ChatGPT

Shared Memory (Blackboard) vs. Event-Sourced Sessions: State Management Paradigms

Introduction

Designing the **state management layer** for an AI system (such as an "AI OS" with multiple agents or processes) can follow different architectural paradigms. Two fundamental approaches are: **(1) a Shared Memory/Blackboard architecture**, and **(2) an Event-Sourced Sessions architecture**. The choice between these is about *how state is organized and updated conceptually*, rather than any specific technology (database, Redis, etc.). In fact, the current prototype simply stores state in a Git repository (files like `SYSTEM_SNAPSHOT.md`, `AUTOMATIONS_REGISTRY.jsonl`, etc.), which provides a basic persistent state with history. Moving forward, it's crucial to focus on the *models and patterns* behind state management, not on the particular stack. Below we explore both paradigms, their principles, and their implications, with illustrative examples in pseudocode.

Shared Memory / Blackboard Architecture

In a **blackboard architecture**, all components of the system share a common global state – the *"blackboard"* – which they can read from and write to. This approach originates from AI systems research, where a *blackboard system* is defined as an AI model in which "a common knowledge base, the *'blackboard'*, is iteratively updated by a diverse group of specialist knowledge sources". Each module (or *knowledge source*) contributes partial results to the shared state when certain conditions are met, and in turn can react to the contributions of others. In this way, multiple specialized agents collaborate on solving a problem by incrementally building the solution on the blackboard.

How it works: A classic metaphor is a team of experts gathered around a single blackboard in a room. The session begins by writing the problem or goal on the board; then all experts watch the board and opportunistically contribute when they have something to add. For example, when one expert writes a partial solution, it may enable another expert to contribute the next step. This process continues until the problem is solved or goal achieved. The blackboard thus serves as a **shared memory** and communication medium for the agents. Importantly, a *control mechanism* (the *control shell*) oversees the process, deciding which agent should act next and when to avoid conflicts or chaos. Just as a moderator prevents everyone from grabbing the chalk at once, the control component orchestrates access to the blackboard so that only one update happens at a time or in a coherent sequence.

Key characteristics of the Blackboard approach include:

- **Shared Global State:** The blackboard holds a unified state (problems, partial solutions, hypotheses, etc.) that *all agents can access and modify*. This is a very **collaborative model**, ideal for scenarios where different modules contribute different pieces of knowledge towards a common goal.
- **Opportunistic Updates:** Agents don't have a fixed order of execution; instead, they *monitor* the shared state and **trigger** when they recognize a condition where they can contribute. This leads to an **opportunistic problem-solving** process – e.g. if agent A's output creates a new data point

on the board that agent B was waiting for, agent B can now act. The sequence of actions emerges from the data, rather than a predetermined script.
- **Control and Coordination:** Because all agents write to one place, a control logic is needed to prevent race conditions and ensure consistency. The control shell can use priorities or conditions to decide which knowledge source runs next. This avoids two agents writing to the same part of the state simultaneously in inconsistent ways.
- **Flexible Knowledge Sharing:** Every agent has *full visibility* into the current state on the board. This can be powerful – for example, an agent can build on any information added by others. It's akin to *"all the papers laid out on one table"* that everyone can read or annotate. No explicit message passing is required since writing to the board implicitly communicates information to all other agents.
- **Complexity of Shared State:** The downside is that a blackboard can become a **"big bag of shared state"** if not designed carefully. Modern software practices favor encapsulating state, whereas the blackboard is the opposite – many actors sharing global data. This free-form sharing can lead to complex interdependencies and make it hard to reason about the state. One developer analogously noted that without clear structure, "too much state [becomes] available to too many actors," which can complicate the system. In practice, the blackboard pattern is most beneficial in *collaborative AI/agent scenarios* (its original intent) and can be overkill or even counterproductive in simpler contexts. Proper design of the data model on the blackboard and strong coordination logic are needed to mitigate chaos.

