Technical Decisions, Implementation Challenges and Observations

Task 1: Wikipedia Text Scraping

Web Search:

- Uses SERP API from searchAPI.io (can easily switch search engines: Google ↔ Bing)
- Provides high-quality, structured search results

Limitation:

- Limited to 100 searches/month on free tier
- Not scalable without paid plan or alternative solution

Future Consideration:

For production deployment, consider a direct Wikipedia API for web search without searches/month constraint.

Used the Wikipedia API for Scraping Text

Rationale:

- Clean data extraction: Provides structured, well-formatted text without HTML noise
- Free and rate-limit friendly: No API key required, suitable for prototype development
- Maintenance-free: No need to handle Wikipedia's HTML structure changes

Limitations & Tradeoffs:

- Does not capture tables or images
- Acceptable for current text-only RAG implementation
- Model performance not degraded since the embeddings are text-based

Future Enhancement:

Use BeautifulSoup with custom parsing logic to extract:

- Tables → Convert to markdown/text representation
- Image captions → Include as contextual metadata
- Infoboxes → Structured data for enhanced retrieval

Task 2: Document Chunking and Vector Database Creation

Chunk Size Strategy

Configuration:

- Chunk size: 1500 characters
- Overlap: 150 characters (10%)
- Splitter: Recursive Character Text Splitter

Justification

1. Content-Aware Splitting

Wikipedia API returns well-structured text:

```
== Section Title 1 ==

paragraph text

paragraph text

== Section Title 2 ==
```

Modified separator hierarchy to preserve semantic structure:

```
separators = [
    "\n== ",      # Section headers
    "\n\n",       # Paragraph boundaries
    "\n",         # Line breaks
    ". ",         # Sentences
    " "           # Words (fallback)
]
```

2. Chunk Size Calculation

Average paragraph analysis:

- 200-300 words per paragraph
- ~5 characters per word (including spaces)
- Calculation: 250 words × 5 chars = 1,250 characters
- Buffer for context: +250 characters
- Final chunk size: 1,500 characters

This ensures most chunks contain 1-2 complete paragraphs, **preserving coherent semantic units.**

3. Overlap Rationale

Why overlap?

- Long paragraphs may get split mid-context
- Cross-chunk references lose meaning without overlap
- 150-character overlap ≈ 2-3 sentences of context

Example scenario:

> Chunk 1: "...Einstein's theory. This discovery led to..."
> Chunk 2 (no overlap): "...modern physics applications."
> Chunk 2 (with overlap): "This discovery led to modern physics applications."

Overlap ensures retrieval captures full context even when query terms span chunk boundaries.

Vector Database Selection

Choice: Milvus (by Zilliz)

Key Benefits

1. Hybrid Search Capability

- Supports dense vectors (semantic similarity via embeddings)
- Supports sparse vectors (BM25 lexical matching)
- Critical for handling both semantic and keyword-based queries

2. Scalability

- Designed for billion-scale vector search
- Horizontal scaling via distributed architecture
- Suitable for production deployment

3. Performance Optimization

- Multiple indexing methods: FLAT, HNSW, IVF, PQ etc.

- HNSW for low-latency approximate search
- Trade-off between accuracy and speed configurable per use case

4. Extensibility

- Multi-vector support for multimodal search (text + image embeddings)
- Future-proof architecture for vision-language models

5. Deployment & Cost

- Open-source with a permissive license
- Managed cloud option (Zilliz Cloud) for easy deployment

Drawbacks - Operational Complexity, Resource Constraints (Memory / Compute)

- Setup and configuration require technical expertise
- Need to understand indexing parameters (nlist, M, efConstruction, ef)
- Cluster management overhead for distributed deployment
- Steeper learning curve than lightweight alternatives
- Higher memory footprint for HNSW indexes
- May be overkill for small datasets (<1M vectors)
- GPU acceleration benefits require appropriate hardware investment
- Tradeoff between index size and query performance

Alternatives Considered

Database	Best For	Why Not Chosen
FAISS	Fast prototyping, single-machine	No native hybrid search, in-memory only
ChromaDB	Simple RAG applications	Limited scalability, fewer indexing options
Pinecone	Managed service, zero ops	Proprietary, vendor lock-in, cost at scale
Qdrant	Rust performance, filtering	Smaller ecosystem, less mature than Milvus

Decision: Milvus chosen for hybrid search support and scalability headroom, accepting higher initial complexity as a worthwhile investment for a production-grade RAG system.

