Make Agents Remember the Right Context at the Right Time

"My agent had perfect reasoning, perfect tools,
and a 77% failure rate.
The problem wasn't intelligence — it was memory."

Anannya Roy Chowdhury

Gen AI Developer Advocate & Architect

AWS Developer Experience

THE GAME: A MEMORY STRESS TEST

Horcrux Hunt: Two AI agents in an adversarial information game

⚔️ HARRY (Protagonist)

LLM-powered agent (Strands SDK + Bedrock)

- Must remember 50 turns of signals
- Track probabilities across 15 locations
- Distinguish decoys from real Horcruxes
- Reduce uncertainty with each observation

🐍 VOLDEMORT (Adversary)

Heuristic + LLM fallback agent

- Actively corrupts Harry's memory
- Relocates Horcruxes mid-game
- Plants decoys to inject false signals
- Increases uncertainty every turn

`Stack: Strands SDK · Amazon Bedrock · Lambda · DynamoDB · Streamlit`

"Two agents. 50 turns. 15 locations. The game is fundamentally an information problem — Harry reducing uncertainty, Voldemort increasing it."

THE MEMORY FAILURES

Real failure scenarios — each is a memory problem, not a reasoning problem

FAILURE	WHAT HAPPENED	ROOT CAUSE
Destroyed empty location	Voldemort relocated same turn	STALE MEMORY Acting on outdated belief
Re-searched confirmed empty	History too long, signal buried	RETRIEVAL FAILURE Relevant signal lost in noise
Agent timed out	6,000 tokens of context, 25s gen	MEMORY OVERLOAD Too much remembered
Ignored positive signal	Critical signal on turn 3, now turn 40	DECAY PROBLEM Old signals not weighted

"The agent could reason perfectly. It just remembered the wrong things."

THE MEMORY TAX

Context windows are a compounding tax, not a flat fee

TURN	HARRY'S CONTEXT	VOLDEMORT'S CONTEXT	COST MULTIPLIER
1	~800 tokens	~1,200 tokens	1×
10	~1,800 tokens	~2,500 tokens	2.25×
25	~3,500 tokens	~4,200 tokens	4.4×
50	**~6,000 tokens**	**~7,500 tokens**	**7.5×**

"You pay for FULL context replay. Every. Single. Turn."

Like re-reading the entire book every time you turn a page.

2 agents

2×

token curve

3 agents

5.8×

token overhead

Output tokens

5× input price

+ 12ms/token sequential

THE ENTROPY RACE

Information Theory of Agent Memory

Shannon Entropy Over Game Turns

```
Turn  1:  H = log₂(15) = 3.9 bits   (maximum uncertainty)
Turn 10:  H ≈ 2.8 bits              (narrowing)
Turn 30:  H ≈ 1.4 bits              (confident)
Turn 50:  H ≈ 0.6 bits              (nearly certain)
```

⬇️ Harry's job: REDUCE entropy

Each observation narrows uncertainty

⬆️ Voldemort's job: INJECT entropy

Relocations and decoys increase chaos

KEY INSIGHT

Memory architecture IS entropy management

What you keep, what you compress, what you discard determines how fast entropy decreases.
Each agent you add = another source of entropy in your system.

THE THREE-LAYER MEMORY FRAMEWORK

LAYER 1: WORKING MEMORY (Context Window) $$$

- What the agent sees RIGHT NOW
- Expensive: every token = $ every turn
- Goal: MINIMAL, COMPRESSED, RELEVANT

▼

LAYER 2: RETRIEVAL MEMORY (Structured + Semantic) $

- What the agent CAN access on demand
- Bayesian maps, belief states, indexes
- Goal: RIGHT context at RIGHT time

▼

LAYER 3: PERSISTENT MEMORY (Event Store + Long-Term) ¢

- What the agent has EVER experienced
- DynamoDB, event logs, preference stores
- Goal: NEVER forget, NEVER re-process

LAYER 1 FIX: Dynamic Context Compression

What to keep, compress, or discard from working memory

BEFORE (naive — everything in context)

```
Turn 1: Harry searched Hogwarts → negative
Turn 2: Harry searched Diagon Alley → positive
Turn 3: Harry destroyed at Diagon → decoy
... (50 lines = 2,000+ tokens)
```

AFTER (compressed AgentContext)

```
AgentContext(
  turn_number=25,              # 1 token
  remaining_budget=1,          # 1 token
  available_tools=["ron"],     # 3 tokens
  cooldown_states={"dumbledore": 2},
  belief_map={
    "Hogwarts": 0.34,
    "Azkaban": 0.22
  }                            # 30 tokens
)  # TOTAL: ~55 tokens
```

