# AI in Society and Public Services

## Session 5: RAG vs. CAG

Mário Antunes

January 30, 2026

Universidade de Aveiro

# Part I: Limitations

# The Problem: The “Frozen” Brain

- LLMs (like GPT-4, Llama 3, Qwen) are **static**.
- They only know what they learned during training.
- **The Knowledge Cut-off:**
- A model trained in 2023 knows nothing of the 2024 Tax Reform Act.
- A model trained on the open internet knows nothing of *your* internal confidential memos.

# Why Not Just Fine-Tune?

- **Common Misconception:** “Let’s feed our documents into the model and re-train it.”
- **Reality:** Fine-tuning is primarily for *behavior* and *format*, not for *knowledge*.
- **The Risks:**

1. **Catastrophic Forgetting:** The model learns new data but forgets basic language skills.
2. **Hallucination:** It might mix internal facts with public training data.
3. **Cost & Slowness:** Retraining every time a policy changes (daily/weekly) is financially impossible.

# Part II: Solutions

# The Solution Space

If we can't change the model's brain (weights), we must change the **input**.

We must provide the relevant facts *inside the prompt* at the moment of asking.

- **Approach 1: RAG (Retrieval-Augmented Generation)**
- *Analogy:* You go to a library, search the index, find 3 specific pages, and read only those.
- **Approach 2: CAG (Context-Augmented Generation)**
- *Analogy:* You put the entire stack of documents on your desk and read everything before answering.

# Part III: RAG

# Deep Dive: RAG (Retrieval-Augmented Generation)

- **Definition:** A system that retrieves relevant document chunks from a database and feeds them to the LLM.
- **Mechanism:** It connects a Generative AI to a Retrieval System (Search Engine).
- **The “Standard” Standard:** Currently the most common way to build enterprise AI.

# How RAG Works: Step 1 - Ingestion

Before anyone asks a question:

1. **Chunking:** Break large government PDFs into smaller pieces (e.g., 500 words).
2. **Embedding:** Convert these text chunks into **Vectors** (lists of numbers representing meaning).
3. **Storage:** Save these vectors in a **Vector Database** (e.g., Chroma, Milvus, pgvector).

# How RAG Works: Step 2 - Retrieval

When a user asks: “What is the penalty for late housing tax?”

1. The question is converted into a vector.
2. The database performs a **Semantic Search** (finding vectors that are mathematically close).
3. The system retrieves the top 3-5 matching chunks of text.

# How RAG Works: Step 3 - Generation

The system constructs a prompt:

> *"Use the following context to answer the question. Context: [Chunk 1: Housing Tax Law Section 4] Context: [Chunk 2: Late Fees Amendment] Question: What is the penalty for late housing tax?"*

The LLM answers using *only* that provided text.

# RAG Use Case: The National Archive

- **Scenario:** A Ministry has 50 years of digitized records (Millions of PDFs).
- **Why RAG?**
- The dataset is **too large** to fit in any context window.
- You only need specific facts (e.g., “Find the 1998 meeting minutes regarding water sanitation”).
- **Implementation:**
- **User:** “What was the decision on the dam project?”
- **RAG:** Searches 1M docs -> Finds 2 relevant pages -> LLM Summarizes.

# Part IV: CAG

# Deep Dive: CAG (Context-Augmented Generation)

- **Definition:** Leveraging massive Context Windows (128k, 1M, to 2M+ tokens) to load *entire* datasets into the model's working memory.
- **The Shift:** Newer models (Gemini 1.5, Claude 3) can "read" hundreds of books in a single prompt.
- **KV Caching:** To make this fast, we "cache" the processed state of the documents so we don't re-read them every time.

# How CAG Works

1. **No Vector Database:** You do not chop up documents.
2. **Pre-loading:** You take the entire relevant dataset (e.g., a specific citizen's entire case history).
3. **The Prompt:** > "Here are all 500 emails, forms, and notes regarding Case ##999. [Insert Full Text]. > Based on this history, what is the current status?"
4. **Processing:** The model attends to the *whole* global context simultaneously.

