Quantized RAG Systems for Edge Deployment: Architecture, Optimization, and Multimodal Extension

Zahraa Selim, Menna Hamed, Wesam Ahmed,
Sohyla Said, Sara Basheer, Rami Zewail

Computer Science and Engineering Department
Egypt-Japan University of Science and Technology (E-JUST)
Alexandria, Egypt
`{zahraa.selim, menna.hamed, rami.zewail}@ejust.edu.eg`

December 10, 2025

1 Methodology

1.1 Overview

This study presents a comprehensive Retrieval-Augmented Generation (RAG) system optimized for deployment on consumer-grade hardware using 4-bit quantized Large Language Models. Our modular architecture addresses the compound resource constraints of edge deployment: concurrent allocation for vector indices, embedding models, dynamic Key-Value caches, and the generation model itself. All experiments are conducted on NVIDIA Tesla T4 (16GB VRAM) to simulate realistic edge deployment scenarios.

The RAG pipeline consists of seven sequential modules: (1) Ingestion for document extraction, (2) Processing for content structuring, (3) Chunking for optimal segmentation, (4) Embedding for vector representation, (5) Indexing for fast retrieval, (6) Retrieval for relevant context selection, and (7) Generation for answer synthesis. We evaluate the complete system under quantized model constraints, measuring both efficiency and faithfulness metrics.

1.2 Hardware and Software Environment

Hardware Configuration:

- **Platform:** NVIDIA Tesla T4 GPU (16GB VRAM, Turing architecture)
- **TDP:** 70W (for energy consumption calculations)
- **System RAM:** 16GB DDR4
- **Storage:** NVMe SSD for index persistence

Software Stack:

- **Framework:** PyTorch 2.x with CUDA 11.8
- **Quantization:** BitsAndBytes (NF4 4-bit), AutoAWQ
- **Embeddings:** sentence-transformers (BGE-small-en-v1.5)

- **Vector Store:** FAISS (IndexHNSWFlat)
- **LLM Inference:** llama-cpp-python (GGUF format)
- **Language Model:** Mistral-7B-Instruct-v0.2 (4-bit quantized)

1.3 Base Models

1.3.1 Generation Model: Mistral-7B-Instruct-v0.2

We select Mistral-7B-Instruct-v0.2 [?] for answer generation due to its strong instruction-following capabilities and efficient architecture:

- **Parameters:** 7.24 billion
- **Architecture:** Decoder-only transformer with Grouped Query Attention (GQA, 8 KV heads) and Sliding Window Attention (4096-token window)
- **Context Window:** 32k tokens (using 4k for memory safety)
- **Quantization:** 4-bit NF4 via BitsAndBytes
- **Model Size:** 7.6 GB (quantized) vs. 14 GB (FP16)
- **Format:** GGUF Q4_K_M for llama-cpp-python inference

Quantization Configuration:

- **Method:** 4-bit NormalFloat (NF4) following QLoRA protocol [?]
- **Double Quantization:** Enabled (quantizes quantization constants using FP8)
- **Compute Dtype:** FP16 for dequantization operations
- **Block Size:** 64 (default BitsAndBytes configuration)

Generation Parameters:

- **Max New Tokens:** 128 (concise answers)
- **Temperature:** 0.3 (low for factuality)
- **Top-p:** 0.9 (nucleus sampling)
- **Repetition Penalty:** 1.15
- **Do Sample:** False (deterministic decoding)

1.3.2 Embedding Model: BGE-small-en-v1.5

For dense semantic embeddings, we employ BAAI/bge-small-en-v1.5 [?], a compact sentence transformer optimized for retrieval:

- **Dimensions:** 384 (compact yet expressive)
- **Model Size:** 133 MB on disk, 300 MB VRAM when loaded
- **Max Sequence Length:** 512 tokens
- **Training:** 1B+ sentence pairs from diverse domains
- **Performance:** SBERT benchmark score: 68.06
- **Inference Speed:** 50ms/chunk on GPU, 200ms/chunk on CPU
- **Normalization:** L2-normalized outputs for cosine similarity via dot product

Content-Type Prefixes:

To improve retrieval accuracy, we apply type-specific prefixes [?]:

