Cloud RAG Workshop

Building AI Search Systems Without the Hardware Headache

Srikrishna S Kashyap
AI Researcher

Liverpool John Moores University
Founder, Kashyap Tech

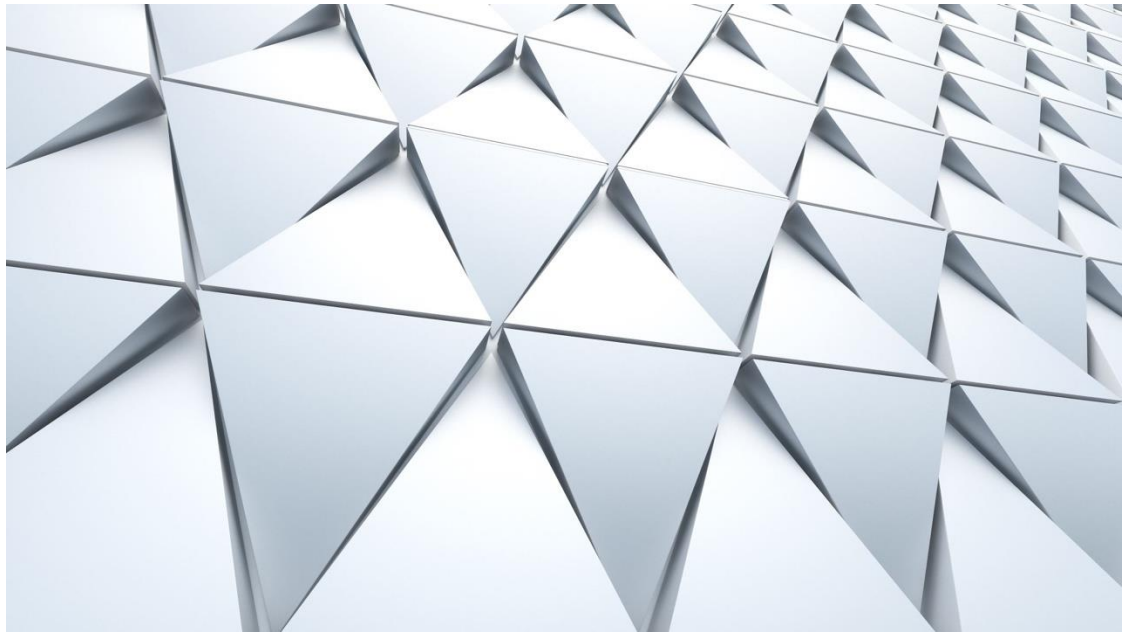
Workshop Approach

Theory → Practice: Each concept followed by hands-on implementation

Cloud-native: Using only free cloud services accessible from anywhere

Real-world patterns: Industry-standard architecture and techniques

Troubleshooting focus: Learn to debug common RAG issues



Workshop Overview

Day 1: Foundations & Infrastructure

- Introduction to RAG & Cloud Architecture
- Text Processing & Embeddings
- Cloud Vector Database Setup
- Document Processing Pipeline

Day 2: Building & Deploying RAG

- LLM Integration with Cloud Providers
- Building the Retrieval Component
- Full RAG Integration & Supabase Backend
- RAG Evaluation & Optimization

What You'll Build

A fully functional RAG system running in the cloud

Experience with production-ready tools and services

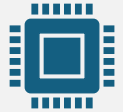
A deployable application that answers questions using your documents

Hands-on experience with vector databases, LLMs, and modern data pipelines

How This Works

CONCEPT INTRODUCTION
(SLIDES)

ARCHITECTURE & COMPONENTS
(SLIDES)



IMPLEMENTATION
(GOOGLE COLAB)

TESTING & VALIDATION
(GOOGLE COLAB)

DISCUSSION & QUESTIONS
(INTERACTIVE)