# The Concept of “Needle in a Haystack”

- **Challenge:** Can the model find one specific sentence hidden in 500 pages of text?
- **RAG Approach:** Relies on the *search engine* finding the chunk first. If search fails, the answer fails.
- **CAG Approach:** Relies on the *LLM* reading everything. Modern models have >99% recall in long context.
- **Benefit:** CAG captures “Global” questions better (e.g., “What is the overall tone of these meetings?”). RAG struggles with summaries of entire datasets.

# CAG Use Case: Active Judicial Proceedings

- **Scenario:** A judge needs to understand a specific active case file (Witness statements, police reports, evidence logs). Total: 400 pages.
- **Why CAG?**
- The data is “Bounded” (it’s not infinite, just large).
- Relationships matter: Statement A contradicts Statement B on page 300. RAG might miss the contradiction if it only retrieves Statement A.
- **Implementation:** Load full case file -> Ask complex reasoning questions.

# Part V: Deployment

# Deployment Guidelines: RAG

**When to deploy RAG:**

- **Data Volume:** Massive (GBs to TBs).
- **Volatility:** Data changes frequently (easy to update one chunk in a DB).
- **Cost:** You want to pay only for the tokens you generate, not reading the whole library every time.

**The Stack:**

- **LLM:** Can be small/standard (Llama 3 8B, Qwen 7B).
- **Database:** Vector Store (Qdrant, Elasticsearch).
- **Middleware:** LangChain or LlamaIndex.

# Deployment Guidelines: CAG

**When to deploy CAG:**

- **Data Volume:** Moderate (Fits within 128k - 1M tokens).
- **Complexity:** Requires connecting dots across the whole document.
- **Latency:** You can tolerate a longer “Time to First Token” (unless using KV Caching).

**The Stack:**

- **LLM:** Must be a “Long Context” model (Gemini 1.5 Pro, Claude 3 Opus, or specialized open weights like Command R+).
- **Hardware:** Massive VRAM (Video RAM) required if running locally to store the context.

# Comparison: Retrieval Accuracy

| Feature | RAG | CAG |
|---|---|---|
| **Precision** | **Variable.** Depends on the search algorithm (BM25/Cosine). | **High.** The model sees everything. |
| **Blind Spots** | **Yes.** If the chunk isn't retrieved, the model doesn't know it exists. | **No.** Everything is in view. |
| **Global Summary** | **Poor.** Can't summarize what it hasn't retrieved. | **Excellent.** Can synthesize the whole text. |

# Comparison: Performance & Latency

| Feature | RAG | CAG |
| --- | --- | --- |
| **Initial Load** | Fast (Just text ingestion). | Slow (Processing 1M tokens takes time). |
| **Query Speed** | **Fast.** Process only ~1k tokens per query. | **Variable.** Fast with Cache, Slow without. |
| **Throughput** | High concurrency possible. | Heavy memory usage limits concurrency. |

# Comparison: Cost (API & Compute)

| Feature | RAG | CAG |
| --- | --- | --- |
| **Token Costs** | **Low.** Input is small (Question + 3 chunks). | **High.** Input is massive (Entire documents). |
| **Maintenance** | **High.** Managing Vector DBs, re-indexing, clean-up. | **Low.** Just load the text files. |

# The Hybrid Approach (The “Sweet Spot”)

- **Scenario:** Public Services often need both.
- **Workflow:**

1. **Step 1 (Broad Search):** Use **RAG** to find the relevant *folder* or *law* from the massive archive.
2. **Step 2 (Deep Read):** Use **CAG** to load that specific 100-page law into context for detailed reasoning.

- **Result:** Efficiency of search + Comprehensiveness of context.