- Math equations: `"equation: {content}"`

- Code blocks: `"code: {content}"`
- Tables: `"table: {content}"`
- Plain text: `"passage: {content}"`
- Headings: `"title: {content}"`

1.4 RAG Pipeline Architecture

Our modular RAG system consists of seven sequential stages, each optimized for edge deployment constraints:

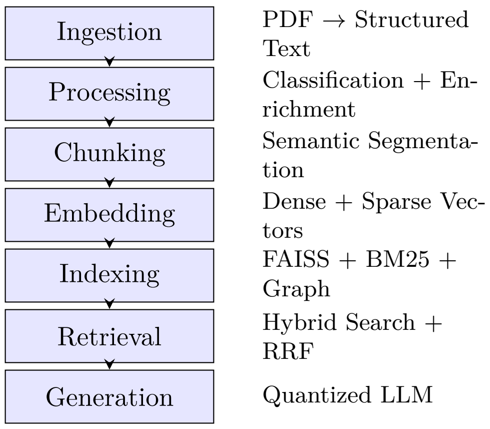

Figure 1: Modular RAG pipeline architecture for edge deployment

1.4.1 Module 1: Ingestion

The ingestion module transforms raw PDF documents into structured, searchable content using intelligent routing to optimize for both speed and quality.

Routing Strategy:

We employ a hybrid extraction approach that analyzes PDF characteristics to select optimal parsers:

- **Native Extraction (70-80% of pages):** For clean PDFs with text layers using PyMuPDF (0.1-0.3s/page, 100MB RAM)
- **Layout-Aware Extraction (15-20%):** For complex multi-column layouts using Marker with Surya layout detection (2-4s/page, 600MB RAM)
- **OCR-Based Extraction (5-10%):** Three-stage cascade (Tesseract → EasyOCR → PaddleOCR) for scanned documents (5-15s/page)

Quality Validation:

Each extraction is validated using coherence metrics:

$$\text{Coherence} = \frac{\text{Valid\_Characters}}{\text{Total\_Characters}} \times \frac{\text{Sentence\_Count}}{\text{Expected\_Sentences}} \tag{1}$$

Extractions scoring below 0.5 are retried with stronger methods (maximum 3 attempts).

Specialized Extractors:

- **Equations:** Multi-method LaTeX extraction (pattern matching, embedded LaTeX, optional Pix2Tex) with SymPy enrichment for variable extraction and natural language descriptions

- **Code Blocks:** Detection via markdown syntax, indentation patterns, and keyword density with language identification for 12+ languages
- **Tables:** Extraction with pdfplumber preserving structure, header detection, and cell alignment
- **Images:** Extraction with bounding boxes, caption detection using proximity heuristics, optional OCR for text-containing images

1.4.2 Module 2: Processing

The processing module transforms extracted content into semantically rich, type-classified blocks.

Content Classification:

Heuristic-based classification (1-3ms/block) assigns types without ML models:

- **Math:** LaTeX delimiters, mathematical operators ($\int$, $\sum$,)
- **Code:** Function definitions, imports, keywords (`def, class, function`)
- **Table:** Pipe separators, column alignment patterns
- **List:** Bullet points, numbered items
- **Heading:** Short capitalized lines, section numbers, markdown headers
- **Text:** Default for standard prose

Each type receives a confidence score (0-1), with classification threshold at 0.6.

Hierarchy Building:

Document structure is detected via:

- Markdown headers (`#, ##, ###`)
- Numbered sections (1., 1.1, 1.1.1)
- Chapter patterns (Chapter 1, CHAPTER I)
- Implicit capitalized headings

The resulting tree structure enables section-aware retrieval and hierarchical chunking.

Relationship Mapping:

Semantic links are established between blocks:

- **Adjacent:** Sequential blocks (confidence: 1.0)
- **References:** Explicit mentions (Figure N, Table N, Equation N) (confidence: 0.8)
- **Explained_by:** Math equations with nearby explanatory text (confidence: 0.7)
- **Described_by:** Code blocks with surrounding descriptions (confidence: 0.7)

Type-Specific Enrichment:

- **Math:** SymPy normalization, variable extraction, complexity indicators (calculus, algebra, matrices), natural language descriptions
- **Code:** Language detection, function/class extraction, import analysis, structure summary
- **Tables:** Row/column counting, header detection, type classification (comparison, financial, performance), summary generation
- **Text:** Definition extraction, acronym detection, key term identification (top 10 capitalized phrases), word/sentence counting