Summary: Design Philosophy for Document Chunking and Vector DB

All technical decisions prioritize:

- Semantic coherence in chunking (content-aware splitting)
- Retrieval quality through hybrid search (dense + sparse vectors)
- Scalability for production deployment
- Future extensibility (multimodal, larger datasets)

These choices balance prototype speed with production-readiness.

Task 3: ASR Implementation

1. macOS Local and Google Colab Setup

Issues:

- Dependency conflicts: huggingface-hub, transformers, tokenizers

FFmpeg missing: `libavutil.so.57: cannot open shared object file`

- Fixed: Replaced torchaudio with librosa + soundfile

GPU not detected: ONNX Runtime CUDAExecutionProvider unavailable

- Fixed: Installed onnxruntime-gpu

Dtype mismatch: Float32 vs Float64 in TorchScript preprocessing

- Fixed: Standardized to float32

2. Model Format Migration (.nemo → .onnx)

Attempted: Switching from `indicconformer_stt_hi` (500 MB) to `indic-conformer-600m-multilingual` (2.3 GB)

Tradeoff:

- Better deployment compatibility
- However, 4.6× larger model, higher compute needs

Result: Failed with TorchScript runtime error in preprocessing module

```
RuntimeError: torch.matmul(torch.transpose(specgram, -1, -2),
CONSTANTS.c4)
```

Root cause: Serialized model carried environment-specific compilation artifacts incompatible with target runtime.

3. Final Solution

Approach: Speech-to-Text API

Rationale: Time-boxed troubleshooting revealed deeper incompatibilities. API provided a production-ready solution while preserving time for other tasks.

Task 4: Translation API Configuration

Implementation:

```
response = client.text.translate(
    input=input_text,           # limit: 1000 chars for mayura:v1 model
    source_language_code="auto",
    target_language_code="en-IN",
    speaker_gender="Female",
    mode="modern-colloquial",   # best for Hinglish input, chatbot use
cases
    model="mayura:v1",          # sarvam-translate:v1 does not support
modern-colloquial
    numerals_format="international",
)
```

Parameter Decisions:

1. `model="mayura:v1"`

- Chosen over sarvam-translate:v1 because mayura:v1 supports modern-colloquial mode
- Critical for handling Hinglish (Hindi-English code-mixed) conversational input

2. `mode="modern-colloquial"`

- Best suited for chatbot and conversational AI use cases
- Produces natural, informal translations rather than formal/literary style
- Handles contemporary slang and mixed-language input better

3. `numerals_format="international"`

- Uses standard Arabic numerals (1, 2, 3...) instead of Devanagari numerals
- Ensures consistency and readability for international users

4. `source_language_code="auto"`

- Automatic language detection for flexibility
- Handles multi-language input without pre-classification

Constraint:

Input text limited to 1000 characters for mayura:v1 model. Longer text requires chunking.

Task 5: RAG Pipeline Integration

Implementation:

Combined retrieval, translation, and LLM response generation into unified pipeline.

Prompt Engineering Strategy:

Designed grounding-focused prompt to ensure factual, context-based responses:

```
- Base your answer primarily on the provided context.
- If the context is insufficient, you may use general knowledge, but keep
the answer factual.
- If you truly don't know, say so honestly.
- Always answer in {output_lang}.
```

Key Design Decisions:

1. Context Grounding

- "Base your answer primarily on the provided context" — prevents hallucination
- Ensures responses are derived from retrieved documents, not LLM general knowledge

2. Fallback to General Knowledge

- "If the context is insufficient, you may use general knowledge" — graceful degradation
- Balances strict grounding with practical utility when retrieval fails

3. Honesty Over Guessing

- "If you truly don't know, say so honestly" — prevents confident misinformation
- Builds user trust by admitting uncertainty rather than fabricating answers

4. Multilingual Output Control

- "Always answer in {output_lang}" — enforces consistent language output
- Ensures response matches user's expected language regardless of context language

Result:

This prompt design creates a trustworthy, grounded RAG system that prioritizes factual accuracy while maintaining usability.