# Security & Privacy in RAG vs CAG

- **RAG Risks:**
- “Poisoning” the vector database.
- Access Control: Ensuring User A doesn’t retrieve User B’s chunks. (Requires Row-Level Security in Vector DB).
- **CAG Risks:**
- **Prompt Injection:** With so much context, malicious instructions hidden in a document might be executed.
- **Data Leakage:** Loading sensitive PII into the prompt context (RAM).

# Deployment Checklist: Public Sector

1. **Data Classification:** Is the data Top Secret? (Air-gapped Local RAG). Is it Public Info? (Cloud CAG).
2. **Infrastructure:**
   - *RAG:* CPU-heavy for indexing, Storage-heavy for vectors.
   - *CAG:* GPU-heavy (VRAM) for context window.
3. **Auditability:** RAG is easier to audit (we know exactly which chunk was cited).

# Implementing RAG: A Practical Snippet

*Conceptual Python/LangChain code:*

```
## RAG: Index and Search
vectorstore = Chroma.from_documents(documents, embedding_model)
retriever = vectorstore.as_retriever()
## Only fetches the top 4 relevant snippets
relevant_docs = retriever.get_relevant_documents("What is the re
chain.run(input_documents=relevant_docs, question="...")
```

# Implementing CAG: A Practical Snippet

*Conceptual Code:*

```
## CAG: Dump it all in
with open("massive_policy.txt") as f:
    full_context = f.read()

## No retrieval step. Just generation.
response = client.chat.completions.create(
    model="long-context-model",
    messages=[
        {"role": "system", "content": "You are an expert analyst
        {"role": "user", "content": f"Context: {full_context} \n
    ]
)
```

# Part VI: RAG/CAG Limitations

# The “Lost in the Middle” Phenomenon

- **Warning for CAG:**
- Even with large windows, models pay most attention to the **Beginning** and the **End** of the prompt.
- Information in the middle can be ignored.
- **Mitigation:**
- Put the most critical instructions at the very end.
- Use RAG to re-rank information before creating the context.

# Cost Analysis for Government Budgets

- **RAG:** Higher Upfront Engineering Cost (Building the pipeline), Lower Operational Cost (Per query).
- **CAG:** Zero Engineering Cost (Just dump text), High Operational Cost (Per query).
- **Decision:**
- High frequency queries (Chatbot for thousands of citizens) -> **RAG**.
- High value, low frequency queries (Analyst reviewing a merger) -> **CAG**.

# Deployment Guidelines: The “Ollama” Context

- If running **Locally** (as per previous lecture):
- **RAG is preferred.** Most local hardware cannot handle 100k+ token context windows effectively without crashing.
- **CAG on Edge:** Only possible for small documents (e.g., summarizing one specific letter).
- **Hardware Requirement:**
- To run a 128k context model locally, you likely need dual A100s or equivalent, rarely found in standard government laptops.

# Future Trends: “Prompt Caching”

- **The CAG Game Changer:**
- API providers (and local engines) are introducing “Prompt Caching”.
- You send the 500-page manual *once*. It is stored in cache.
- Subsequent questions are cheap and fast.
- **Impact:** This makes CAG much more competitive with RAG for static documents.

# Summary Comparison Table

| Criteria | RAG | CAG |
|---|---|---|
| **Best For** | Massive Knowledge Bases (Wikipedia size) | Specific Heavy Documents (Book size) |
| **Setup** | Complex (Vector DB, Chunking strategies) | Simple (Prompt Engineering) |
| **Reasoning** | Fragmented (sees chunks) | Holistic (sees connections) |
| **Hallucination** | Low (grounded in chunks) | Low (grounded in context) |
| **Hardware** | Storage Intensive | VRAM/Compute Intensive |

# Final Recommendation

**For Public Services:**

1. Build a **RAG** foundation for your “Enterprise Knowledge Base” (Laws, Regulations, FAQs).
2. Build a **CAG** workflow for “Task-Specific Analysis” (Reviewing a specific application or contract).
3. Do **not** fine-tune models for knowledge.

# Q&A