| **Types of RAGs** | Strength, Weakness, Use Case, Relevance to Automation query on CLI Docs task |
| --- | --- |
| **1. Original RAG** | * Strengths: Well-established foundation, versatile across tasks. * Weaknesses: Computationally intensive, basic capabilities in comparison to other RAGs. * Use Case: General-purpose question answering systems or basic text summarization.   The foundational model conditions on retrieved passages per sequence; while more compute‑intensive than EfficientRAG, it remains simpler to implement and reasonably efficient on small doc corpora.   * **Simplicity**: Condition on a fixed top‑k passages (e.g., k=5) drawn from AWS CLI docs corpus. * **Proven performance**: Sets state‑of‑the‑art on open‑domain QA in EMNLP 2020 benchmarks citeturn2search0turn2search1. * **Minimal engineering**: Can be built rapidly using Hugging Face’s [RagRetriever + RagSequenceForGeneration] pipeline. |
| **2. Graph RAG** | * Strengths: Enhanced contextual understanding through entity relationships, ideal for structured data and knowledge graph completion. * Weaknesses: Complex implementation, scalability challenges with large graphs or unstructured data. * Use Case: Drug discovery by relating chemical compounds, proteins, and diseases. |
| **3. LongRAG** | * Strengths: Handles extended contexts effectively, reduced memory overhead. * Weaknesses: Potential for information overload, slower performance with longer documents. * Use Case: Generating summaries of lengthy legal or scientific documents.   Tailored to very long contexts with chunking or sliding‑window retrieval—unnecessary for concise CLI docs, and extra context handling slows down lookups. |
| **4. Self-RAG** | * Strengths: Minimal supervision required, continuous improvement through self-supervised learning. * Weaknesses: Less predictable outputs, complex training setup. * Use Case: Generating product descriptions based on large volumes of unlabeled product data.   Uses self‑supervised signals to refine retrieval over time—reduces manual labeling but unpredictably varies retrieval quality, and self‑training loops consume additional compute. |
| **5. Corrective RAG** | * Strengths: High accuracy through corrective mechanisms, learns from mistakes. * Weaknesses: Increased model complexity, potential performance overhead. * Use Case: Medical diagnosis systems where accuracy is paramount.   Incorporates feedback loops that detect and correct generation errors—boosts factuality for CLI edge‑cases, but corrective passes add inference steps. |
| **6. EfficientRAG** | * Strengths: Optimized for speed and cost-effectiveness. * Weaknesses: Potential trade-offs in accuracy and relevance, limited flexibility for complex tasks. * Use Case: Real-time chatbots requiring fast response times.   Optimized for multi‑hop retrieval without repeated LLM calls, drastically cutting inference time and compute costs—ideal for rapid CLI lookups.   * **No extra LLM calls per hop**: EfficientRAG reformulates multi‑hop queries iteratively on the retriever alone, eliminating costly LLM invocations between hops. * **Filtering of irrelevant info**: It prunes spurious retrievals, keeping only high‑value snippets for final generation * **Open‑source implementation**: Official code and data are available, accelerating prototyping and integration. |
| **7. Golden-Retriever RAG** | * Strengths: High-accuracy retrieval, reliable results. * Weaknesses: Specialized use cases, resource-intensive for high accuracy. * Use Case: Fact-checking applications demanding precise information retrieval.   Leverages high‑precision retrievers (e.g., tuned BM25 or dense models) for top‑quality context—improves accuracy for critical CLI commands but incurs heavier retrieval cost. |
| **8. Adaptive RAG** | * Strengths: Task-specific tuning, versatile across different domains. * Weaknesses: Complex tuning process, potential for overfitting. * Use Case: Personalized recommendation systems adapting to user preferences.   Dynamically adjusts retriever/generator pipeline per query (e.g., changing top‑k or model temperature), offering flexibility but adding tuning complexity and slight runtime overhead. |
| **9. Modular RAG** | * Strengths: Customizable components, flexible architecture. * Weaknesses: Integration challenges, higher maintenance requirements. * Use Case: Building complex conversational agents with interchangeable modules.   Splits retrieval, reranking, and generation into interchangeable modules—great for customization across AWS/Azure/Databricks docs, though integration layers introduce extra latency. |
| **10. Speculative RAG** | * Strengths: Proactive retrieval for potential performance gains, innovative approach. * Weaknesses: Risk of irrelevant retrievals, increased model complexity. * Use Case: Predictive text or email completion anticipating user needs.   **Proactively fetches likely needed passages** before full decoding, reducing average latency at the cost of occasional irrelevant retrievals—suited to interactive CLI autocompletion.   * **Draft–verify workflow**: A small specialist LM drafts multiple responses from different snippet subsets; the main LLM verifies the most promising one in a single pass citeturn1search0turn1search1. * **Latency gains**: When user queries follow predictable patterns (e.g., aws s3 ls), the drafter can pre‑fetch likely relevant docs before full decoding. * **Adaptive to CLI patterns**: Drafting rules can be guided by CLI command hierarchies to further reduce mis‑retrievals. |
| **11. RankRAG** | * Strengths: Improved relevance through better ranking algorithms, enhanced precision. * Weaknesses: Complex ranking algorithm development, potential for bias. * Use Case: Search engines prioritizing highly relevant results.   Incorporates advanced ranking metrics (e.g., MRR, Recall@k) to fetch fewer, more precise snippets—trims retrieval overhead without modifying generation, yielding good speed/accuracy trade‑off. **Precision Boosted Retrieval.**   * **Unified ranking & generation**: A single LLM is instruction‑tuned to both rank snippets and generate answers, yielding higher retrieval relevance and smaller context windows . * **Compute savings**: By limiting to the very top passages, the LLM processes fewer tokens per query. * **Domain adaptation**: Easily fine‑tuned on a small sample of AWS CLI Q&A pairs to learn command‑specific ranking nuances. |
| **12. Multi-Head RAG** | * Strengths: Parallel processing for potential speedup, diverse outputs. * Weaknesses: High computational demands, complex coordination between heads. * Use Case: Handling complex queries requiring multiple perspectives or information sources   Runs parallel retrievers (“heads”) for diverse sub‑queries and merges their outputs—enhances coverage but multiplies retrieval and fusion costs. |

