - **Semantic Similarity Search:** For more abstract cues, we might generate an embedding (vector) of the query context and do a nearest-neighbor search among embeddings of stored events. This would handle cases where the scenario is similar but not identical. For example, the system might infer that "delivery of a package" is similar to "pickup of an item" in structure (both are logistics events), so if one happened before, it might recall it even if keywords differ. The key is that this similarity is computed on semantic structure, not raw text. Internally, it might use descriptions, but those descriptions come from the structured data, possibly concatenated or encoded in a model. - **Subscription/Trigger Mechanism:** The memory system can also support a publish-subscribe model. Certain events can register interest in future events of a type. For instance, if an event was very important (like "Mission failed due to battery outage"), we might want any event with "battery" in the future to automatically recall that failure event as a warning. This is implemented by setting a trigger on the keyword or concept "battery outage" – effectively an event pattern that when matched in a new event's encoding, causes an immediate recall of the linked memory. This is beyond static queries; it's dynamic, event-driven recall. It's akin to how hearing a familiar phrase might immediately remind a person of a past incident without conscious search.

Non-Tokenized Advantage: By avoiding tokenization as the primary key, Kimera SWM can retrieve memories even when the exact vocabulary or phrasing differs. It focuses on *concepts and relationships*: - If the user said "I'm feeling chilly" and previously they said "I am cold," a token match might miss it (different words), but an EchoForm representation might categorize both under a feeling: temperature sensation. The memory query for similar user states would retrieve the past event. - Non-tokenized also implies we don't rely solely on something like a giant embedding vector for the entire memory. Those approaches (common in LLM context) might treat memory as a black box of text. Instead, our approach allows pinpointing precise events relevant to specific facets of the query.

However, we can incorporate tokens as needed within fields. For example, if an event has a text transcript, that might still be searched with keywords, but only after narrowing down by context (like find all events with speaker=Alice, then search their transcripts for "project"). This multi-step approach is more efficient than searching everything blindly.

Efficiency Considerations: Event-driven retrieval is designed to be computationally efficient given structured indexes. A naive approach would be to scan all events for ones matching criteria. With indexing, we can do much better: - Maintain a map for each field value to the list of event IDs containing that value. For example, an index from entity -> list of events involving that entity. This makes single-field lookups O(1) for the map plus O(k) for k results. - For multi-criteria queries, intersect those lists (which is typically fast if lists are sorted or if one is much smaller). - Use time constraints: if looking for recent relevant events, the events could be stored in time-order and we break out of search after going back a certain period unless

needed. - Use caching: if a certain query repeats (like the agent repeatedly asking “when did I last charge battery?”), caching that answer or the relevant event speeds up subsequent recall.

Integration with Contradiction Engine: The contradiction engine’s output also influences access. If an event is marked disputed, the access module might by default not surface it unless specifically asked, or it might annotate it as “(controversial)” in responses. This ensures that what is retrieved and used by the agent doesn’t blindly include bad data. Conversely, if two conflicting memories exist, a query might retrieve both but the system will present or use them in a careful way (perhaps triggering a contradiction resolution process before acting on them).

Real-World Usage Example: Suppose the agent is a personal assistant AI with Kimera SWM, and the user asks: “Do I have any meetings related to Project X soon?” The query is parsed into semantic form: (user=Self, action=meeting, topic=Project X, time=future?). The memory access will find all events that are meetings about Project X. Because events are stored with topics and times, it fetches those events (maybe ones where a meeting was scheduled). It finds one such event, a scheduled meeting next week. It also checks if any contradictions: maybe two meetings double-booked? If yes, the contradiction engine flags it, and the system might respond with an alert. If not, it simply answers based on the memory. Notice no single token “meeting” needed to match, it could work even if the event was recorded as “Call with John regarding Project X” – because the structured topic is Project X and type is meeting/call.

By designing memory access in this semantic-first way, we ensure **flexibility** (robust to wording), **precision** (targeted by context), and **recall** (able to find relevant knowledge widely). Importantly, it’s all “implementation-ready”: building an index on structured data and writing query handlers is straightforward engineering, and avoids reliance on large external search services.

To tie this back to engineering realism: These patterns can be implemented with standard data structures (hash tables, inverted indexes, etc.). They do not require exotic hardware or theoretical oracles. They are analogous to how a well-designed database or knowledge base operates, but tailored to the semantic memory use-case. Many cognitive architectures and databases already support similar queries; we are simply ensuring to incorporate them in a way that leverages our event structure and entropy annotations.

Having covered storage, pruning, consistency, and retrieval, we now move to practical aspects of implementing Kimera SWM, discussing data structures, code considerations and computational overhead to demonstrate feasibility.

Implementation Guidelines

Designing the Kimera SWM concept is only half the battle; implementing it in code with efficiency and clarity is the other half. In this section, we provide guidelines for how one might implement the system, covering data structures, code-level patterns, and computational complexity. The goal is to show that each part of the design can be realized with today’s engineering tools and to flag potential performance issues and how to mitigate them.

Event Representation Data Structure: At the code level, each **scar (event)** can be represented as a structured object or record. In an object-oriented language, this could be a class `Event` with fields corresponding to the EchoForm grammar: e.g., `actors`, `action`, `objects`, `timestamp`, `location`,

`outcome`, `causes`, etc. Alternatively, a simple approach is to use a dictionary (hash map) or a struct. The key requirement is that this data structure is serializable (for saving to disk) and easy to compare/search (so accessing a field is O(1)). For instance, in Python it might be a dictionary, in C++ a struct with some enum types for actions perhaps. An example in pseudo-code:

```
event = {
    "id": 1001,
    "actors": ["robot1", "user42"],
    "action": "deliver",
    "object": "package123",
    "location": "Warehouse_A",
    "time": "2025-06-05T14:30:00Z",
    "outcome": "success",
    "cause": 1000  # reference to prior event ID that triggered this
}
```

This shows an event with an ID (for reference and linking), with fields filled. Storing an event like this in memory is straightforward – it's just a few bytes of data per field plus overhead. A million such events might be large but not infeasible on a modern system (and we plan to prune far before that in edge cases).