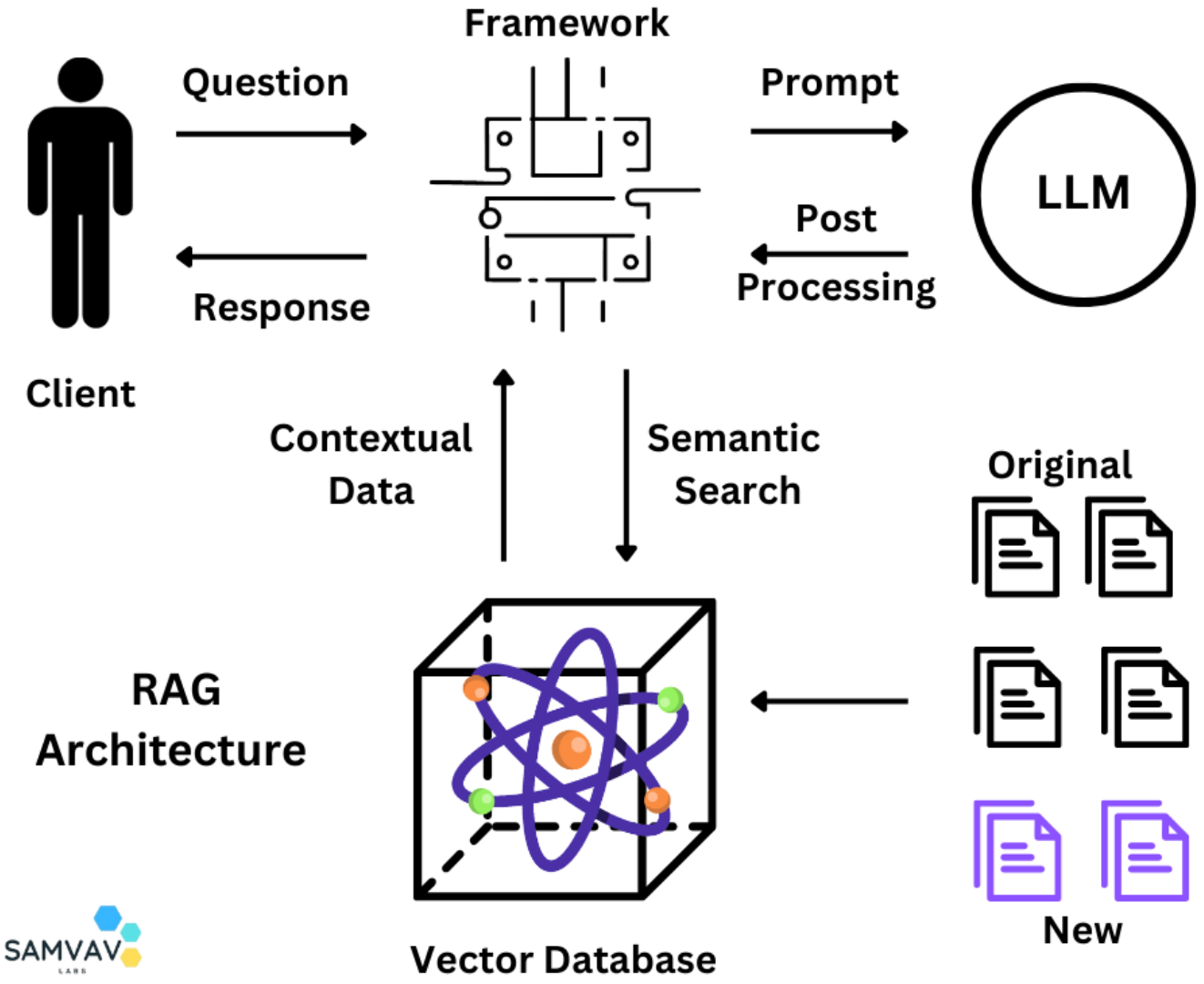
Let's get started!

Introduction to RAG & Cloud Architecture

What is Retrieval-Augmented Generation (RAG)?

RAG combines:

- **Retrieval:** Finding relevant information from a knowledge base
- **Generation:** Creating natural language responses with an LLM



