

ELIATra

The OpenSearch Experts

Smart Recall

Enhancing Local LLM Conversations with Embedding-Aware Context Retrieval

HayStack 2025

Introduction

Lucas Jeanniot

Machine Learning Engineer

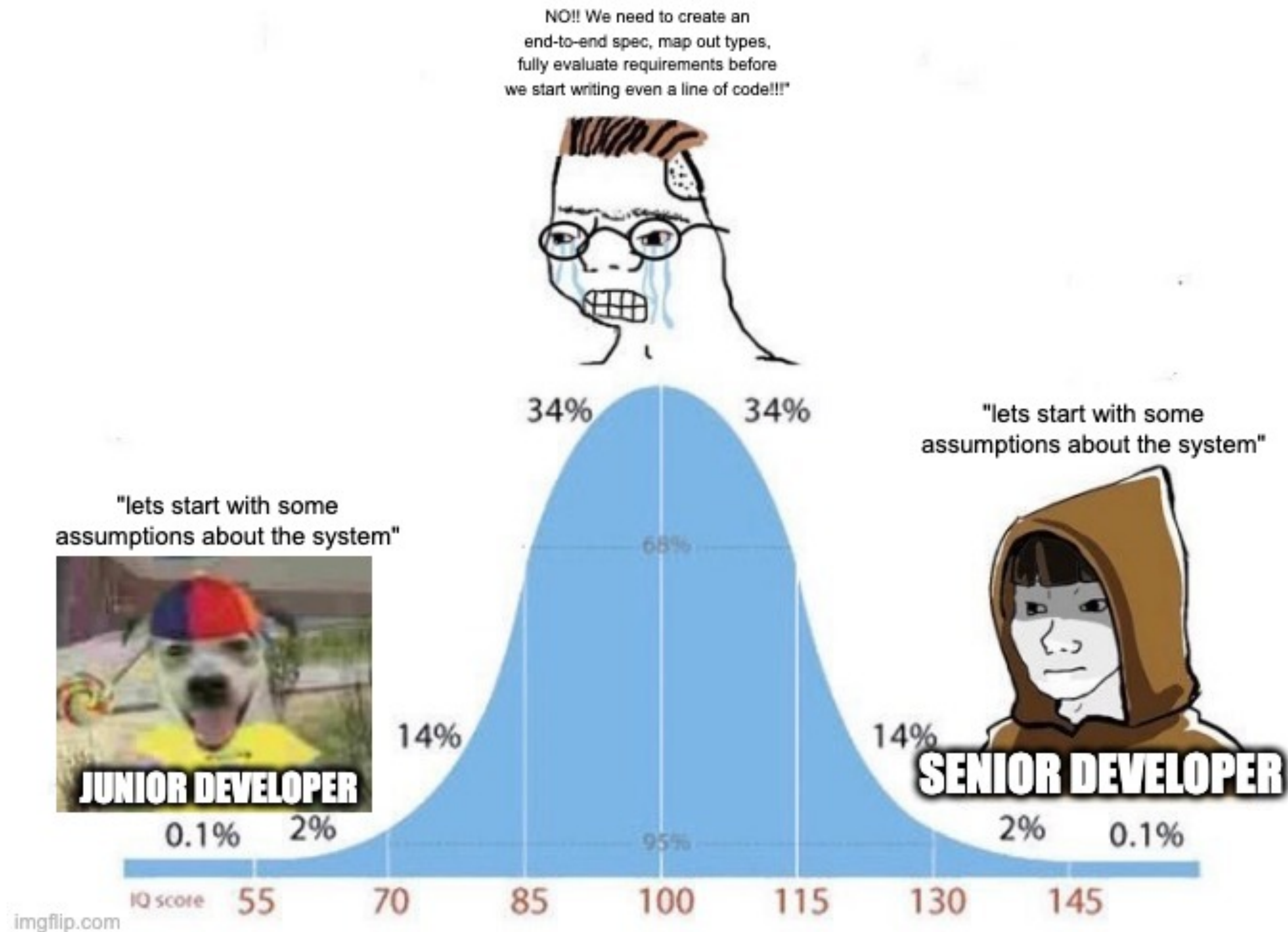
Developer on the Coretex Axiom Project.

Automation Enthusiast

MSc Machine Learning @ NUI Limerick



Let's start with some assumptions on our imagined system



Our imagined system

- We have a vector database, with parsed and embedded content in OpenSearch
- We have some locally hosted LLMs, capable of single shot generation (but no existing conversational persistence)
- We have an index for creating, storing, and retrieving our single shot generations, and being able to formulate them into a “conversational” format (OpenSearch Memory Index/API)
- We have a frontend where a user can send a query, and get a response from a Chatbot
- Optional, but nice to have: unlimited GPU power, and perfect data.

The Problem

“What did we discuss about the API design?”