Memory Store Implementation: The memory store can be in-memory for speed. A baseline implementation could simply be a list or array of events. However, as mentioned, we'll want indexes for efficient retrieval. We can maintain dictionaries mapping from field values to sets of event IDs. For example: - `index_by_actor: { "robot1": [1001, 1007, ...], "user42": [1001, 1010, ...], ... }` - `index_by_action: { "deliver": [1001, 1005, ...], "move": [1002,...] }` - and similarly for location, object, etc.

These indexes need to be updated whenever an event is added or removed. This is O(1) insertion into a few hash maps, which is fine. Removal is also O(1) per index if we keep pointers. The memory footprint of indexes is something to consider (they essentially duplicate pointers to events), but for manageable memory sizes this is fine.

For linking causes and effects, we might maintain a graph adjacency list as well: - `causal_links: { event_id: [list_of_consequence_event_ids] }` This helps the contradiction engine and also influence tracking (which events depend on which).

Entropy Calculation: We'll need functions to calculate entropy for events or sets. The entropy monitor can maintain running counts for distributions to make this efficient. For instance, to know entropy of the "action" field, it keeps a count of how many events of each action type exist. From these counts it can compute entropy H = -Σ (p_i log2 p_i) easily whenever needed. Updating these counts on event add/remove is O(1). Similarly for other fields or combinations. If we want entropy of an individual event being added, we can compute its surprisal using a predictive model or distribution from past frequency (like if "deliver" happened 500 times out of 1000, probability ~0.5, surprisal ~1 bit for action field). Summing across fields or using joint distribution would refine this, but as a first pass, field-wise entropy is a proxy.

Entropy Tension and Clustering: Implementing the entropy tension field concept could use clustering algorithms (like k-means or hierarchical clustering on events by feature) to find clusters of similar events and measure their entropy. This can be computationally heavy if done frequently. A heuristic approach is to maintain cluster centroids that update incrementally: e.g., maintain a few "contexts" and assign events to them, updating running entropy. Alternatively, since our memory sizes are intended to be trimmed, we might be able to afford clustering occasionally on a batch of events.

However, a simpler method: maintain a summary of entropy by category (like by actor, by time period, etc., as mentioned). Differences between those summaries indicate tension. This is easier to compute: - Entropy by actor: some actors have many varied events (high H), others repetitive (low H). - Entropy by time: last week vs this week. If a difference is big, we trigger an analysis or smoothing action. So code-wise, loops over categories and compute entropies periodically (maybe each time N new events added or every T seconds).

Pruning Implementation: The pruning algorithm outlined can be realized with a priority queue or sorted list of events by their "prune score" (a combination of low entropy and low influence). We could maintain this in real-time: whenever an event is added, compute its score and insert into a min-heap. Influence changes are trickier – if an event gains influence (because a new event linked to it), we should update its score (meaning it's less removable now). This suggests that pruning cycles might be easier to manage: e.g., recompute scores on the fly during a pruning cycle rather than constantly updating. Given moderate memory sizes (maybe thousands, not billions, of events after pruning), recomputation is fine.

When the system decides to prune: - Identify candidates (heap gives the lowest score events quickly). - For each, do the checks (these checks might involve scanning some links or recomputing MI if needed). - Remove and update indexes accordingly.

We must carefully handle data structures while pruning: remove event from list, from each index map (which is easy by ID, just pop it from the lists), and from any link references (remove from causal_links references). If we keep a persistent archive, write the event out to disk (e.g., append to a log file) before removing.

Contradiction Engine Implementation: This can be approached with logic programming or constraint solvers, but a simpler approach: - Whenever an event is added or updated, run a targeted check against related events. For example, if event says `Device X = ON at t`, check if we have any event with `Device X = OFF at t`. That's a direct conflict. Because of indexing, we can find all events about `Device X` and then filter by overlapping time. This is much faster than checking everything against everything. - For more global consistency, certain rules might require scanning (like check if any cycles in causality graph). That could be done periodically (maybe after major updates or daily). - Use a status field in event data structure like `event["status"] = normal/conflict/resolved`. The engine can set this. The memory access can then filter or treat accordingly.

We could leverage existing libraries for logical consistency if needed (like an SMT solver), but in keeping with no external heavy dependencies, simple self-coded checks or leveraging something like Python's `z3` if already in environment could be optional. But it's doable manually for most straightforward contradictions.

Memory Access Implementation: The access interface can be implemented as a set of query functions or methods. For example:

```
def query(events, criteria):
    # criteria could be a dict of field->value that must match (or list of
possible)
    results = initial_set = None
    for field, value in criteria.items():
        if field in index:  # use index if available
            ids = index[field].get(value, [])
            ids = set(ids)
        else:
            ids = {e["id"] for e in events if e[field] == value}
        if results is None:
            results = ids
        else:
            results &= ids  # intersection
        if not results:
            return []  # no match early break
    return [memory_store[id] for id in results]
```

This simplistic approach handles exact matches. For similarity or range queries (like time within range, or numeric tolerance), we'd need to adapt (maybe maintain sorted lists for time for binary search, etc.).

For embedding-based similarity, one could integrate an ANN (approx nearest neighbor) library, but that might be considered an "external framework". However, one can also code a basic vector search if scale is small, or use a library like FAISS if allowed. But the key is we only embed the semantic content of events, not rely on raw text.

Computational Overheads: - **Memory growth:** If unchecked, storing events will eventually fill memory. We handle this via pruning triggers (perhaps when memory usage passes a threshold). So in steady state, memory size oscillates around some desired maximum. This makes operations mostly bounded. - **Insertion cost:** Adding an event requires updating indexes (if we have, say, 5 indexes, that's 5 insertions) and maybe updating some entropy counts (a few increments) and running a quick contradiction check (usually O(k) where k is small subset). This is all quite fast, likely on the order of milliseconds per event or less, which is fine for many realtime applications. - **Pruning cost:** Pruning is more expensive since it may scan many events to choose victims. But if done periodically, it's manageable. Also, because we aim to prune a lot at once and then not do it again until needed, the amortized cost is okay. For example, prune 100 events in one go after adding 1000 events gradually – the cost of scanning 1000 events with updated stats is fine. - **Query cost:** Querying by structured index is very fast if the criteria are selective. Worst case (open-ended query) might require scanning many events, but typically queries are about something specific (an entity or an action + timeframe). We can optimize by short-circuiting as shown. In pathological cases, we might add a full-text search fallback if needed (if someone queries a random text not indexed, but that's more for text memory; for structured events, nearly everything should be indexed). - **Contradiction check cost:** A full consistency check of entire memory could be exponential in theory (like NP-hard SAT). We avoid that by incremental localized checks. In practice, contradictions will likely be caught at insertion time if related events are present. The complexity then is manageable as discussed. Also many contradictions are pairwise, which is easier than arbitrary.