97% COMPRESSION

2,000+ tokens → 55 tokens

Intentional lossy compression — deliberately discard:

- Full turn history → replaced by probability scores
- All 15 locations → filtered to only 2-4 valid targets
- Previous agent reasoning → discarded entirely

"Less information = fewer tokens = lower cost... AND the system works BETTER"

LAYER 2 FIX: Bayesian Belief Maps

Replace tokens with math — the LLM should CONSUME memory, not PRODUCE it

```
class BeliefMapManager:
    def update_on_signal(self, loc, signal):
        if signal == POSITIVE:
            self.beliefs[loc] *= 0.9
        elif signal == NEGATIVE:
            self.beliefs[loc] *= 0.1
        elif signal == DESTROYED:
            self.beliefs[loc] = 0.0
        self.normalize()
```

What the LLM actually sees:

```
"top_target: Hogwarts (p=0.34)"
```

8 tokens. Not 5,000.

APPROACH	TOKENS	LATENCY	COST
LLM reads 50 turns	5,000	8s	$0.015
Bayesian belief update	0	<1ms	$0.000

Why this beats naive RAG for structured domains:

- RAG retrieves text → still costs tokens | Belief map retrieves probabilities → ~30 tokens total
- RAG needs vector DB infrastructure | Belief map = in-memory dict
- RAG adds latency (embedding + search) | Belief map = O(1) lookup

LAYER 2 FIX: Entropy-Gated Retrieval

Match memory depth to decision difficulty — dynamic memory routing at runtime

```
def choose_strategy(belief_map):
    entropy = -sum(
        p * log2(p) for p in beliefs.values() if p > 0
    )

    if entropy < 1.0:     # low uncertainty
        return "heuristic"    # don't even ask LLM
    elif entropy < 2.5:   # medium uncertainty
        return "compressed"   # send belief map only
    else:                 # high uncertainty
        return "full_context" # send richer retrieval
```

ENTROPY GATING = Dynamic Memory Routing

LOW ENTROPY (< 1.0)

Heuristic • 0 tokens • <1ms

MEDIUM ENTROPY (1.0-2.5)

Belief map • 55 tokens • fast

HIGH ENTROPY (> 2.5)

Full context • justified cost

ε-greedy exploration: 90% exploit top belief / 10% random explore

Prevents memory-driven tunnel vision — the agent doesn't get trapped by its own beliefs

"Don't give every decision the same memory budget. Match memory depth to decision difficulty."

LAYER 3 FIX: Persistent Memory

What never enters the context window but is never lost

```
# Single-item DynamoDB design

{
  "session_id": "game_42",
  "game_state": full_state.to_json(),
               # 5-10KB (NEVER sent to LLM)
  "status": "active",
               # queryable without deserializing
  "winner": None,
  "ttl": epoch + 30_days
               # auto-delete, no cleanup Lambda
}
```