Why RAG Matters

Knowledge Limitations

LLMs have knowledge cutoffs and outdated information

Hallucination Reduction

Grounding responses in retrieved facts

Domain Adaptation

Specializing general models for specific domains

Cost Efficiency

Smaller models with external knowledge can outperform larger models

Core RAG Components

DOCUMENT PROCESSOR

CHUNKING, CLEANING, AND PREPARING DOCUMENTS

EMBEDDING MODEL

CONVERTING TEXT TO VECTOR REPRESENTATIONS

VECTOR DATABASE

STORING AND RETRIEVING VECTORS EFFICIENTLY

QUERY PROCESSOR

PREPARING USER QUERIES FOR RETRIEVAL

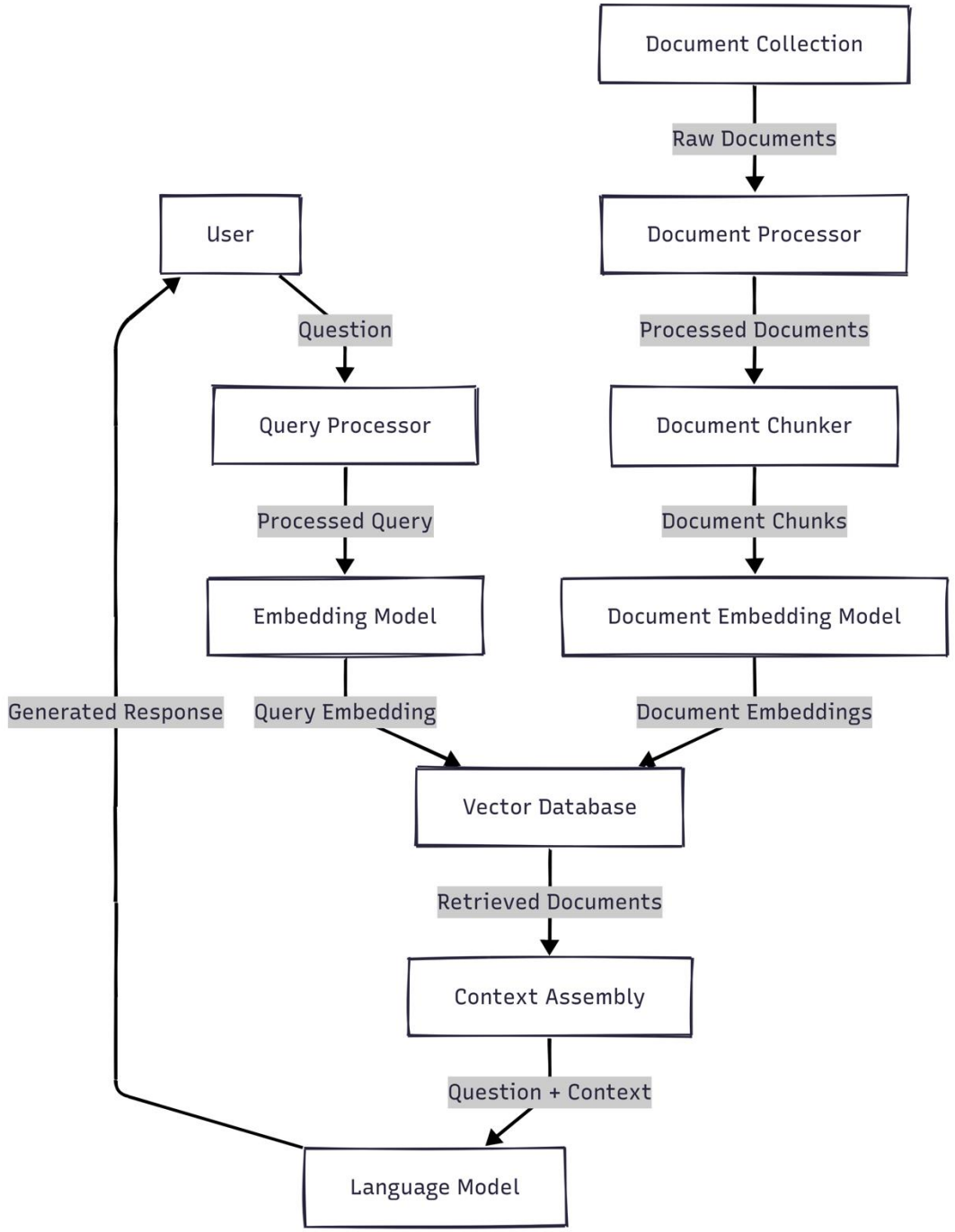
RETRIEVER

FINDING RELEVANT INFORMATION

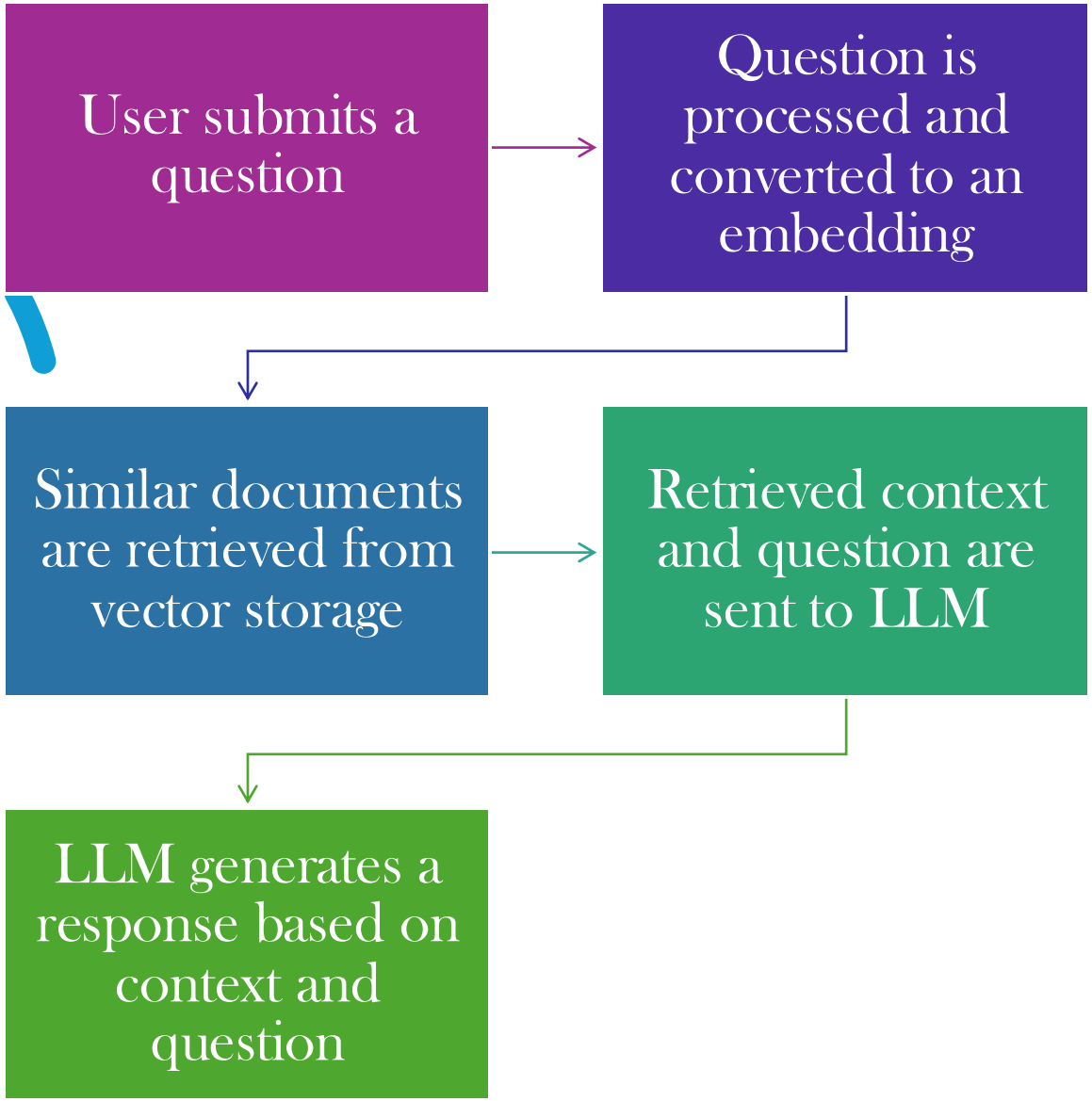
GENERATOR

CREATING NATURAL LANGUAGE RESPONSES WITH LLMS

Basic RAG Architecture



The RAG Pipeline



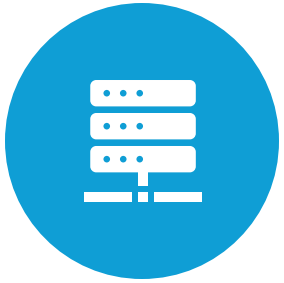
Cloud-Based RAG Architecture

Vector Store

Qdrant Cloud (vector database)

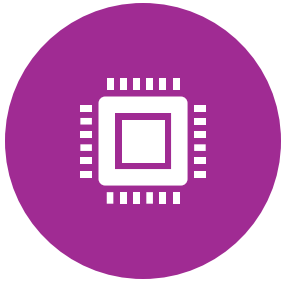
Database/Backend

Supabase (PostgreSQL + APIs)



LLM Provider

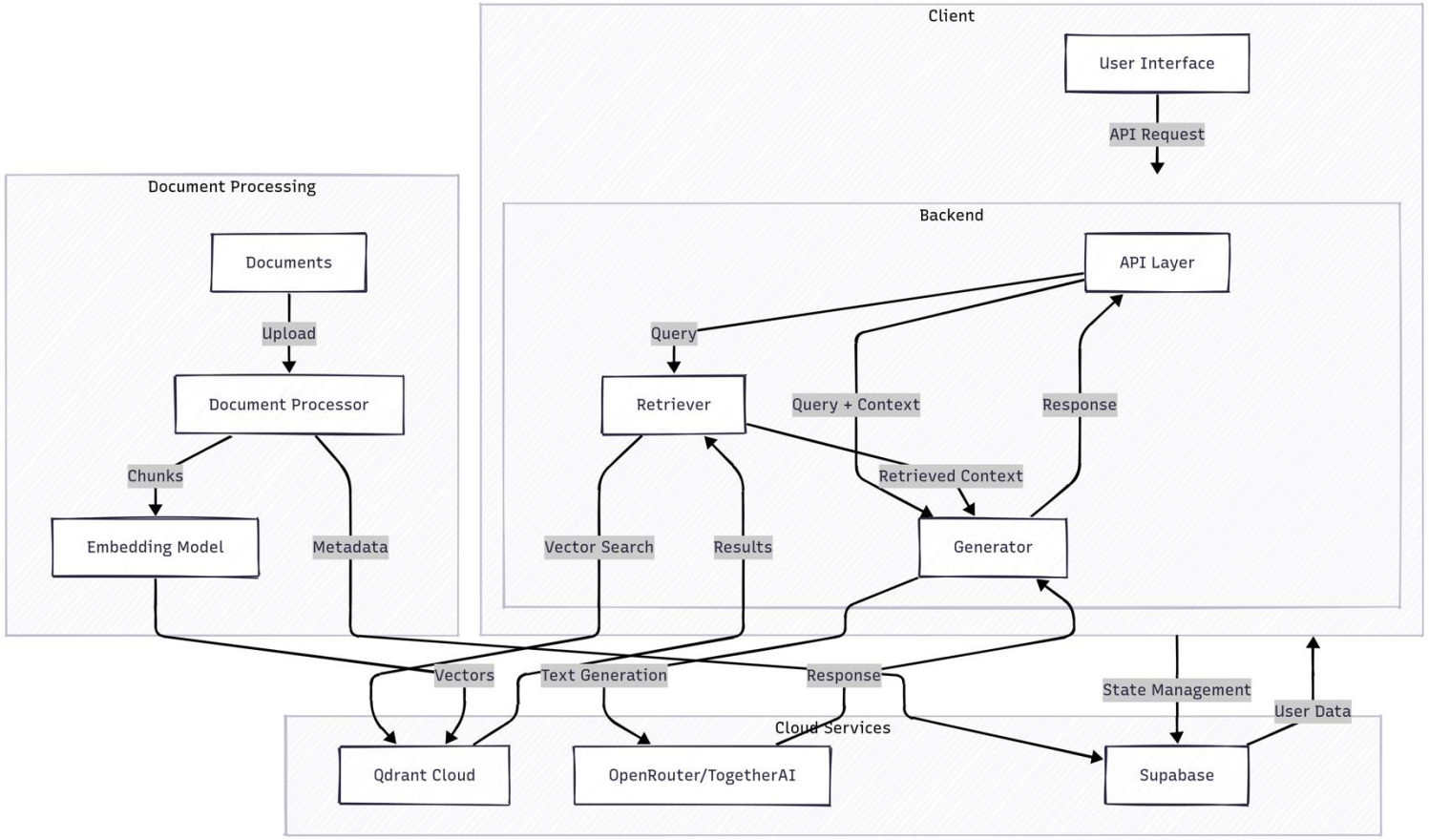
OpenRouter/TogetherAI (hosted models)