Parallelism and Concurrency: In a multi-threaded or distributed scenario, care must be taken with memory modifications. We likely treat the memory store as a critical section (or use concurrent data structures). For realism, we should mention that if the AI agent has multiple processes (or threads) reading/writing memory, we need locking or atomic operations to avoid race conditions (e.g., two threads pruning or adding simultaneously). A simple approach is to funnel all memory writes through a single thread or queue (serializing modifications), while reads can be mostly concurrent if using immutable snapshots or locks. This is an engineering choice depending on required performance. Given edge-AI context, often a single agent might not need heavy multithreading for memory, but it's a consideration.

Language/Platform Considerations: The implementation can be language-agnostic, but each has trade-offs: - In Python, ease of development is high, and it's suitable if memory sizes are not huge (or if using something like PyPy or C extensions for heavy parts). - In C++ or Rust, one can optimize memory usage better and possibly handle larger scale, at cost of development complexity. - For an embedded device (edge AI), C/C++ might be necessary, or even storing memory in a small SQLite database might be a pragmatic choice (though that introduces an external component, but maybe acceptable as it's just an embedded library). - If persistence is needed across reboots, some lightweight file storage (JSON/CSV logs, or a binary dump of the structures) can be implemented. Realistically, the system would have a routine to snapshot memory to disk and load it on startup.

Code-Level Example: As an illustrative snippet, consider how to handle adding a new event and triggering possible recall:

```
def add_event(event):
    memory_store[event["id"]] = event
    for field, index in indices.items():
        val = event.get(field)
        if val is None: continue
        index.setdefault(val, []).append(event["id"])
    update_entropy_stats(event, addition=True)
    # Check contradictions with related events (e.g., same actors or objects)
    related_ids = set()
    for key in ["actors","object","location"]:
        if event.get(key):
            related_ids |= set(indices[key].get(event[key], []))
    for rid in related_ids:
        if rid == event["id"]: continue
        check_contradiction(event, memory_store[rid])
    # Trigger recall if needed
    recall_candidates = retrieve_related_events(event)
    for past_event in recall_candidates:
        # perhaps merge past context with current or alert another module
        handle_recalled_memory(past_event, event)
```

This pseudo-code shows the integration: add to store, update indices and entropy, do a localized contradiction check (by gathering related events via indices), then retrieve related events for context (using a function that queries by overlapping fields or similarity). Each of those sub-calls

(`update_entropy_stats`, `check_contradiction`, `retrieve_related_events`) would be implemented per guidelines above.

We see that none of this is beyond standard programming – it's mostly bookkeeping and some logic. The complexity lies in designing good criteria and thresholds, but not in the actual coding.

In terms of computational load, if we assume memory holds, say, 10,000 events after pruning, an index lookup touches maybe tens or hundreds of entries, a contradiction check iterates maybe those, and an entropy stat update is constant. So real-time additions (even dozens per second) are feasible on a modest CPU. Pruning might freeze things for a short time if done synchronously; one can mitigate by doing pruning in a background thread or during idle periods (like overnight maintenance).

One must also consider *edge cases*: - If memory corruption or inconsistency happens (like indexes out of sync due to a bug), we need consistency checks or rebuild routines. - If contradictory info continuously flaps (one sensor says on, another says off repeatedly), the system might oscillate. That's more of a logic issue, but we should design smoothing (maybe trust one source or average out). - If an event doesn't fit the grammar (maybe some unexpected new type of event), the system should still store it in a catch-all format (so extensibility in the schema is needed to avoid losing data just because it's unfamiliar).

To sum up implementation: - Use **structured objects** for events. - Maintain **indexes** for fast lookups on important fields. - Continuously maintain **entropy metrics** as events are added/removed. - Use straightforward loops or modest algorithms for contradiction checking and clustering – given our controlled memory size, these will be efficient. - Be mindful of concurrency and persistence as practical aspects. - Test each module independently (e.g., create a fake scenario and see if contradiction engine flags the expected issues, if pruning removes what it should, etc.) to ensure reliability.

By following these guidelines, an engineer could build a prototype of Kimera SWM and iterate on it. The key takeaway is that while the conceptual architecture is novel, its implementation relies on well-understood engineering methods (no mysterious black boxes needed). Next, we will consider possible blind spots and opportunities in this design – acknowledging what might be missing or oversimplified, and how future iterations could address those.

Blindspot Evaluation and Engineering Opportunities

No design is without limitations. In this section, we critically evaluate Kimera SWM to identify blind spots – areas the current design may not fully address or potential pitfalls if deployed. We also outline opportunities for engineering improvements, such as incorporating entropy-aware hierarchies and other enhancements to strengthen the system.

Identified Blind Spots and Naïve Assumptions:

- **Quality of Semantic Encoding:** The EchoForm grammar is central to our design. If this grammar is poorly designed or too rigid, it could become a bottleneck. A blind spot is assuming we can pre-define a schema that cleanly fits all important events. In reality, unexpected events might not slot neatly into the predefined fields, or some information might not be captured. For example, emotional tone or nuanced context might be lost if our grammar only captures concrete facts. The

current design doesn't explicitly handle things like uncertain information or multi-faceted events well (it assumes fields are filled with fairly crisp values). An engineer should plan for schema evolution – the grammar might need extensions (like an "additional_notes" field or ability to link to unstructured detail) to avoid losing data that doesn't fit. Without this, the system might silently drop or ignore aspects of experience, which is a blind spot.