“Sorry, I don’t have access to previous conversations”

After 100+ exchanges: 30–50% performance degradation

Users repeat context every 3–5 messages

Critical Information lost between sessions

Why this matters now

Business Reality	Technical Challenges
78% of organizations reporting to use AI*	Context Windows filling up
Local deployments for privacy	No persistent memory
GDPR, HIPPA compliance	Expensive redundant searches

**Source: McKinsey & Company, The State of AI Report (2025)*

Context Loss = Productivity Lost = Money Lost

What we'll discuss today

User Query → Semantic History Retrieval → Query Rewriting



Hybrid Search → Context Integration → Response Generation

Outcomes

- Fewer retrieval failures
- Performance Improvements
- Complete conversation persistence

Features

- Privacy-first local deployment
- Automatic context management
- Intelligent query understanding

OpenSearch Memory API Foundation

The Three Pillars

```
# 1. Create Conversation (Memory Container)
def create_conversation(self, conversation_name: str) -> str:
    response = self.OPENSEARCH_CLIENT.request(
        url="_plugins/_ml/memory",
        method="POST",
        payload={"name": conversation_name},
    )
    return response["memory_id"]
```

Implementation Pattern

- Always store query AND context used
- Track source documents for transparency
- Maintain completion status

OpenSearch Memory API Foundation

Message Storage with Rich Metadata

```
def create_message(self, conversation_id: str, prompt: str) -> str:
    payload = {
        "input": prompt,
        "additional_info": {
            "is_error": False,
            "prompt_date": now().isoformat(),
            "sources": [], # Enriched after retrieval
            "search_strategy": None,
            "token_count": len(encoder.encode(prompt))
        }
    }
    response = self.OPENSEARCH_CLIENT.request(
        url=f"_plugins/_ml/memory/{conversation_id}/messages",
        method="POST",
        payload=payload
    )
    return response["message_id"]
```



Create message FIRST, update after generation

OpenSearch Memory API Foundation

Three pillars together

```

# 1. Create Conversation (Memory Container)
def create_conversation(self, conversation_name: str) -> str:
    response = self.OPENSEARCH_CLIENT.request(
        url="_plugins/_ml/memory",
        method="POST",
        payload={"name": conversation_name},
    )
    return response["memory_id"]

# 2. Store Messages with Rich Metadata
def create_message(self, conversation_id: str, prompt: str) -> str:
    payload = {
        "input": prompt,
        "additional_info": {
            "is_error": False,
            "prompt_date": now().isoformat(),
            "sources": [], # Will be enriched later
        }
    }

# 3. Update with Context & Sources
def update_message(self, message_id: str, response: str, sources: list):
    # Enrich with retrieved documents, scores, timestamps

```

Implementation Pattern

1. Create our conversation.
2. Create our current turn message, enrich with existing metadata.
3. After generation, update our message with new metadata, sources, responses to provide better context for the next turn.

Data Structure Design

Metadata from our retrieved documents

```
class ChatSource(TypedDict):  
    text: str | None  
    file_name: str  
    file_id: str  
    score: float # Relevance  
    page_index: int  
    modified_at: str
```

Metadata from our retrieved conversation

```
class ConversationContext:  
    current_query: str  
    prev_user_turn: str  
    prev_assistant_turn: str  
    relevant_history: str  
    messages: list[Message]
```

✅ **DO:** Store retrieval scores and timestamps

❌ **DON'T:** Store raw embeddings in message metadata

Semantic History Retrieval

```
def get_semantically_relevant_history(
    self,
    query_embedding: list[float],
    recent_pairs: ConversationTurns,
    similarity_top_k: int = 4,
    max_tokens: int = 256
) -> str:
    # Step 1: Generate embeddings for recent history
    hist_embeddings = self._get_history_embeddings(
        recent_pairs[-50:] # Limit to recent 50 turns
    )

    # Step 2: Calculate cosine similarity
    sims = cosine_similarity(query_embedding, hist_embeddings)


    # Step 3: Select top-k relevant messages
    top_indices = sims.argsort()[-similarity_top_k:]

    # Step 4: Token management - critical!
    if token_count > max_tokens:
        return self._summarize(selected_texts)

    return chronological_texts
```

Pattern – Progressive Context Filtering

1. Try full history (if it fits in token limits) ↓
2. Fall back to semantic filtering ↓
3. Summarize if still too large



```
if len(encoder.encode(full_history)) <= max_tokens:  
    return full_history  
elif semantic_results := filter_by_similarity(history):  
    return semantic_results  
else:  
    return aggressive_summary(history)
```


Dynamic Query Rewriting

Before/After Comparison

✗ Before	✓ After
“Show me more”	“Show more Python async examples”
“What about performance?”	“OpenSearch query performance”
“Can you elaborate?”	“Elaborate on embedding latency”

Transform Questions Using Context

```
if history.get("relevant_history"):
    rewritten = llm_service.rewrite_query(
        conv_ctx={k: str(v) for k, v in history.items()}
    )
    return rewritten # Context-aware query
```


Hybrid Search Architecture

```
def enhanced_document_search(self, query: str, k_nn: int = 5):  
    # Step 1: Extract filename references  
    filename_refs = self._extract_filename_references(query)  
  
    # Step 2: Cascading search strategy  
    if filename_refs:  
        results = self._search_by_filename(filename, query_embedding)  
        if results:  
            return results  
  
    # Step 3: Hybrid search fallback  
    return self.keyword_hybrid_search(query, k_nn, query_embedding)
```