Normalization:

Text cleaning removes PDF artifacts:

- Whitespace collapse and hyphenation rejoining
- Quote standardization (curly $\rightarrow$ straight)
- Header/footer removal (heuristic: short lines with "page"/numbers)
- Unit expansion (km $\rightarrow$ kilometers)
- Scientific notation conversion ($3.5 \times 10^2 \rightarrow 350.0$)

1.4.3 Module 3: Chunking

The chunking module creates optimized segments preserving semantic coherence and special content integrity.

Fixed-Smart Chunking (Default):
Our primary strategy targets 512 tokens (min 256, max 768) with 50-token overlap:

1. Accumulate blocks until target size reached
2. Check if next block exceeds maximum
3. Finalize chunk at smart boundary (paragraph > sentence > clause)
4. Add 50-token overlap from previous chunk
5. Continue with remaining content

Boundary Detection:
Content-aware boundary detector identifies safe split points:

- Section breaks and paragraph boundaries (highest priority)
- Sentence endings (. ! ?)
- Clause boundaries (, ; :)
- Before/after special content blocks

Never splits:

- Inside equations (`$...$, $$...$$`)
- Inside code blocks (`"‘..."‘`)
- Inside tables (`|...|`)
- Mid-sentence unless forced by size constraints

Alternative Strategies:

- **Semantic Chunking:** Embeds sentences, splits at < 0.7 cosine similarity drops (100-200ms/page, requires embedding model)
- **Hierarchical Chunking:** Aligns chunks to sections/subsections from document hierarchy (20-40ms/page, requires clear structure)

Chunk Enrichment:
Each chunk receives metadata:

- **Adjacent IDs:** Links to previous/next chunks for context expansion
- **Key Terms:** Pattern-matched capitalized phrases (top 10)
- **Summary:** First sentence or first 150 characters
- **Section Path:** Hierarchical path (Chapter > Section > Subsection)

- **Content Distribution:** Percentage breakdown by type (text, math, code, table)

Validation:
All chunks pass quality checks:

- Size bounds (256-768 tokens)
- Complete sentences (no mid-sentence cuts)
- Balanced delimiters (matched brackets, quotes)
- Semantic coherence (not overly fragmented)

Invalid chunks are automatically corrected through merging (too small) or splitting at better boundaries (too large).

1.4.4 Module 4: Embedding

The embedding module generates dual representations: dense semantic vectors and sparse keyword vectors.

Dense Embedding (BGE-small-en-v1.5):
Neural embeddings capture semantic similarity:

Listing 1: Dense embedding generation

```python
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('BAAI/bge-small-en-v1.5', device='cuda')

# Apply content-type prefixes
prefixed_text = f"{content_prefix}: {chunk_content}"

# Generate and normalize embeddings
embeddings = model.encode(
    prefixed_text,
    batch_size=64,  # GPU-optimized
    normalize_embeddings=True  # L2 normalization
)
```

Batching Strategy:

- GPU: batch_size=64 (optimal throughput)
- CPU: batch_size=16 (memory-constrained)
- Dynamic adjustment on OOM errors
- Length-based sorting for efficient padding

Sparse Embedding (BM25):
Statistical keyword matching using Okapi BM25 [?]:

$$\text{score}(D, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})} \tag{2}$$

where $f(q_i, D)$ is term frequency, $|D|$ is document length, avgdl is average document length, $k_1 = 1.5$ (term frequency saturation), and $b = 0.75$ (length normalization).