- **Knowledge Integration and Inference:** Kimera SWM stores events and finds links, but it doesn't explicitly include a general inference engine beyond contradiction checking. A potential gap is how new implicit knowledge is derived. For instance, if event A and B together imply C, do we store C or infer it on the fly? Currently, we lean on explicit stored links and human-coded logic for contradictions. There is an opportunity to incorporate an *inference module* or rule engine that can derive new scars from old ones (like concluding "maintenance is needed" if many failure events happen). Without some inference, the system is purely reactive. However, adding inference also adds complexity and risk of speculative data. Our realism constraint may have made us shy away from heavy inference, but perhaps a limited form (like a rule to aggregate patterns) could improve the system's utility.

- **Memory Capacity Limits:** We have assumed we can keep memory pruned to a manageable size. A blind spot might be scenarios where information influx is so high and varied that the system either has to drop potentially important things or performance degrades. For instance, an edge device might experience a spike of events (like a sensor flood during a failure) – our strategy would prune aggressively after, but during the spike memory and CPU could overload. The design doesn't yet incorporate *back-pressure* or flow control mechanisms (like temporarily slowing event intake or compressing on the fly when overwhelmed). In engineering terms, we might need to integrate load-shedding strategies so that if events come in faster than can be processed, some are queued or summarized in real-time to avoid collapse.

- **Context and Task Blindness:** The current design treats all events somewhat uniformly in terms of entropy and retention. It doesn't explicitly account for *task relevance* or *goals*. In practice, what's important to remember can depend on what the AI's current objectives are. If the system's priorities shift, our memory management should possibly shift as well (e.g., things relevant to the current mission should be kept even if statistically low entropy). If we naively prune purely by entropy math, we might drop something critical for a rarely occurring but crucial scenario. One opportunity is to incorporate a weighting mechanism for entropy that takes into account relevance to active goals or user-defined importance. For example, tag certain events as "pinned" (never prune) because an engineer or the AI's higher-level logic marked them as mission-critical knowledge, regardless of entropy. Our design didn't explicitly mention that, but it's a straightforward addition.

- **Contradiction Resolution Limitations:** While we have a plan for contradictions, subtle inconsistencies might slip through. The contradiction engine might catch direct conflicts but could miss more complex ones that require reasoning (e.g., a sequence of events that is collectively impossible but no two directly conflict). Also, in dynamic environments, something can become contradictory only in hindsight. Our engine as described doesn't proactively re-evaluate old memories unless triggered. There is an opportunity to schedule periodic deep consistency audits, or to integrate more advanced consistency checking (like using temporal logic frameworks). But those come at computational cost. We have to strike a balance, and the blind spot here is that *ensuring*

consistency in a large knowledge base is notoriously hard (NP-hard in general cases) – our solution is heuristic and might not guarantee a contradiction-free memory, just a managed one.

- **Human Input and Explainability:** An often overlooked aspect is how to debug or intervene in the memory system. If an engineer wanted to know "why did the system forget X?" or "what caused this contradiction flag?", can the system explain its reasoning? Right now, the design includes data to possibly trace that (like event scores, timestamps of deletion, etc.), but we haven't explicitly stated an *explainability interface*. This is an opportunity: providing logs or rationales (e.g., event X pruned on date Y due to low entropy and redundancy) would greatly help trust and debugging. Without it, the system might appear as a black box that forgets things unpredictably, which is not ideal for engineers or users.

Entropy-Aware Hierarchies (Opportunity): One promising improvement is to introduce a **hierarchical memory structure** that is aware of entropy at each layer: - We could divide memory into tiers: e.g., *Short-Term Memory (STM)* for very recent raw events (high detail, high entropy but ephemeral), *Intermediate Memory* for consolidated events over the last day/week, and *Long-Term Semantic Memory* for distilled knowledge (low entropy per unit, highly stable). Each layer operates at different "temperatures" in thermodynamic analogy. The STM is "hot" (lots of new info, high entropy influx) but we allow it to evaporate quickly (only the most important transitions move to the next layer). The long-term memory is "cold" and stable. - Using such hierarchy, we can apply different pruning strategies: STM prunes aggressively based on recency (like a sliding window), intermediate prunes based on entropy as we described, and long-term prunes rarely (maybe only if logically necessary or storage absolutely full). - This also gives a natural way to incorporate *spaced repetition or re-encoding*: as events trickle from short-term to long-term, the act of consolidation can reinforce important patterns (like counting how often something happened). - Entropy awareness means at each boundary, we use entropy to decide what passes upward. For example, only events that reduce uncertainty about the agent's model of the world are kept in long-term.

Implementing this hierarchy would require some additional design (like triggers to move events between layers), but it's an extension rather than a fundamental change. The opportunity is that it could improve scalability and biological plausibility. The blind spot of not having it is that our current single-layer memory might either behave too much like short-term (if we prune a lot) or too much like long-term (if we prune too little), and might not capture the advantages of multi-timescale processing. Real brains have sensory memory, working memory, long-term memory – mimicking that could yield better performance.

Learning from Data (Opportunity): Another improvement area is making some aspects trainable. For instance, the thresholds for pruning or the weighting of entropy vs. recency could be learned from experience (meta-learning) rather than fixed. If the system notices it often pruned something it later needed, it could adjust thresholds. This moves from a hand-engineered system to a more adaptive one. It must be done carefully to remain realistic (maybe using statistical feedback loops). Currently, parameters like "entropy threshold for pruning" or "time to live for unused events" are assumed to be set by engineers. A smarter system could tweak them, which is an opportunity for more autonomy.

Multimodal and Complexity Blind Spot: At present, we focused on semantic data possibly text-like or categorical. A blind spot is how we handle rich data like images or audio as part of events. We mention future multimodal scaling in the next section, but it's worth noting here: if an event includes an image or a complex data blob, our entropy measures and comparisons become more complicated. We might need special handling (like use an image hashing or feature extraction to compare images, etc.). Without that, the

system might either treat each image as opaque (thus possibly high entropy always) or ignore them. This is a known complexity but also a big opportunity, since many real-world agents will have non-text inputs. The entropic associative memory research suggests storing images and multimodal data is feasible with an entropy model [15], so we can take cues from there in future expansions.

