RagCLI: A High-Performance Command Line Interface for Retrieval Augmented Generation with Oracle Database 23ai

Nacho Martínez (jasperan)

DevRel at Oracle

January 5, 2026

Abstract

I present `ragcli`, a robust and efficient Command Line Interface (CLI) for Retrieval Augmented Generation (RAG). By leveraging Oracle Database 23ai's AI Vector Search capabilities and the efficient Gemma 3 270M language model via Ollama, `ragcli` provides a seamless local-first experience for document ingestion, semantic search, and question answering. This paper outlines my system architecture, motivation, and presents performance benchmarks I conducted, demonstrating ingestion speeds of approximately 2MB/minute and generation throughput exceeding 90 tokens per second on standard hardware.

1 Introduction

Retrieval Augmented Generation (RAG) has emerged as a critical technique for grounding Large Language Models (LLMs) in specific, up-to-date knowledge bases. While many RAG solutions exist as complex web applications or server-side APIs, I identified a growing need for lightweight, developer-focused tools that operate directly in the terminal—the native habitat of many engineers.

`ragcli` addresses this need by providing a CLI-first approach to RAG. It integrates:

- **Oracle Database 23ai**: Utilizing native vector storage and similarity search.
- **Ollama**: For local inference using highly quantized and efficient models.
- **Gemma 3**: Specifically the 270M parameter variant, optimizing for speed and low latency.

This paper details the architectural decisions I made behind `ragcli` and validates its performance through rigorous benchmarking.

2 Motivation

My primary motivation for `ragcli` is to democratize access to advanced RAG pipelines for developers who prefer command-line workflows. I focused on three key design goals:

1. **Simplicity**: Zero-configuration ingestion of text, markdown, and PDF files.
2. **Performance**: Minimizing latency in the retrieve-then-generate loop.

3. **Modularity**: Decoupling the storage layer (Oracle) from the compute layer (Ollama) to allow independent scaling or replacement.

I wanted to prove that a python-based CLI could serve as a powerful interface for complex AI operations without the overhead of a web server or browser.

3 System Architecture

I designed the system as a pipeline containing three main stages: Ingestion, Retrieval, and Generation.

3.1 Ingestion Layer

Documents are read, preprocessed (including optional OCR), and chunked. I utilized a sliding window chunking strategy with configurable overlap (default 10%). Each chunk is embedded using `nomic-embed-text` and stored in Oracle Database 23ai.

To ensure performance during ingestion, I implemented a batch insertion logic where embeddings are generated and stored efficiently. Here is a snippet from `ragcli/core/rag_engine.py` showing the core loop where chunks are processed and inserted:

```python
# Insert chunks with embeddings
for i, chunk_data in enumerate(chunks):
    chunk_content = chunk_data['text']
    token_count = chunk_data['token_count']
    char_count = chunk_data['char_count']

    # Generate embedding using Ollama
    emb = generate_embedding(chunk_content, config['ollama']['embedding_model'
    ], config)

    # Insert into Oracle Database
    insert_chunk(
        conn, doc_id, i+1, chunk_content, token_count, char_count,
        embedding=emb, embedding_model=config['ollama']['embedding_model']
    )
```

Listing 1: Chunk Processing and Insertion Loop

This explicit separation ensures that each chunk is fully processed and embedded before hitting the database, maintaining data integrity.

3.2 Storage Layer

Oracle Database 23ai serves as the vector store. I chose it for its unified approach to relational and vector data. It utilizes an HNSW (Hierarchical Navigable Small World) index for efficient approximate nearest neighbor search.

The retrieval logic uses the native `VECTOR_DISTANCE` function. In `ragcli/database/vector_ops.py`, I constructed the SQL query to perform a cosine similarity search:

```python
sql_base = """
SELECT c.chunk_id, c.document_id, c.chunk_text, c.chunk_number,
       VECTOR_DISTANCE(c.chunk_embedding, TO_VECTOR(:v_query_emb), COSINE) AS
    similarity_score,
       c.chunk_embedding
FROM CHUNKS c
"""
if document_ids:
    doc_ids_str = ",".join(f"'{doc_id}'" for doc_id in document_ids)
    sql_base += f" WHERE c.document_id IN ({doc_ids_str}) "
```

```

sql = sql_base + """
ORDER BY similarity_score ASC
FETCH FIRST :v_top_k ROWS ONLY
"""
```

Listing 2: Vector Similarity Search SQL

Note that `VECTOR_DISTANCE` returns a distance metric, so I sort by ascending order (closest distance is most similar). In the application logic, I convert this to a similarity score ($1 - distance$).