Orchestration

Google Colab (our implementation environment)

Cloud-Based RAG System



Services We'll Use

- **Qdrant Cloud:** Vector database for semantic search
- **Supabase:** Backend database and serverless functions
- **OpenRouter:** Multi-model LLM gateway
- **TogetherAI:** Alternative LLM provider
- **Google Colab:** Development environment

Building in the Cloud vs. Locally

ASPECT	CLOUD APPROACH	LOCAL APPROACH
Setup Complexity	Simple account creation	Complex environment setup
Hardware Requirements	Minimal (browser only)	Significant (especially for LLMs)
Scalability	Built-in	Manual configuration
Costs	Usage-based (free tiers available)	Upfront hardware investment
Accessibility	Available anywhere	Limited to local machine

RAG Use Cases

- **Documentation Assistants:** Technical support and knowledge bases
- **Research Tools:** Literature analysis and synthesis
- **Legal Document Analysis:** Contract review and legal research
- **Customer Support:** Personalized responses based on company knowledge
- **Educational Systems:** Interactive learning assistants

Workshop Environment Setup

Google Colab for code execution

API connections to our cloud services

Testing connectivity to all services

Time to switch to Colab for environment setup!

Text Processing & Embeddings

Text Preprocessing Fundamentals

Cleaning: Removing irrelevant content (HTML tags, special chars)

Normalization: Converting to consistent format (lowercase, unicode)

Tokenization: Breaking text into tokens (words, subwords)

Filtering: Removing stopwords and noise

What are Embeddings?

Text Embeddings:
Dense vector representations of text that capture semantic meaning

Texts with similar meaning have similar vectors

Allow machines to understand semantic relationships

Enable "fuzzy" matching beyond exact keyword search

Typically have hundreds of dimensions (e.g., 384, 768, 1536)