Robustness to Faulty Data: Another blind spot is how errors in input data affect the system. If sensors give a burst of nonsense (e.g., due to a malfunction), the memory might store some garbage events. These could look high entropy and the system might erroneously preserve them, or they could cause contradictions. We might need a sanity-check layer on inputs (like filtering extremely improbable events or marking them as suspect). The design didn't explicitly mention this kind of validation. Realistically, an engineering solution often includes input validation to avoid polluting the knowledge base. This could be integrated as a pre-processing step (e.g., don't record an event that violates basic physical constraints unless multiple sensors agree, etc.). Without it, the system's memory could be thrown off by a single glitch (like a sensor reading "temperature 1000°C" once).

User Control and Configurability: From an engineering perspective, we want to allow configuration of the memory system depending on use-case. Some might want a very forgetful system (for privacy or compliance), others want maximum retention. Our design is flexible enough to tune (by adjusting pruning thresholds, etc.) but we haven't explicitly enumerated those knobs. It's more of a documentation issue, but an opportunity is to clearly expose parameters such as: - Max memory size or age, - Entropy threshold, - List of rules for contradiction (which could be customizable per deployment), - Fields included in EchoForm (extensible schema), - etc. Documenting and providing these hooks increases the system's practical adaptability. Without them, one might have to dive into code to change behavior, which is less ideal.

In conclusion, while Kimera SWM provides a strong framework, these blind spots and considerations remind us that: - We should be prepared to iterate on the semantic schema and incorporate flexibility. - We should consider multi-layer memory for better performance. - We need to guard against over-pruning or mis-prioritizing due to blind application of entropy metrics. - Additional modules (like inference or learning components) could be integrated for a more complete cognitive system, if aligned with testability. - Practical issues like debugging, error-handling, and configurability are as important as the core logic.

Addressing these in future versions can turn potential weaknesses into strengths. Many of these points lead naturally into future development scenarios, which we will explore next, particularly focusing on expanding the system's scope and adapting it to other contexts like multimodal data and edge devices.

Future Development Scenarios

Looking ahead, the Kimera SWM system can be extended and applied in numerous ways. In this section, we outline several future development scenarios that build on the current design while staying within the realm of engineering feasibility. These include handling multiple data modalities with entropy scaling, enabling more autonomous semantic adaptation of the system, and deploying the system in resource-constrained edge AI environments.

Multimodal Entropy Scaling: Thus far, we have primarily discussed semantic memory in terms of textual or symbolic information (events described by words or categorical data). A clear next step is to incorporate **multimodal data** – images, audio, sensor signals – into the working memory. Each event (scar) could include not just symbolic fields but also references to raw data (e.g., an image from a camera at the time of

the event, an audio clip of a conversation, or a time-series snippet from a sensor). Managing these within Kimera SWM requires extending our entropy modeling to such data: - For images and audio, we can use their information content as measured by algorithms (like the entropy of an image's pixel distribution, or more practically, the file size after compression as a proxy for information content). If an image is very similar to a previous image (low information gain), the system might not need to store it fully – it could store a reference or not at all. Conversely, a novel scene would have high entropy and should be stored or at least have features stored. - We might incorporate **feature extraction**: for example, use a vision model to extract semantic features from an image (objects detected, their relationships). Those features can be incorporated into the EchoForm description (e.g., event says "Camera saw Object=Car at Location X" plus maybe a hash of the image for uniqueness). The raw image can be archived offline unless needed. This aligns with our approach of semantic compression: we keep the meaning, not necessarily the raw pixels, unless the raw data is crucial. - The **entropy tension fields** concept could be applied across modalities. Imagine a scenario with a robot that has camera input and textual instructions. If the textual memory expects a certain state but the images consistently show something else, that discrepancy is an entropy tension between vision and text modalities. The system could detect this and flag, for example, "the environment looks different than described instructions, uncertainty is high". It might then prioritize storing more images or rechecking the textual data for errors. - As a reference, entropic associative memory research indicates that a single memory system can store multiple modalities and retrieve them associatively [15]. We can foresee Kimera SWM doing something similar: an event might be recalled by an image cue (e.g., seeing a location triggers recall of text info about that location, because the location was an indexed field). The memory system needs to handle embedding and linking across modalities but the architecture is fundamentally capable of that since events can have pointers to any data. It's more about implementing the similarity search (like using image embeddings to find matching images in memory, etc.).

Autonomous Semantic Adaptation: By this, we mean the system's ability to evolve its own knowledge representation and strategies without human reprogramming, guided by semantic entropy considerations. Some possibilities: - **Adaptive Schema Evolution:** Over time, the system might discover new recurring patterns or categories that weren't part of the original EchoForm fields. For instance, if it repeatedly encounters events that involve a certain concept not originally isolated, it could add a field or classification for it. This could be done by noticing high entropy in some free-text description portion of events and deciding to structure it. While implementing self-modifying schema is complex, a simpler approach is to have some flexible slots or tagging system that the AI can populate as it sees fit. Essentially, the system *learns new semantic features* that reduce entropy. For example, maybe initially events just had a generic "description" text, but the system notices many descriptions mention "error code X". It could formalize "error_code" as a field, which reduces entropy (because now it's explicitly tracked and can be compressed if repeating). - **Self-tuning Parameters:** The system could use a feedback loop to adjust its memory retention policies. As noted in blindspots, it might increase retention if it found that it pruned too aggressively (like if it frequently had to fetch from archive or experienced knowledge gaps). It can treat its own performance (in answering queries or making decisions) as data: if performance drops because needed info was pruned (detected e.g. by a user correcting it or a failure that could've been prevented with memory), then it adapts by pruning less or altering thresholds. Over time, the system "learns" the optimal entropy level to maintain for its environment. This is analogous to an autonomous thermostat adjusting to keep a stable temperature; here it keeps optimal knowledge volume. - **Online Learning of Contradiction Resolution:** The contradiction engine could improve by learning from past contradictions how to handle new ones. For example, suppose initially on contradictions it always deferred to the most recent info. If it observes that sometimes recent info was wrong (perhaps later corrected), it might adapt the rule (like incorporate a credibility measure or consistency-over-time measure). This could be achieved with a small reinforcement

learning loop or just heuristic tweaking. The key is that the engine doesn't need to be static; it can be made to evaluate outcomes (did resolving contradiction this way avoid future issues?) and adjust. - **Emergent Semantic Relationships:** As the memory grows, new connections between concepts may emerge. The system might autonomously form what in knowledge representation would be called *ontologies or semantic networks*. For instance, it might notice that every time event type = "delivery" occurs, there's also an event "feedback received" after – forming a causal rule. It could then add that rule to its knowledge (somewhere between memory and inference). This is stepping into the territory of knowledge discovery. While not originally in scope, a semantic working memory with an entropy focus might naturally highlight such patterns (because if two events are always paired, treating them as one higher-level event reduces entropy significantly). The future system could autonomously compress those into a single composite event type or a rule that binds them. This is an opportunity for building more abstract knowledge out of raw episodic data.