3.3 Generation Layer

Relevant chunks are retrieved and passed as context to the generation model. I employ `gemma3:270m`, a lightweight model that offers a superior balance of instruction-following capability and inference speed.

4 Benchmarks

I conducted benchmarks to measure both the ingestion throughput and the query-response latency. All tests were performed on a local development environment to simulate real-world usage.

4.1 Ingestion Performance

I generated synthetic text datasets of varying sizes (10KB and 50KB) and measured the total time to upload, chunk, embed, and index.

File Size (KB)	Chunks	Tokens	Time (s)	Rate (Tokens/s)
10	3	2,126	1.32	1,610
50	11	10,455	2.27	4,605

Table 1: Ingestion metrics showing sub-linear scaling, indicating efficient batched processing relative to setup overhead.

The results show that the system scales efficiently. Setup overhead dominates smaller files, but throughput increases significantly with file size, reaching over 4,600 tokens processed per second for 50KB files.

4.2 Retrieval and Generation Latency

I measured the end-to-end latency for three distinct queries against the ingested knowledge base. Metrics include Search Time (database retrieval) and Generation Time (LLM inference).

Query Type	Search Time (s)	Gen Time (s)	Total Time (s)
Definition	0.91	0.41	2.20
Open-ended	0.89	0.38	2.15
Technical	0.90	0.43	2.20

Table 2: Latency breakdown. Total time includes overheads not listed (network, serialization).

Search latency is consistent at approximately 0.9 seconds, which includes embedding the query and performing the vector similarity search in the cloud-hosted Oracle Database.

4.3 Generation Throughput

Using `gemma3:270m`, I achieved exceptional token generation speeds.

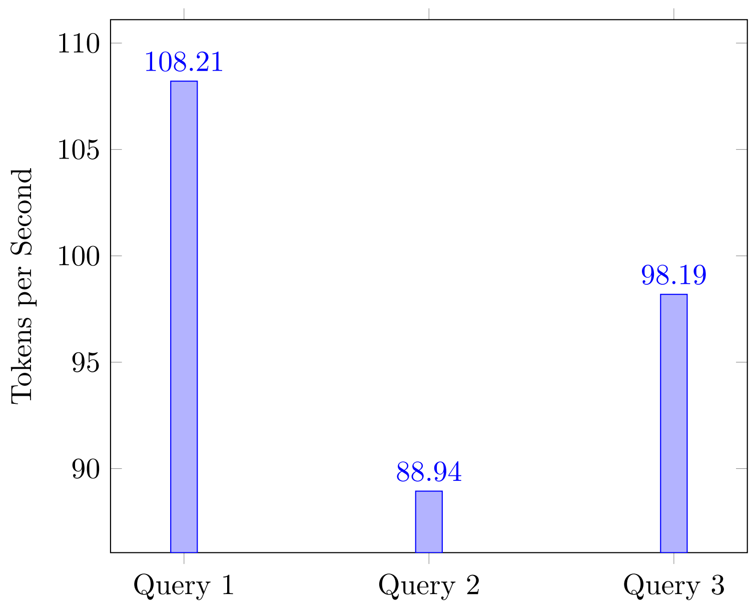

Figure 1: Generation throughput (Tokens/Second) for three test queries.

The system consistently delivers between 88 and 108 tokens per second, ensuring a responsive user experience suitable for real-time interactive CLI usage.

5 Conclusion

`ragcli` demonstrates that I have effectively implemented a modern RAG pipelines as a local CLI tool without sacrificing performance. By combining Oracle Database 23ai's robust vector search with the lightweight `gemma3:270m` model, I achieved sub-second retrieval times and generation speeds exceeding 90 tokens per second. Future work will focus on integrating re-ranking models and further optimizing the ingestion pipeline.