THE ARCHITECTURE PRINCIPLE

Layer 3 stores EVERYTHING

▼

Layer 2 COMPRESSES it

▼

Layer 1 receives ONLY what's needed

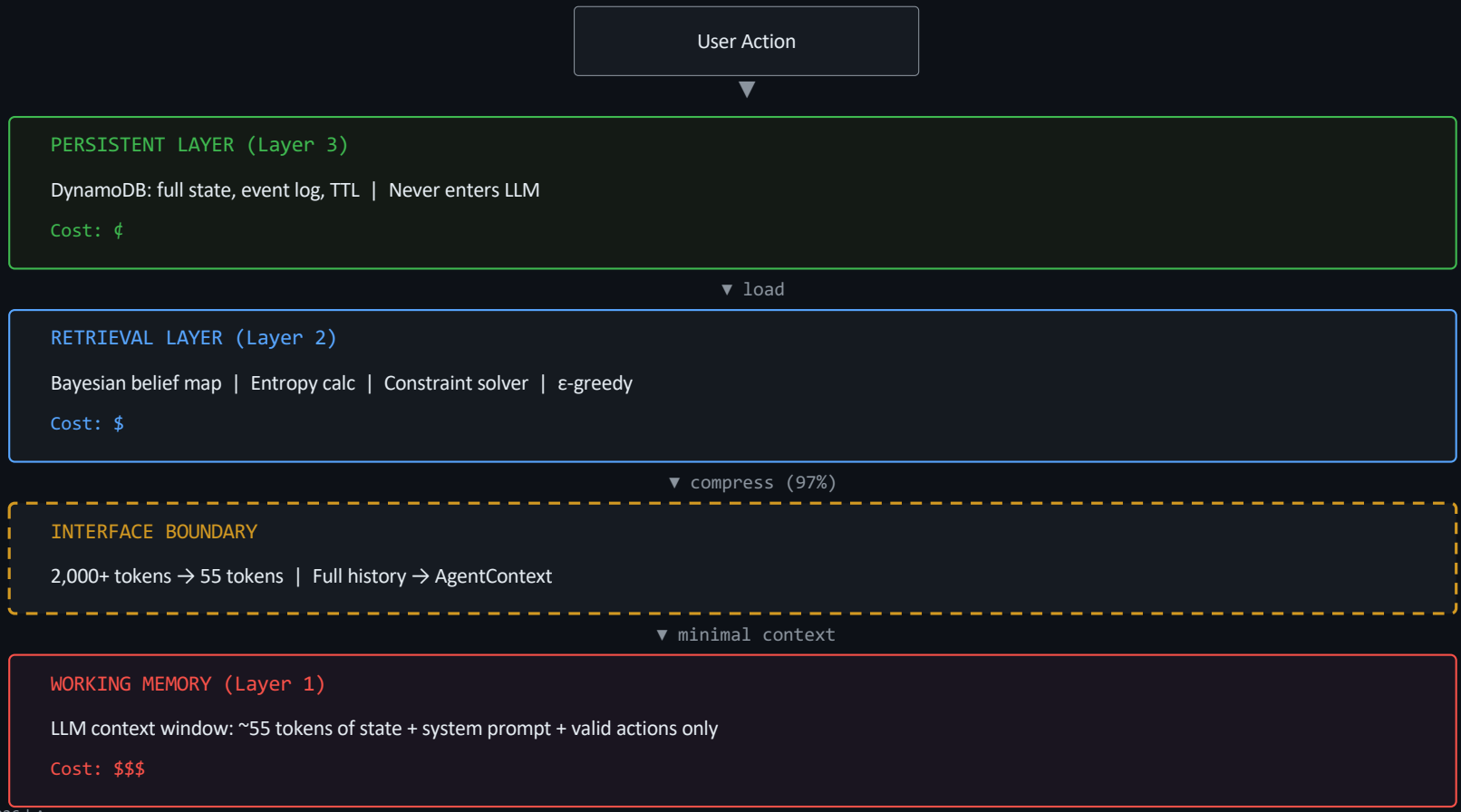
FAILURE MODES AVOIDED (without Layer 3):

1. Stuff everything in context → overload → hallucination
2. Forget everything between turns → amnesia → repeated mistakes

The fix: Persist abundantly. Retrieve selectively. Present minimally.

THE REFERENCE ARCHITECTURE

Full memory-aware agent architecture — only 2 of 8 modules ever touch the LLM



RESULTS: Memory-Optimized vs Naive

Same agents. Same reasoning capability. Radically different memory architecture.

METRIC	NAIVE (all-in-context)	MEMORY-OPTIMIZED	CHANGE
Tokens per decision	5,000	55	**-97%**
Context at Turn 50	6,000 tokens	~200 tokens	**-97%**
Cost/game	$1.95	$0.35	**-82%**
Latency/turn	12s	3s	**-75%**
Harry win rate	23%	52%	**+29pp**
"Wrong memory" failures	~20% of turns	<5%	**-75%**
Hallucination rate	~15%	~3%	**-80%**

Annual savings at scale: $576,000

MEMORY DESIGN PRINCIPLES

Five rules for production agent memory

01 The LLM should consume memory, not produce it

Structured data > natural language for state

02 Match memory depth to decision difficulty

Entropy-gate your retrieval

03 Compress at boundaries, not inside the LLM

The Interface Boundary does the work

04 Persist abundantly, retrieve selectively, present minimally

Three layers, each with a job

05 Less context = better decisions

Agents perform BETTER with focused, compressed memory

"Your agents don't fail because they can't reason.
They fail because they remember wrong.
Fix the memory. The reasoning follows."


The future is for
Agents that
remember better !

APPENDIX A

Bayesian Update Math + ε-Greedy Exploration

```
# Bayesian posterior update for belief map
P(H_i | signal) = P(signal | H_i) * P(H_i)
                / sum(P(signal | H_j) * P(H_j))

# Likelihood ratios for signal types:
# POSITIVE signal at location:  P(pos|H) = 0.9
# NEGATIVE signal at location:  P(neg|H) = 0.1
# DESTROYED (decoy) at loc:     P(H) = 0.0

# ε-Greedy action selection:
def select_action(beliefs, epsilon=0.1):
    if random() < epsilon:
        return random_location()  # explore
    else:
        return argmax(beliefs)     # exploit
```

Why ε-greedy?