All these adaptations should be done within "strict engineering realism," meaning we'd implement them as incremental improvements or use existing machine learning techniques, not magic. For example, using clustering or pattern mining algorithms on the event log to propose schema changes is feasible. Testing different pruning thresholds and measuring query success is feasible.

Edge-AI Deployment: One of the envisioned scenarios is deploying Kimera SWM on **edge devices** – devices with limited compute, memory, and no constant cloud connectivity (e.g., a home robot, an IoT hub, a smart vehicle). This imposes certain constraints and opens opportunities: - **Lightweight Footprint:** We must ensure our implementation is lightweight in terms of memory overhead (both in memory stored and code). That's why the design eschews heavy frameworks. We might implement it in C++ for efficiency. Also, an edge device might only have storage for, say, a few thousand events, meaning the entropy-guided pruning is not just a luxury but a necessity. We might integrate with flash storage for archival, as edge devices often can dump old data to an SD card or similar. - **Intermittent Processing:** Edge devices often can't do heavy processing continuously (battery life, etc.). Kimera SWM might therefore operate in bursts – e.g., do pruning or entropy recomputation during idle times or when connected to power. This requires scheduling and perhaps a notion of degrade mode: if the device is busy doing real-time tasks, the memory management might defer operations (which is okay for a while but not indefinitely). Building in those considerations (like thread priorities or external triggers such as "power plugged in, run maintenance") would be practical. - **Privacy and On-Device Learning:** Using SWM on edge means data (events about a user or environment) can be retained and processed locally without uploading to cloud, preserving privacy. This is a big selling point. Our approach is well-suited since it doesn't assume cloud connectivity at all. A future scenario could be personal assistants that keep a long-term memory of user interactions on-device. The SWM could ensure it remembers important preferences (high entropy events like unique user feedback) while trimming routine interactions. If the device runs low on space, it prunes using our strategies, ideally not losing anything critical. Because it's entropy-driven, one could argue it prunes in a way that *minimizes loss of useful info*, which is what you want on an edge device with limited capacity. - **Integration with Edge AI Models:** Many edge scenarios involve smaller neural networks or rule-based systems doing tasks. Kimera SWM could serve as a knowledge cache for those. For example, a small speech recognizer runs on device producing transcripts (events), and SWM stores what's said and relevant context. Later, when the device needs to respond or make a decision, it queries SWM for context (like "did the user mention this appliance recently?"). This can make even a small model seem more intelligent because it has memory. Future development might involve providing a simple API for local AI modules to store and retrieve events from SWM. The opportunity here is to become a sort of *middleware for memory* in edge AI deployments. - **Resilience:** Edge devices might face power loss or reboots, so SWM should be resilient – regularly

checkpoint memory to flash, and recover quickly. This is engineering detail, but future improvements could include robust serialization and perhaps redundancy (like store crucial memory in non-volatile storage immediately). A scenario is a robot that crashes but after reboot it loads its SWM state and recalls what it was doing; that continuity is crucial for real-world use.

Collaborative and Distributed Memory: Another forward-looking scenario is multiple agents or devices sharing aspects of their memory. While we insisted on no external dependencies, one could imagine multiple Kimera SWM instances syncing certain scars (like via a network) to create a collective memory. For instance, a swarm of edge devices each with SWM might periodically merge high-level summaries of events to learn from each other. This introduces complexities (consistency across devices, privacy), but conceptually fits: you'd exchange only high-level, low-entropy summaries to avoid bandwidth overload, which is precisely our idea of compression. Implementing a distributed SWM might use entropy as a filter for what information to share (only share unexpected or important events with peers). This could be explored in the future to enable group learning without central servers.

Use in Edge-AI Systems (Summary of Advantages): To connect to source content: entropy-based pruning and compression have been highlighted as key for deploying models in resource-constrained environments [16]. Kimera SWM embodies that philosophy: by keeping memory trim and relevant, it allows an AI to run long-term on a device without exhausting memory. It also means less energy use (since processing fewer irrelevant bits) which suits battery-powered devices. So the future scenario is quite plausible where SWM is a standard component in on-board AI systems for drones, appliances, or personal devices that require local decision-making with memory.

Beyond 2025 – Speculation Avoidance: We consciously avoid sci-fi leaps (like "the system will become self-aware"). All described scenarios are extensions of current trends: multimodal integration, self-optimization, edge deployment, distributed systems – all under active research already. The difference is that we'd apply them within our semantic-thermodynamic framework. For example, none of these require new physics or unproven algorithms – they'd rely on integrating known techniques (like clustering for schema learning, small ML models for adaptation, or network protocols for sharing summaries).

In conclusion, the future of Kimera SWM is bright and multifaceted: - It could handle images, sound, and beyond, treating them with the same entropy lens to decide what to remember. - It could become more self-managing, adjusting its own knobs as it experiences the world, thereby reducing the need for constant human tuning. - It is well-poised for the move to the edge, enabling smarter local AI that respects resource limits and privacy. - It even has the potential to serve as the backbone of a collaborative memory across devices, though that remains a more complex endeavor.

Each of these developments would further validate the core idea that semantic memory can be engineered like a thermodynamic system – balanced, efficient, and adaptive.

Conclusion and Engineering Appendix

Conclusion

Kimera SWM (Semantic Working Memory) represents a novel synthesis of semantic memory architecture with principles drawn from thermodynamics and information theory. In this comprehensive exploration, we

have presented a system design that is both conceptually innovative and grounded in practical engineering feasibility. Key contributions of the design include:

- **Event-Centric Memory Units:** By introducing *scars* as structured event records encoded in the *EchoForm* grammar, we ensure that memory is stored at a meaningful granularity. This allows the system to recall and reason about past episodes in a way that mirrors human episodic memory and aligns with cognitive theories where events form the basis of recall [7]. It also standardizes memory storage, aiding consistency and comparison across events.