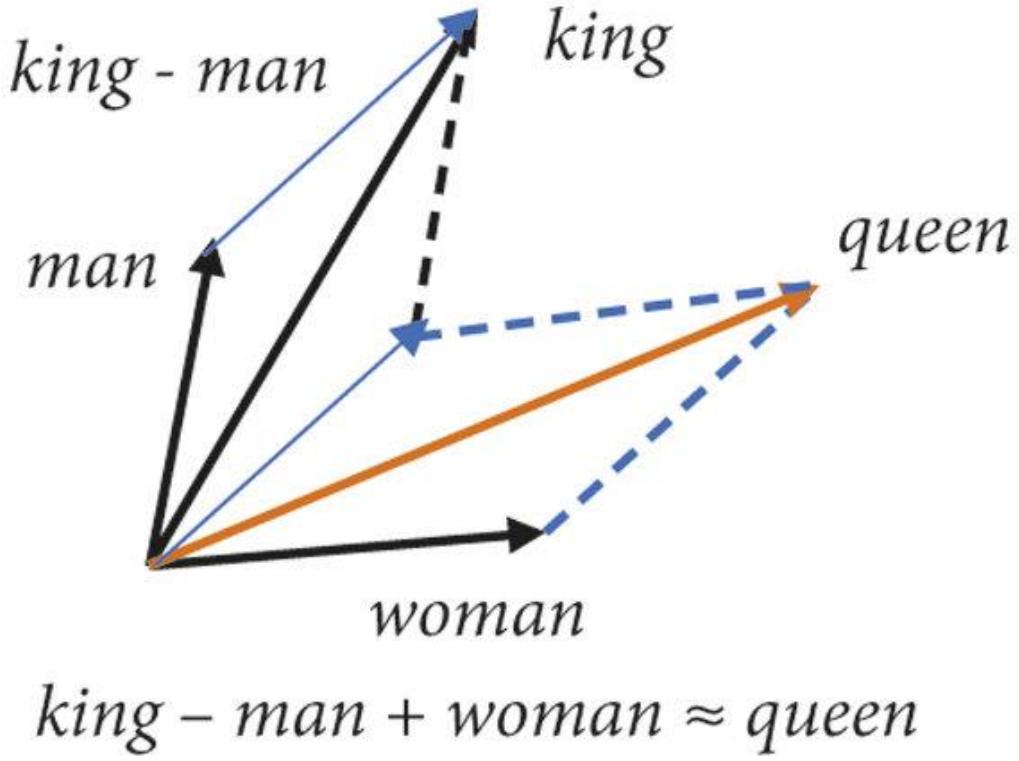
How Embeddings Work

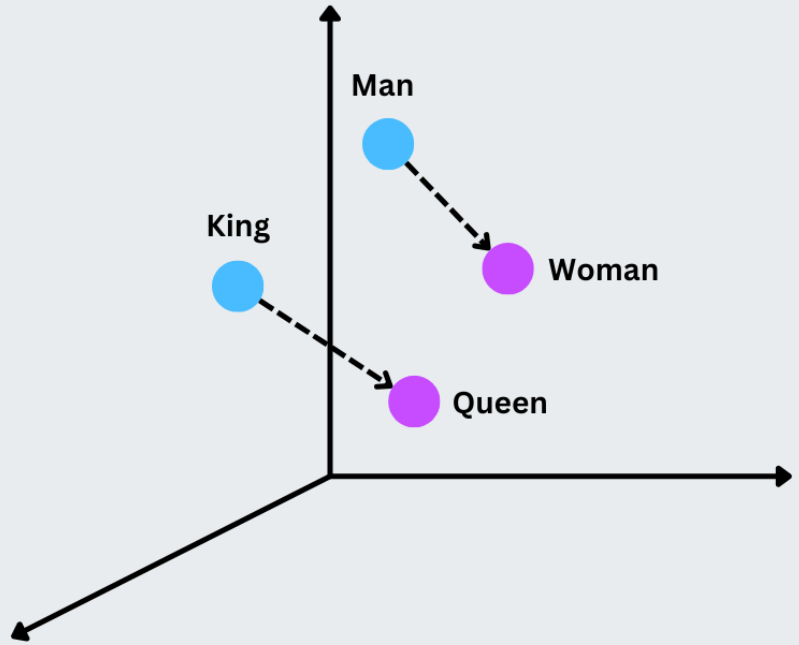
Each dimension represents some aspect of meaning

Similar concepts cluster together in vector space

Mathematical operations can reveal semantic relationships

Example: "king" - "man" + "woman" ≈ "queen"

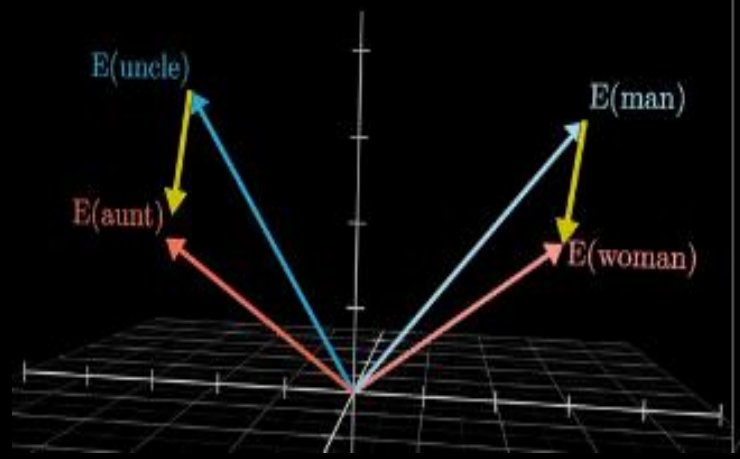




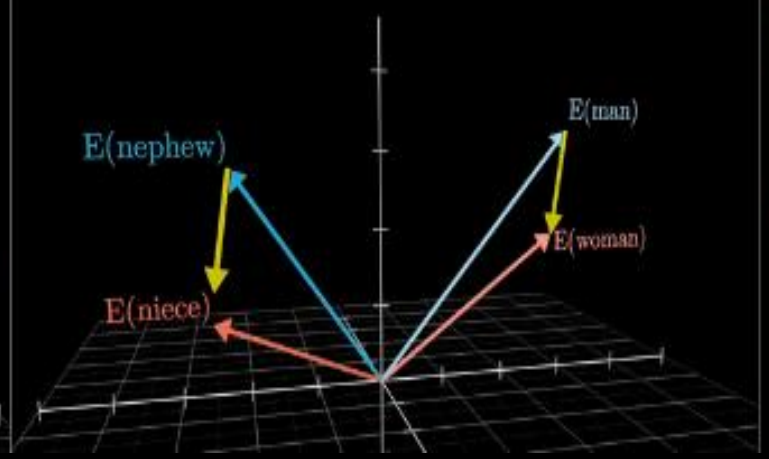
Word Embedding

Example: "king" - "man" + "woman" ≈ "queen"

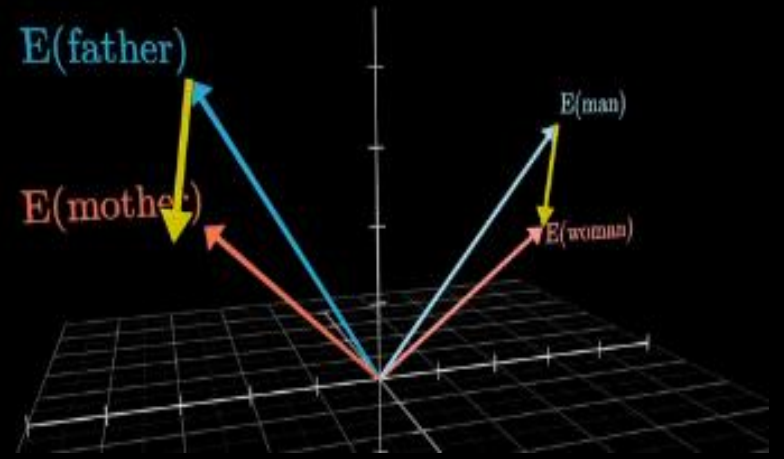
E(aunt) - E(uncle) ≈ E(woman) - E(man)