- Pure exploitation gets trapped by adversarial belief manipulation
- 10% random exploration prevents Voldemort from gaming the belief map
- Balances cost (fewer LLM calls when exploiting) vs accuracy (discovering relocated Horcruxes)

Convergence Properties:

- With uniform prior (1/15 per location), system converges to <1 bit entropy in ~30 turns without adversary
- Voldemort's relocations inject ~0.5 bits/relocation, extending convergence to ~45 turns
- Decoys temporarily increase entropy by ~0.3 bits but are resolved on destruction

APPENDIX B

Entropy-Gating Thresholds — Calibration Guide

ENTROPY RANGE	STRATEGY	TOKENS USED	LATENCY	% OF DECISIONS
H < 1.0 bits	Heuristic only	0	<1ms	~35%
1.0 ≤ H < 2.5 bits	Compressed belief map	55	~2s	~45%
H ≥ 2.5 bits	Full context retrieval	200-500	~5s	~20%

How to calibrate for your domain:

1. Start with max_entropy = $\log_2(N)$ where N = number of possible states
2. Set heuristic threshold at 25% of max_entropy (confidence is high)
3. Set compressed threshold at 65% of max_entropy (moderate uncertainty)
4. Above compressed threshold = full context (genuinely uncertain)

Tuning tips:

- If win rate drops, LOWER the compressed threshold (more LLM usage)
- If cost is too high, RAISE the heuristic threshold (more heuristic usage)
- Monitor the % split weekly — adjust as your agent learns patterns

APPENDIX C

Context Orchestration Pattern

Authoritative vs Conversational Context

AUTHORITATIVE CONTEXT

System-generated, always current:

- Current game state (ground truth)
- Valid actions for this turn
- Belief map probabilities
- Cooldown states
- Turn number & budget remaining

Trust level: ABSOLUTE
Source: Game engine (not LLM)

CONVERSATIONAL CONTEXT

LLM-generated, may be stale/wrong:

- Previous reasoning chains
- Past turn narratives
- Agent-to-agent messages
- Historical observations
- Strategy explanations

Trust level: CONDITIONAL
Source: LLM outputs (may hallucinate)

DESIGN RULE

Always prioritize authoritative context over conversational context in the prompt. When they conflict, the authoritative source wins — this prevents context poisoning where outdated LLM reasoning overrides current system state.

APPENDIX D

DynamoDB Persistent Memory Schema + TTL Patterns

```
# Table: HorcruxHunt_Sessions
# Key: session_id (partition)
# GSI: status-index (status + created_at)

{
  "session_id": "uuid-v4",          # PK
  "created_at": "2024-03-15T...",   # Sort
  "status": "active|completed|timeout",
  "game_state": {
    "turn": 25,
    "harry_position": "Hogwarts",
    "horcrux_locations": [...],     # encrypted
    "belief_map": {...},
    "action_history": [...],        # full log
    "destroyed_count": 2
  },
  "metrics": {
    "total_tokens": 45000,
    "total_cost_usd": 0.85,
    "avg_latency_ms": 3200
  },
```

Design Decisions

- Single-item design
  One read = full game state
- TTL auto-cleanup
  No Lambda needed for expiry
- GSI on status
  Query active games without scan
- Metrics embedded
  Cost tracking per-session
- Never sent to LLM
  5-10KB stays in persistence
  Only 55 tokens cross boundary

DynamoDB cost: ~$0.0004/game (1 write + 50 reads) vs LLM context cost: ~$1.95/game without optimization

APPENDIX E

Cost Math + Bedrock Pricing Assumptions

```
Cost = Agents × Turns × (Context_tokens + Response_tokens) × Price_per_token
```

Scale: 1,000 games/day = $49K/month = $588K/year (naive approach)

MODEL	INPUT $/1K	OUTPUT $/1K	NAIVE COST/GAME	OPTIMIZED/GAME
Claude 3 Sonnet	$0.003	$0.015	$1.95	$0.35
Claude 3 Haiku	$0.00025	$0.00125	$0.16	$0.03
Claude 3.5 Sonnet	$0.003	$0.015	$1.95	$0.35

Cost Breakdown (naive approach, per game):

Bedrock (LLM)
67%
$1.31 / game

Lambda
26%
$0.51 / game

DynamoDB
7%
$0.13 / game