- **Entropy-Guided Memory Management:** We applied the concept of entropy to measure and control the information content of the working memory. This led to mechanisms for *pruning* and *compression* that eliminate low-value data (analogous to removing low-entropy, uninformative parts of a model [1]) and preserve high-value content (ensuring mutual information is retained in the knowledge base [2]). The **entropy tension field** concept further provides a dynamic equilibrium model, whereby the system self-balances between order and disorder, preventing both chaotic overload and oversimplified forgetting.

- **Consistency Preservation via Contradiction Engine:** Recognizing that any persistent memory must deal with accumulating inconsistencies, we included a contradiction detection and resolution module. Inspired by approaches like the Aletheion Agent's recursive consistency checks [10], our contradiction engine ensures that the set of stored beliefs/events remains largely self-consistent or at least flags uncertainties clearly. This is crucial for reliability – an AI that contradicts itself is not trustworthy. By catching contradictions, we either resolve them or isolate them, maintaining the overall structural integrity of the agent's knowledge.

- **Non-Tokenized, Contextual Recall:** Memory retrieval in Kimera SWM operates on a semantic, context-driven basis. This not only makes recall more flexible and robust to wording changes but also dovetails with the idea that memory should be accessed when contextually relevant (much like cue-based recall in humans [12]). By avoiding dependence on exact token matches, we increase the chance the system finds relevant past knowledge whenever it's needed, and does so efficiently through structured indices.

- **Realizability and Efficiency:** Throughout the paper, we addressed how each component can be implemented with existing data structures and algorithms, from hash maps to priority queues. We discussed computational overheads and ways to mitigate them (like incremental entropy updates and background pruning tasks). This shows that Kimera SWM is not just a theoretical concept; it's an engineering blueprint that could be coded and run on current hardware. Moreover, aligning with Landauer's principle [4] and related physical insights has an added benefit: it implicitly guides us to designs that are energy-efficient – a trait desirable in any modern computing system, especially for edge devices.

- **Adaptability and Future-Proofing:** We also critically examined the design for blind spots and highlighted future improvements, such as hierarchical memory layers and multimodal integration. This forward-looking analysis ensures that the core design is not a dead-end but rather a foundation that can evolve. In particular, the capacity to incorporate multimodal data and to self-optimize memory retention policies will make Kimera SWM applicable to a wide range of AI systems, from personal assistants to autonomous drones.

In closing, Kimera SWM offers a path towards AI systems that can **remember and manage knowledge in a human-like yet principled way**. It treats memory as a dynamic system where information is conserved, entropy is monitored, and meaningful structure is maintained without external intervention. By enforcing engineering realism in every aspect, we have confidence that such a system can be built and experimented with. The hope is that Kimera SWM could serve as a critical component in creating AI agents that are not only intelligent in the moment but also historically informed, self-consistent, and efficient over the long term.

Engineering Appendix

This appendix provides additional technical details, formulas, and pseudo-code snippets to complement the main text, serving as a quick reference for implementation and validation of the Kimera SWM design.

A. Entropy and Information Metrics

- **Landauer's Limit:** Erasing a bit of information has a minimum energy cost:

$$E_{\text{erase}} \geq k_B \, T \ln 2,$$

  where k_B is Boltzmann's constant and T is temperature in Kelvin [4].
  Interpretation: While our system is software, this principle underlines that deleting information (pruning memory) isn't free in a physical sense. It encourages minimal necessary deletions and suggests monitoring the volume of bits pruned as a proxy for energy usage.

- **Shannon Entropy (Memory State):** We define the entropy of a field X in memory (e.g., the distribution of an event attribute) as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x),$$

  estimated by frequency of values in the memory. For an individual event e, with attributes $X_1, X_2, \dots, X_n$, one could define a rough entropy contribution as:

$$h(e) = \sum_{i=1}^{n} -\log_2 p(X_i = x_i),$$

  summing surprisals for each attribute value it carries. This treats each attribute as independent for simplicity. The system can maintain counts to compute $p(X_i = x)$ on the fly.

- **Mutual Information Preservation:** For two random variables (or two layers of network) U and V, mutual information $I(U;V) = H(U) + H(V) - H(U,V)$. In pruning, if removing an event e disconnects two parts of memory (think of those parts as U and V), we want to avoid a large drop in $I(U;V)$. Practically, before removing e, check if e carries unique links: compute connectivity or conditional entropy with and without it. We don't have an exact formula here, but use

heuristic: if e is the only path between knowledge of type A and B, I loss would be high – so don't prune.

- **Information Concentration (IC) – from draft notes:** An informal metric mentioned in planning was:

$$IC = \frac{\text{rank}(W) \cdot H(\text{activations})}{\|W\|_0},$$

used in context of networks [17]. In our memory terms, we could analogously define:
IC_{\text{memory}} = \frac{\text{#distinct patterns} \times H(\text{memory})}{\text{#events stored}},

where #distinct patterns could be thought of as something like the rank of a matrix representing memory. This is an exploratory metric to quantify how efficiently information is packed. A high IC means we get a lot of entropy per event stored (good compression). We would aim to maximize this via pruning.