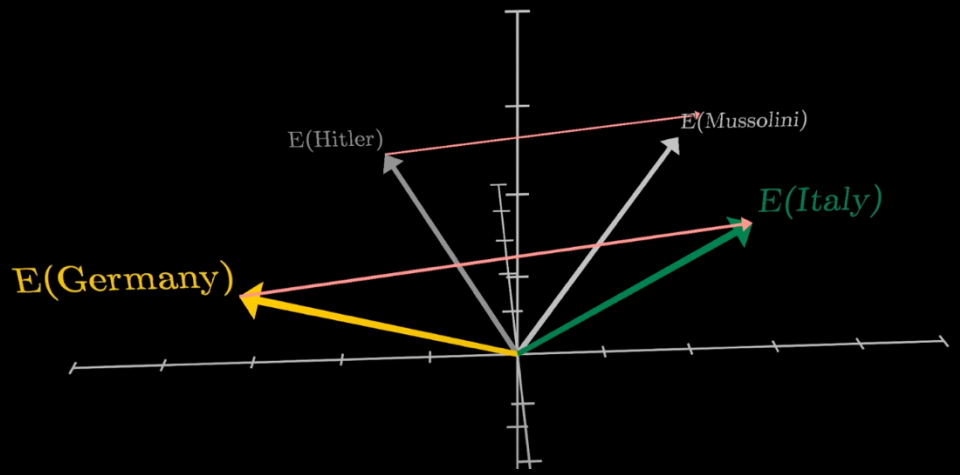
E(niece) - E(nephew) ≈ E(woman) - E(man)



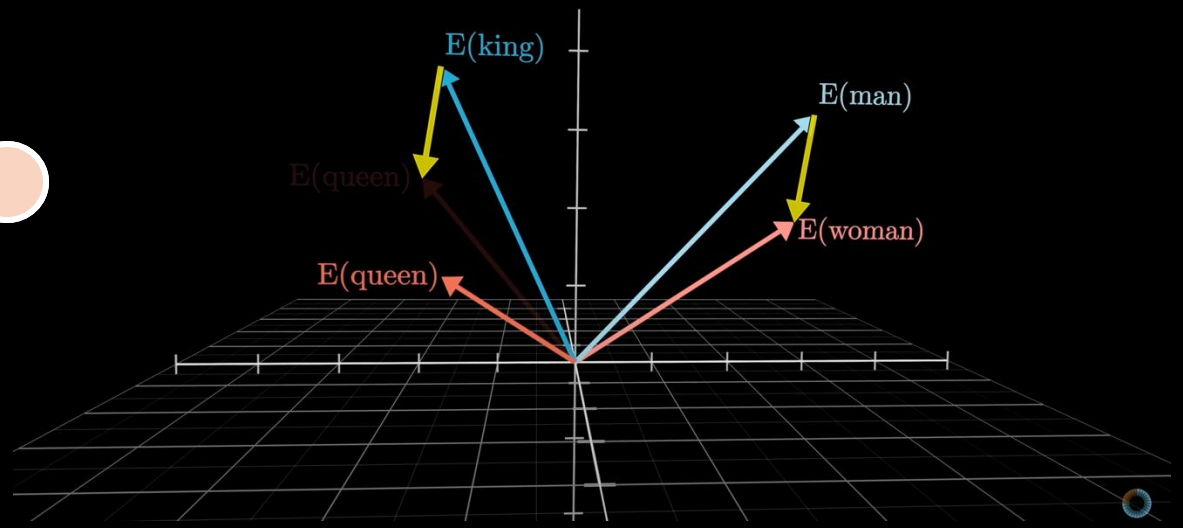
E(mother) - E(father) ≈ E(woman) - E(man)



E(Hitler) + E(Italy) - E(Germany) ≈ E(Mussolini)



E(queen) ≈ E(king)+E(woman) - E(man)