Tokenization:

- Lowercase conversion
- Regex-based word splitting
- Stopword removal (English stopwords)

- Minimum token length: 3 characters

Two-Tier Caching:
To avoid redundant computation:

- **L1 Memory Cache:** LRU cache (10,000 embeddings, <1ms access)
- **L2 Disk Cache:** SQLite database (unlimited capacity, 5-10ms access)
- **Cache Key:** SHA-256(model_name + content)
- **Hit Rate:** 60-80% for similar documents
- **Cleanup:** Automatic removal of entries >30 days old

Fallback Mechanism:
If BGE model fails:

1. Attempt CPU fallback
2. If still fails, use TF-IDF vectorization (384 dimensions via feature selection)
3. Lower quality but ensures system never fails

1.4.5 Module 5: Indexing

The indexing module builds multi-store architecture for fast, accurate retrieval.
Vector Store (FAISS IndexHNSWFlat):
Hierarchical Navigable Small World (HNSW) index [?] for approximate nearest neighbor search:

- **Index Type:** IndexHNSWFlat (no quantization, maximum accuracy)
- **M:** 32 connections per layer (memory vs. accuracy trade-off)
- **efConstruction:** 40 (build-time accuracy)
- **efSearch:** 16 (query-time accuracy, tunable)
- **Distance Metric:** Cosine similarity (via dot product on normalized vectors)
- **Build Time:** 30 seconds for 100k vectors
- **Query Time:** <50ms for k=10 in 100k vectors
- **Memory:** 2KB per vector (384-dim)

Keyword Store (BM25):
Pure statistical ranking:

- **Implementation:** rank-bm25 library
- **Index Build:** 1 second per 10k documents
- **Query Time:** <10ms regardless of corpus size
- **Memory:** 2KB per document
- **Score Normalization:** Divided by max score for [0,1] range

Metadata Store (SQLite + FTS5):
Relational database with full-text search:

Listing 2: Metadata store schema

```sql
CREATE TABLE chunks (
    id INTEGER PRIMARY KEY,
    doc_id TEXT NOT NULL,
    content TEXT NOT NULL,
    content_type TEXT,  -- text/math/code/table
    page_num INTEGER,
    section_path TEXT,
    has_math BOOLEAN,
    has_code BOOLEAN,
    token_count INTEGER,
    metadata TEXT  -- JSON blob
);

CREATE VIRTUAL TABLE chunks_fts USING fts5(
    content,
    content=chunks,
    content_rowid=id
);

CREATE INDEX idx_chunks_type ON chunks(content_type);
```

Graph Store (NetworkX):
Relationship graph for context expansion:

- **Nodes:** Chunks with metadata
- **Edges:** Directed relationships (adjacent, explains, references) with weights
- **Traversal:** BFS for 2-3 hop expansion
- **Storage:** Pickle serialization (50MB for 100k chunks)

Index Optimization:
Automatic tuning for production deployment:

- **FAISS efSearch Tuning:** Target 95% recall with minimal latency
- **SQLite Optimization:** ANALYZE, VACUUM, PRAGMA optimize
- **Validation:** Test queries measure latency improvements

1.4.6 Module 6: Retrieval

The retrieval module implements hybrid search combining semantic and lexical matching.
Hybrid Retrieval Strategy:
Reciprocal Rank Fusion (RRF) [?] combines dense and sparse results:

$$\text{RRF\_score}(d) = \sum_{r \in \{dense, sparse\}} \frac{1}{k + rank_r(d)} \tag{3}$$

where $k = 60$ (default RRF constant) and $rank_r(d)$ is the rank of document d in ranking r.
Retrieval Pipeline:

1. **Dense Search:** Embed query, retrieve top-20 from FAISS (50ms)
2. **Sparse Search:** Tokenize query, retrieve top-20 from BM25 (10ms)
3. **RRF Fusion:** Combine rankings with RRF scoring (10ms)
4. **Filtering:** Apply score threshold (min_score=0.3), diversity limits (max 2/section)
5. **Graph Expansion:** Add adjacent chunks if similarity > 0.7 (optional, 50ms)
6. **Return:** Top-k final results (default k=5)

Query Analysis:
Heuristic-based query understanding without ML:

- **Content Type:** Math/code/table indicators for type-specific retrieval

- **Intent:** Definition/procedure/example/comparison patterns
- **Complexity:** Simple (< 10 words), moderate (10-20), complex (> 20)

Two-Level Caching:

- **L1:** In-memory LRU (100 queries, <1ms)
- **L2:** SQLite persistent (10k+ queries, <10ms)
- **Key:** hash(query + k + filters)
- **TTL:** 3600 seconds (1 hour)
- **Hit Rate:** 30-40% typical usage

Performance Characteristics:

- **Hybrid:** 120-270ms (embedding + search + fusion)
- **Semantic only:** 100ms
- **Keyword only:** 11ms
- **Recall@5:** 85% (hybrid), 70% (semantic), 65% (keyword)