B. Sample Data Structures (Pseudo-Code)

```
# Core data structures
memory_store = {}  # dict: event_id -> event dict
index = {
    "actors": {},    # e.g., "robot1" -> set of event_ids
    "action": {},
    "object": {},
    "location": {},
    # ... other fields
}

# Example event record
event_id = 42
event = {
    "id": 42,
    "actors": ["robot1", "user7"],
    "action": "pickup",
    "object": "itemX",
    "location": "Room101",
    "time": "2025-06-01T10:00:00Z",
    "outcome": "success",
    "cause": 41  # linked to previous event
}
memory_store[event_id] = event
# Update indices
for field, val in event.items():
    if field in index:
        if isinstance(val, list):
```

```
            for v in val:
                index[field].setdefault(v, set()).add(event_id)
        else:
            index[field].setdefault(val, set()).add(event_id)
```

C. Pruning Algorithm Outline

```
function prune_memory(target_size):
    compute entropy_score[e] for each event e in memory_store:
         entropy_score = h(e) (as defined above)
         influence_score = number of events depending on e (links or queries)
         combined_score = f(entropy_score, influence_score, age(e))
         # e.g., could be entropy_score + λ * (1/(1+influence_score)) - μ *
age_weight
    sort events by combined_score ascending
    while memory_store.size > target_size:
         candidate = next event in sorted list
         if influence_score(candidate) > threshold:
             continue  # skip crucial events
         remove candidate from memory_store
         for field in index:
             remove candidate.id from index[field] lists
         mark candidate as pruned (move to archive if needed)
```

Tuning parameters: - `λ` could weight influence (to heavily penalize pruning of influential events). - `μ` could incorporate age (so older events get slightly lower score, hence more likely to prune if not otherwise important).

D. Contradiction Check Examples

- State conflict rule (in pseudo-code):

```
def check_state_conflict(event):
    # If event has a state field (or action implies a state change)
    state_info = extract_state(event)  # e.g., ("deviceX","ON")
    if state_info:
        entity, state = state_info
        if entity in state_index:  # state_index could track last known
state
            last_state, last_time = state_index[entity]
            if last_state != state and events_overlap(last_time,
event["time"]):
                flag_contradiction(entity, last_state, state, event["id"])
```

```
    # Update state_index
    state_index[entity] = (state, event["time"])
```

• Causal loop check:

```
def check_causal_loop(event):
    # simple detection: if an event indirectly causes itself
    seen = set()
    def dfs(eid):
        if eid in seen:
            return True  # loop found
        seen.add(eid)
        for out in causal_links.get(eid, []):
            if dfs(out):
                return True
        return False
    if dfs(event["id"]):
        flag_contradiction("causalloop", event["id"])
```

E. Memory Recall Trigger (Cue Match Pseudo-code)

```
def recall_by_context(current_event):
    # Use partial match: find events sharing actor or location
    related = set()
    for field in ["actors","location","action"]:
        if field in current_event:
            val = current_event[field]
            if isinstance(val, list):
                for v in val:
                    related |= index[field].get(v, set())
            else:
                related |= index[field].get(val, set())
    # Filter by time or other context if needed
    recalls = []
    for eid in related:
        e = memory_store[eid]
        # maybe ensure it's not too old or too general
        if similarity(e, current_event) > SIM_THRESHOLD:
            recalls.append(e)
    return sorted(recalls, key=lambda e: similarity(e, current_event),
reverse=True)[:5]
```

Where `similarity(e, current_event)` could be a simple count of matching fields or a learned function.

F. Performance Considerations:

- With N events in memory, typical operations complexities:
- Adding an event: O(number of indexed fields) ~ O(F).
- Removing an event: O(F).
- Query by exact match on one field: O(results) (due to direct index hit).
- Query by two fields: O(min(results_field1, results_field2)) for intersection.
- Entropy update: O(1) per event add/remove per distribution tracked.
- Pruning cycle: O(N log N) for sorting scores, but N is limited by design.
- Contradiction check on insert: O(k) with k related events, usually much smaller than N.
- Memory overhead: indices roughly use memory proportional to N * F (but each just storing IDs, possibly much smaller than events themselves). This is manageable if F is moderate (tens of fields).
- If N gets large (which we try to avoid via pruning), we might consider external storage for older events or a more scalable database, but then we depart from fully self-contained. Our assumption is that by the time N threatens system limits, pruning will have kicked in.

G. Verification and Testing:

During implementation, one should test components in isolation: - Generate synthetic events with known patterns to verify entropy calculations (e.g., 50% of events one type, 50% another => entropy ~1 bit). - Test pruning logic by injecting dummy events and seeing if low entropy ones get removed and critical ones stay. - Simulate a contradiction (like add two opposing events) and ensure the engine flags it. - Simulate a recall scenario: given an event, pre-fill memory with one clearly related and one unrelated event, see if recall_by_context returns the related one.

By approaching testing in this controlled manner, we ensure each piece behaves as expected before integrating into an agent.

H. Related Concepts and References (for further reading):

- *Entropy-Based Pruning:* NEPENTHE method for layer removal [18].
- *Mutual Information Pruning:* MIPP for preserving info flow [8].
- *Thermodynamic Memory:* Discussion on traces of the past and entropy in memory [5] [6].
- *Human Memory Modeling:* Memory recall and consolidation factors (context, time, frequency) [13].
- *Entropic Memory Models:* Entropic associative memory storing multimodal info [15] [14].

These provide theoretical backing and potential inspiration for refining Kimera SWM.

This concludes the white paper on Kimera SWM. The design presented marries theoretical rigor with actionable engineering, laying the groundwork for semantic memory systems that are efficient, consistent, and cognizant of the informational energy they steward. We anticipate that future iterations and implementations will further validate and enrich the concepts detailed here.

---

[1] [11] [18] openreview.net
https://openreview.net/pdf?id=fk5ePN7YCS

2 8 9 openreview.net
https://openreview.net/pdf?id=2IhkyiF3to

3 12 13 ”My agent understands me better”: Integrating Dynamic Human-like Memory Recall and Consolidation in LLM-Based Agents
https://arxiv.org/html/2404.00573v1

4 Landauer's principle - Wikipedia
https://en.wikipedia.org/wiki/Landauer%27s_principle

5 6 Memory and Entropy
https://www.mdpi.com/1099-4300/24/8/1022

7 Event boundaries in memory and cognition - ScienceDirect.com
https://www.sciencedirect.com/science/article/abs/pii/S2352154617300037

10 AI 2027: The Scenario That Collapses Itself
https://www.linkedin.com/pulse/ai-2027-scenario-collapses-itself-constantine-vassilev-1tz6c

14 15 Imagery in the entropic associative memory | Scientific Reports
https://www.nature.com/articles/s41598-023-36761-6?error=cookies_not_supported&code=b1dd18eb-cc98-44e4-82fa-b416faa3172e

16 17 Safari.pdf
file://file-3yz2knG9HiHXwAbvbYaQtX