Illustration – Blackboard Example: Below is a simple pseudocode illustration of a blackboard system. In this toy example, we have two "agent" functions collaborating on a task (say, designing and implementing a feature) via a shared dictionary acting as the blackboard. A central loop (control) triggers agents opportunistically based on the state:

```
# Shared blackboard (global state repository)
blackboard = {}

# Knowledge Source 1: e.g., Architecture Designer
def agent_designer():
  # This agent runs if requirements are present but no design yet
  if "requirements" in blackboard and "design" not in blackboard:
    spec = blackboard["requirements"]
    # Produce a design based on requirements
    blackboard["design"] = f"System design for {spec}"
    print("Designer: Added system design to blackboard")

# Knowledge Source 2: e.g., Implementer
def agent_implementer():
  # This agent runs if a design is available but no implementation yet
  if "design" in blackboard and "implementation" not in blackboard:
    design = blackboard["design"]
    # Produce an implementation based on the design
    blackboard["implementation"] = f"Code implementing ({design})"
    print("Implementer: Added implementation to blackboard")

# Initial problem specification
blackboard["requirements"] = "User Authentication Feature"
```

```python
# Control loop: keep allowing agents to run until solution is complete
while "implementation" not in blackboard:
  agent_designer()
  agent_implementer()

print("Solution complete:", blackboard["implementation"])
```

In this simplistic flow, the **Designer** agent sees the "requirements" on the board and adds a "design" to the shared state. That new state then triggers the **Implementer** agent to produce an "implementation". The control loop ensures each agent gets a chance to run when its conditions are met, and stops when the final solution is reached. In a real system, the control logic could be more complex (scheduling, conflict resolution, etc.), and the shared state could contain rich data structures (like the real files in the Git repo – e.g. a system snapshot or registry in a structured format). The key takeaway is that all intermediate results were stored in one common place that both agents read and write, illustrating the blackboard paradigm.

Event-Sourced Sessions Architecture

In an **Event-Sourced** architecture, the state is not maintained as one mutable whole, but rather *reconstructed from a log of events*. **Event Sourcing** means that **every change to the system's state is captured as an event object and appended to an event log**, instead of overwriting a central state in place [1]. The sequence of events *is* the source of truth: to get the current state, the system *replays* or applies all the events in order. In other words, the state is **defined by the history of actions** that occurred, rather than a single authoritative snapshot. This approach is often used with a **thread or session context**, meaning events are grouped by context (for example, per user session, per conversation thread, or per domain entity). Each session or thread has its own stream of events that can be independently recorded and analyzed.

How it works: Think of a session (or an agent's activity thread) as a timeline. Every significant action – user input, agent decision, state update, etc. – generates an *event* which gets logged in chronological order. No state is ever mutated directly; instead, if an agent wants to change something, it emits an event (e.g. "Fact X was added" or "Action Y was taken"). The *cumulative effect* of all these events produces what we consider the state of that session. Formally, "whenever the state of an entity changes, a new event is appended... The application reconstructs an entity's current state by replaying the events" [1]. This design guarantees that **every change is recorded**, creating a complete history of the session's state evolution. We can query not only the latest state, but also examine the past states or events to see *how* we got there. The system can even rewind or branch in time by choosing a point in the event log and replaying from there (much like checking out an older commit in Git, or branching off a previous state).

In practice, implementing event sourcing might involve storing events in an append-only log or database table, with fields like timestamp, event type, and details. The *State* of a given session is then a derived construct: the result of applying all events in that session's log. Optionally, the system might store periodic **snapshots** of the state for efficiency (so it can start from a snapshot and apply only new events, rather than replay from the very beginning every time). But the golden record remains the sequence of events.

Key characteristics of the Event-Sourced approach include:

- **Complete History & Auditability:** Because *all changes are stored as events*, the system inherently keeps a full log of what happened and when. You can always answer "who did what, and in what order" by inspecting the event log. This is extremely useful for debugging and accountability. For example, in a voice assistant scenario, every dialogue turn and state modification can be logged; if something went wrong, you can replay the exact sequence of events that led to the issue. This traceability means every decision made by an AI agent is *reproducible*: given the same prior events, the state would be the same.
- **Reconstructing or Rewinding State:** Since current state is derived from events, we can **recompute state at any past point in time** by replaying events up to that moment. If an error is detected, one can *rewind* the state by removing or altering the offending event and then replaying subsequent events. In practice, this might look like: "Made a mistake? Just remove the last few events and rebuild the session state". For instance, if an agent's action turned out to be wrong, you can strike that event from the log and recalc the state as if it never happened, then perhaps insert a corrected event. This ability to **"time-travel"** in state (retroactive fixes, what-if scenarios, branching) is a powerful advantage of event sourcing. It's analogous to version control in software: you have the entire commit history, you can checkout an old version or create a new branch from it. In fact, using a Git repository for state (as is being done currently) is conceptually an event-sourced approach – each commit is like an event, and the files represent state snapshots.
- **Isolation by Session (Contextual Streams):** By organizing events **per session or thread**, each context's state is isolated to its own event stream. This naturally prevents interference between different sessions – e.g. events in Session A don't directly affect Session B's state unless explicitly linked. Concurrency is easier to manage at this logical level: multiple sessions can progress in parallel, each appending to its own log. Even within a single session, if multiple agents contribute, they do so by generating events, which can be interleaved in a controlled sequence. The log provides a single, ordered source of truth, so even distributed components can collaborate without a shared mutable memory – as long as they all append and read the event store in order, consistency is maintained. (Under the hood, ensuring a consistent total order of events per session is critical, which might involve using timestamps or a centralized event broker to serialize them.)
- **Debugging and Observability:** An event-sourced system is highly **observable**. Since every state change is explicit (as an event), developers and tools can inspect the logs to understand system behavior. One real-world implementation of event-sourced conversations noted that this approach yielded a "robust, observable, and flexible system" – the team could handle concurrency, recover from errors gracefully, and maintain complex context easily. The transparency of the log means failures can be analyzed post-mortem by retracing event sequences, and test scenarios can be reproduced by replaying the same events. This greatly improves reliability and developer productivity in complex AI systems.
- **Overhead and Complexity:** Event sourcing does introduce some **overhead in implementation**. The system must define clear event schemas and maintain them (which can require versioning as the system evolves). Storing every event can consume more space than storing just the latest state, and reconstructing state via replay can be slower than a direct read of stored state (though snapshotting and efficient event storage mitigate these issues). There is also a mental shift for developers: instead of thinking "set value X in the state," one must think "emit event X_occurred." However, many find that this upfront complexity "is not that high, and it pays dividends in system reliability, compliance capabilities, and developer productivity". In systems where understanding and controlling the evolution of state is important (such as financial systems, or multi-step AI reasoning), the benefits often outweigh the costs.

Illustration – Event-Sourced Session Example: To illustrate event sourcing, consider a simplified session that maintains a few pieces of state. We will simulate an event log and how the state is derived from it:

```
# Define a list to hold events for one session (event log)
session_events = []

# Helper: apply events to derive session state
def get_session_state(events):
  state = {}
  for evt in events:
    event_type, key, value = evt # simple event tuple (type, key, value)
    if event_type == "SET":
      state[key] = value      # setting/updating a field
    elif event_type == "DELETE":
      state.pop(key, None)    # deleting a field
    # (Other event types could be handled here)
  return state

# Simulate some events in the session:
session_events.append(("SET", "user", "Alice"))      # user Alice logged in
session_events.append(("SET", "auth", False))       # authentication status
False
session_events.append(("SET", "auth", True))        # authentication status
True (user authenticated)

# Compute current state by replaying events:
current_state = get_session_state(session_events)
print("Current session state:", current_state)
# Output: Current session state: {'user': 'Alice', 'auth': True}

# The full history of changes is preserved in session_events.
# If needed, we could rewind:
session_events = session_events[:-1] # remove last event (auth True)
rewound_state = get_session_state(session_events)
print("Rewound state:", rewound_state)
# Output: Rewound state: {'user': 'Alice', 'auth': False}
```

In this example, the **state** after all events is `{'user': 'Alice', 'auth': True}` – reflecting that Alice is logged in and authenticated. But we didn't store that state directly; we derived it by applying each event in order. We also retain the knowledge that initially `auth` was False and later became True. If we decide to **undo** the last action, we can drop the last event and recompute: the state then reverts to `{'user': 'Alice', 'auth': False}`. This is a trivial example, but it demonstrates how event sourcing works. In a larger system, events would be richer (with types like "UserLoggedIn", "CheckedPermission", "SentMessage", etc., possibly stored as structured records). The core idea is that the log of events is the canonical source of state, and the system can regenerate any session's context from this log at any time. Tools or functions like `GetSession(events)` (as seen in some real implementations) encapsulate this replay mechanism to produce a convenient state object for use in prompts or decision-making.