Implementation Pattern

Cascading Strategy:
Specific ↓
Hybrid ↓
General



**Don't waste valuable context from a users query
when deciding what search to perform!**

Hybrid Query Structure

```
hybrid_query = {  
  "queries": [  
    # 1. Lexical search (BM25)  
    {"match": {  
      "content": {"query": query, "fuzziness": "AUTO"}  
    }},  
  
    # 2. Phrase matching (higher weight)  
    {"match_phrase": {  
      "content": {"query": query, "boost": 1.5}  
    }},  
  
    # 3. Vector similarity  
    {"knn": {  
      "embedding": {"vector": query_embedding, "k": k_nn}  
    }}  
  ]  
}
```

Performance Tips:

- Filename boost (users remember filenames)
- Phrase matching boost
- Keep k_nn between 5–10

Token Management Strategies

```
class TokenManager:
    MAX_REWRITE_TOKENS = 256      # For context
    MAX_SEARCH_TOKENS = 512       # For retrieval
    MAX_RESPONSE_TOKENS = 2048   # For generation

    def manage_context_window(self, content: str, limit: int):
        token_count = len(self.encoder.encode(content))

        if token_count <= limit:
            return content

        # Progressive degradation
        strategies = [
            self._remove_examples,
            self._summarize_middle,
            self._keep_recent_only,
            self._aggressive_summary
        ]

        for strategy in strategies:
            content = strategy(content)
            if len(self.encoder.encode(content)) <= limit:
                break

        return content
```

Implementation Pattern

- Set customizable limits on **all** your generations
- Like our search & history context implement progressive fallbacks in order to stay within our limits.
- Experiment with different degradation strategies

Three Layer Cache Strategy

```
# Layer 1: Embedding Cache (Most Valuable)
@lru_cache(maxsize=10000)
def get_embedding(text_hash: str) -> list[float]:
    return generate_embedding(text)

# Layer 2: Search Result Cache (24hr TTL)
search_cache = {
    "query_hash": {"results": [...], "timestamp": now(), "ttl": 86400}
}

# Layer 3: Conversation Summary Cache
summary_cache[conv_id] = {
    "100_turn_summary": "...",
    "50_turn_summary": "...",
    "25_turn_summary": "..."
}
```

Caching Layers Overview

- **Layer 1 – Embedding Cache:** Fastest + most valuable; stores embeddings in memory (@lru_cache) to avoid recomputing.
- **Layer 2 – Search Result Cache:** Keeps query results for 24h (TTL = 86400s) to save redundant searches.
- **Layer 3 – Conversation Summary Cache:** Stores rolling summaries (100/50/25 turns) to quickly recall context without reprocessing full history.

Putting it all together

Implementation Pattern

```
async def process_turn(self, user_input: str):
    # 1. Always create message first (track everything)
    message_id = await self.memory_api.create_message(user_input)

    try:
        # 2. Get context with timeout
        context = await asyncio.wait_for(
            self.get_context(), timeout=2.0
        )

        # 3. Search with fallback
        results = await self.search_with_fallback(user_input, context)

        # 4. Generate and stream
        async for chunk in self.generate_response(results):
            yield chunk

    finally:
        # 5. Always update message (even on error)
        await self.memory_api.update_message(
            message_id, status="complete" if success else "error"
        )
```

Common Pitfalls to avoid

Not Handling overflow

```
• • •  
# ❌ BAD  
messages = get_all_messages() # OOM on long conversations  
  
# ✅ GOOD  
messages = get_recent_messages(limit=100)  
if needs_more_context:  
    summary = get_conversation_summary()
```

Ignoring search failures

```
• • •  
# ❌ BAD  
results = search(query)  
context = results[0] # KeyError when no results  
  
# ✅ GOOD  
results = search(query) or []  
context = results[0] if results else self.get_fallback_context()
```


Collecting metrics, measuring impact

```
class MetricsCollector:
    def track_performance(self):
        return {
            # Quality Metrics
            "retrieval_precision": self.relevant_docs / self.total_retrieved,
            "context_relevance": self.cosine_sim(query, context),

            # Performance Metrics
            "avg_response_time": statistics.mean(self.response_times),
            "cache_hit_rate": self.cache_hits / self.total_requests,

            # Business Metrics
            "searches_saved": self.answered_from_memory / self.total_queries,
            "conversation_continuity": self.avg_turns_before_reset
        }
```


Three Critical Success Factors

Store Everything, Retrieve Selectively	Hybrid Search > Pure Vector Search	Cache Aggressively, Fail Gracefully
Every turn creates a message	Users remember keywords and filenames	Embeddings are expensive. Cache them.
Semantic filtering finds relevance	Combine BM25 + vectors + metadata	Searches timeout – have fallbacks
Progressive summarization manages scale	Cascade from specific to general	Conversations overflow, summarize early

Q&A

Thank You!



info@eliatra.com



eliatra



eliatra_ire

lucas.jeanniot@eliatra.com

www.linkedin.com/in/lucas-jeanniot/