1.4.7 Module 7: Generation

The generation module synthesizes answers from retrieved context using quantized Mistral-7B.
Prompt Construction:
Type-specific prompts optimized for Mistral-7B-Instruct format:

Listing 3: Mistral prompt template

```
<s>[INST] {system_prompt}

Context:
{retrieved_chunks_with_citations}

Question: {user_query}

{type_specific_instructions}
[/INST]
```

System Prompts by Type:

- **General:** "You are a helpful educational assistant. Answer based ONLY on provided context. Cite sources using [1], [2] format."
- **Math:** "You are a mathematics tutor. Show step-by-step work. Explain each step clearly. Use proper notation. Cite source of formulas."
- **Code:** "You are a programming expert. Explain before showing code. Use markdown code blocks. Highlight language-specific features."
- **Table:** "You are a data analyst. Present findings clearly. Analyze patterns. Use structured format. Cite specific table rows."

Context Building:
Token budget management for 4k context window:

- Total tokens: 4000
- System/prompt: 200
- Context: 2500 (retrieved chunks)

- Query: 100
- Generation space: 1200

Chunk Selection:

1. Prioritize by relevance score (highest first)
2. Prioritize by content type if query-specific
3. Truncate chunks exceeding max_chunk_tokens (512)
4. Select until budget exhausted

Answer Validation:
Five-dimensional quality assessment:

1. **Groundedness (50-70%):** Word overlap between answer and context, citation patterns
2. **Relevance (20-30%):** Query term coverage, intent matching, non-answer detection
3. **Completeness (10-20%):** Length vs. query complexity (optimal: 500-1000 words)
4. **Quality (10-15%):** Capitalization, sentence structure, vocabulary diversity
5. **Citations (5-10%):** Presence of [1], [2] markers, sufficient source references (min 1, ideal 3+)

Validation threshold: score ≥ 0.6 and issues ≤ 2 for acceptance.
Post-Processing:

- **Text Cleaning:** Remove generation artifacts ([INST], </s>), normalize whitespace
- **Equation Formatting:** Normalize LaTeX markers (\[\] → \$\$ \$\$)
- **Code Formatting:** Infer language, add syntax highlighting
- **Citation Extraction:** Map [1], [2] to chunk IDs and source metadata

Fallback Strategy:
If generation fails or produces low-quality output:

1. **Extractive Summary:** Concatenate top 3 chunks with citations
2. **Simple Answer:** Return best chunk verbatim
3. **Chunk List:** Format as numbered list for user browsing

Fallback selection based on error type (timeout → extractive, OOM → simple).

1.5 Multimodal Extension

To enable visual document understanding, we integrate Florence-2 [?] vision encoder with our quantized LLM through a "Vision-as-Language" architecture that avoids expensive multimodal training.

1.5.1 Vision Agent Architecture

Instead of projecting image embeddings into LLM hidden space, we employ textual translation:

1. **Visual Ingestion:** Convert document pages to images
2. **Dense Captioning:** Florence-2 generates detailed textual descriptions using compound prompting:
    - `<OD>` (Object Detection): Identify labeled diagram components

    - `<MORE_DETAILED_CAPTION>`: Generate paragraph-level semantic descriptions
3. **Context Fusion:** Structure descriptions as "[Image Context]: The figure shows... [Vision Agent Output]"
4. **Cross-Modal Reasoning:** Quantized LLM processes combined visual + textual context

1.5.2 Vision Model Selection

We evaluated six lightweight vision encoders on 50 tri-modal QA pairs (Image, Text, Question) stratified by complexity:

- **Text-Only (Type A):** Control questions ensuring vision doesn't degrade text performance
- **Vision-Only (Type B):** Questions requiring visual interpretation (e.g., "What color represents...")
- **Combined (Type C):** Synthesis questions requiring both modalities

Models Evaluated:

- Microsoft Florence-2 (Base & Large): Unified model trained on FLD-5B
- Moondream2 (1.86B): Edge-optimized VLM
- BLIP-Base: Baseline image captioning
- GIT-Base: Generative Image-to-Text transformer
- ViT-GPT2: Classic encoder-decoder baseline