**Theoretical Key takeaway**

For a CLI‑docs use‑case—where queries are short, the corpus is structured, and interactivity demands low latency—**EfficientRAG** and **Original RAG** lead on efficiency, followed by ranking‑based variants that minimize documents per inference. At the far end, graph‑based and multi‑headed schemes, while powerful for complex knowledge graphs, impose too much overhead for routine CLI lookups.

**To Implement…**

We can start implementing following and compare w.r.t. Multiple metric - scores ( which metric will be suitable?)

**Recommending RAG for AWS CLI docs**

**EfficientRAG** should be the primary choice for AWS CLI docs given its speed and iterative retrieval without extra LLM calls. **Original RAG** is a solid second choice because of its simplicity and moderate cost. **RankRAG** comes third to improve relevance and reduce documentation volume. **Speculative RAG** may be useful for pre-fetching, but other variants like GraphRAG or Multi-Head RAG are likely too complex.

**Recommendation Summary**

For the AWS CLI documentation use‑case—where queries are short, the corpus is structured (~500 command pages), and low latency is critical—**EfficientRAG** stands out as the top choice. It iteratively generates retrieval queries without invoking the LLM at each hop, dramatically cutting inference time and cost.

**Original RAG** remains a robust, easy‑to‑implement fallback, conditioning on a fixed set of passages per query with moderate compute overhead. To further trim retrieval costs and boost precision, **RankRAG**—which unifies context ranking and generation in a single model—offers more targeted snippet selection, reducing the number of passages the LLM must process .

**Speculative RAG** can be slotted in next: by drafting multiple retrieval‑augmented candidates in parallel with a small specialist LM and verifying with the main model, it achieves lower average latency when hints of likely-needed commands are known upfront.

## **Implementation Outline**

1. **Preprocess AWS CLI docs**
   * Crawl and parse command pages into JSONL: { "command": "...", "description": "...", "examples": [...] }.
2. **Indexing**
   * Build a vector index via FAISS or a BM25 index using ElasticSearch.

**EfficientRAG Setup** from efficientrag import EfficientRAG

retriever = EfficientRAG(document\_index="aws\_cli\_index.faiss")

model = load\_llm("gpt-neo-1.3B") # or any hosted/ in‑house LLM

* + EfficientRAG will iteratively refine the query until no new information emerges, then pass the filtered snippets to model.

**Original RAG Baseline** from transformers import RagTokenizer, RagRetriever, RagSequenceForGeneration

tokenizer = RagTokenizer.from\_pretrained("facebook/rag-sequence-nq")

retriever = RagRetriever.from\_pretrained("facebook/rag-sequence-nq", index\_name="custom", passages\_path="aws\_cli\_docs.tsv")

model = RagSequenceForGeneration.from\_pretrained("facebook/rag-sequence-nq", retriever=retriever)

1. **RankRAG Adaptation**
   * Instruction‑tune a Llama2‑7B model on pairs of (query, positive passage, negative passages), using the RankRAG objective.
   * Deploy via Hugging Face’s PEFT + TRL pipelines.
2. **Speculative RAG**
   * Use the SpeculativeRAG framework (ICLR 2025) to spin up a small 1–2 B‐parameter drafter LM and the main 7–13 B model.
   * Combine drafts via a minimal verification pass on the main model.

**Final Note:** Start with **EfficientRAG** for maximum throughput and minimal engineering overhead, validate accuracy against Original RAG, then iterate with **RankRAG** and **Speculative RAG** to squeeze out precision and latency improvements.