Comparing the Paradigms and Choosing a Focus

Both the blackboard approach and the event-sourced approach aim to manage state for complex, multi-step AI processes, but they do so in fundamentally different ways:

- **Data Sharing vs. Event Communication:** The blackboard is a *shared data space* – agents communicate implicitly by reading/writing shared memory. Event sourcing, by contrast, treats communication as first-class: state changes are *messages (events)*, not direct writes. This reflects the general software principle "*share memory by communicating, not communicate by sharing memory*": the blackboard leans toward the former (shared-memory concurrency), whereas event-sourced systems lean toward the latter (message-passing style concurrency). In practical terms, blackboard updates happen in-place, while event-sourced updates happen by appending to a log and later interpretation.
- **Consistency and Control:** In a blackboard system, maintaining consistency is the responsibility of the control mechanism and the careful design of shared data structures. In an event system, consistency is maintained by the ordering of events – if two updates happen “at the same time”, they are queued as events in some sequence. With proper design, event sourcing can naturally serialize concurrent modifications (each new event goes on the log in a definite order). Blackboard systems might require explicit locking or scheduling to avoid concurrent writes. Essentially, blackboard trades *immediate, direct data access* for the need to orchestrate access; event sourcing trades *simplicity of a single state* for an explicit log that can be managed and ordered.
- **State Evolution Transparency:** Event logs provide built-in transparency of how state evolves over time. Blackboard state typically provides only the *current* picture unless you manually preserve history. It is possible to hybridize these – for instance, one could use a blackboard but still log every change as an event (for audit purposes). But in pure form, the blackboard’s strength is *current shared truth*, while event sourcing’s strength is *historical trace plus the ability to derive truth*.
- **Use Cases:** A **shared memory/blackboard** paradigm shines in scenarios where *many components need to collaborate tightly on the same knowledge at once*. Classic examples are expert systems or modern multi-agent AI setups where agents with different expertise write on a common board until a solution emerges. It’s a natural fit if you envisage the system as a team discussion (and indeed, some recent LLM-based systems have revisited blackboard designs to let multiple AI “experts” work together with full information sharing). On the other hand, **event-sourced sessions** are advantageous when you have distinct *streams of actions or conversations* that need to be tracked over time – such as dialogue sessions, user workflows, or any process where tracking the sequence of steps and being able to rewind or branch is important. Event sourcing is also common in distributed systems and microservices, since it facilitates decoupled components and reliable messaging. In an AI orchestration context, if you anticipate needing robust recovery from errors, or auditing every step an autonomous process took, event sourcing is very appealing.

Emphasis on Concepts, not Technology: Crucially, these paradigms are *architectural models*. Either pattern can be implemented with various technologies. For example, a blackboard could be backed by an in-memory data store or a centralized database or even a simple file that all agents read/write (the current Git-based state files are essentially acting as a primitive blackboard where different processes could read/write shared state). Event logs could be implemented via an append-only file, a message queue (Kafka, etc.), a relational table of events, or again even a Git commit history. The choice of **DB vs. Redis vs. file** is secondary – the primary design decision is how you want your system to **treat state changes**: as collaborative edits to a shared state, or as append-only facts that are later aggregated.

Once that conceptual decision is made, the technology can be chosen to suit performance and scaling needs.

In summary, the **Shared Memory/Blackboard** model provides a *single evolving state* that all components concurrently work on, enabling rich collaboration at the cost of potential complexity in managing access. The **Event-Sourced** model provides a *chronological ledger of states* that ensures nothing is lost or overwritten, enabling superb traceability and recovery at the cost of managing an event pipeline and reconstruction logic. The best approach depends on the system's requirements. Often, elements of both can even coexist: for instance, using event sourcing under the hood for durability and debugging, while presenting a shared-state view for agents to work with at runtime. Regardless, the focus at this stage should be on understanding these paradigms deeply. By grasping their concepts, one can then implement the chosen pattern with any suitable stack (from a simple Git-based log to a cloud database) with confidence that the architectural foundation is sound.

---

1 Pattern: Event sourcing
https://microservices.io/patterns/data/event-sourcing.html