https://www.3blue1brown.com/lessons/gpt#embedding

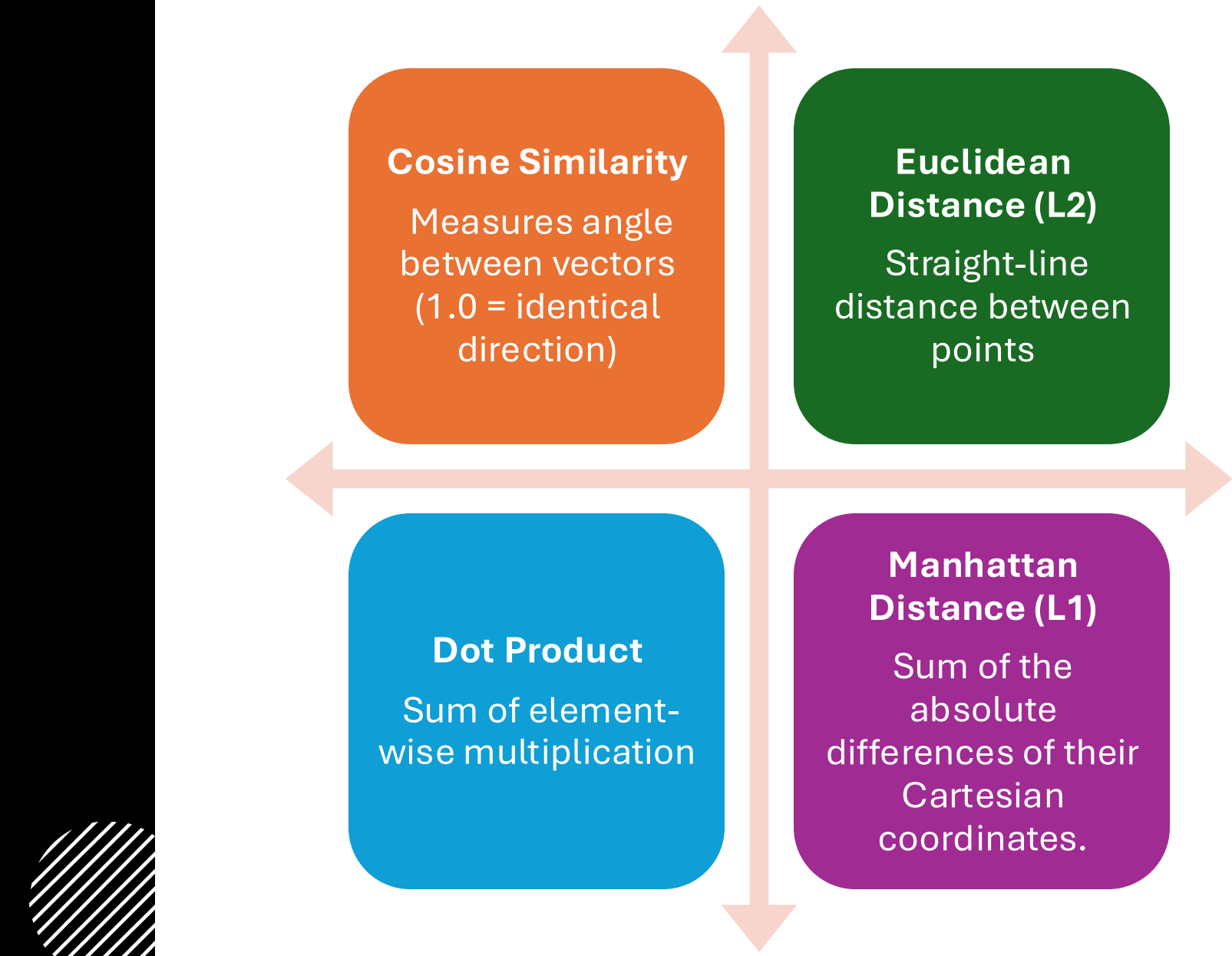
Embedding Models

Model	Dimensions	Size	Strengths
all-MiniLM-L6-v2	384	Small	Fast, good for RAG
all-mpnet-base-v2	768	Medium	High quality for English
text-embedding-ada-002	1536	Large	Strong performance, multilingual
E5-large	1024	Large	State-of-the-art for retrieval

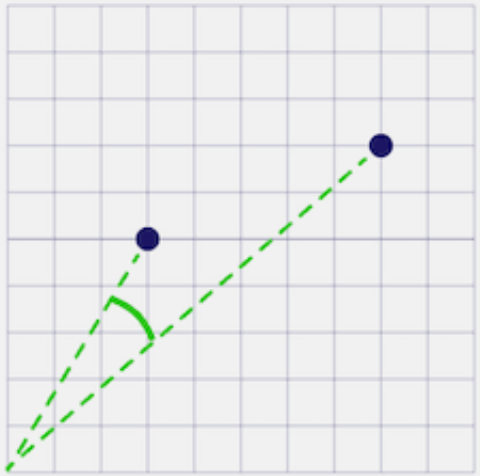
We'll use all-MiniLM-L6-v2 for its balance of quality and speed.

Vector Similarity Metrics

RAG systems typically use cosine similarity for text embeddings.

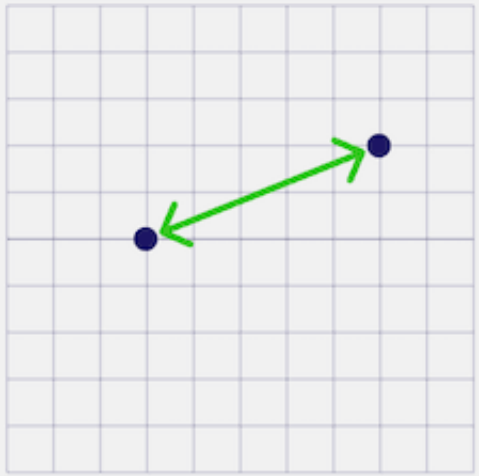


Distance Metrics in Vector Search



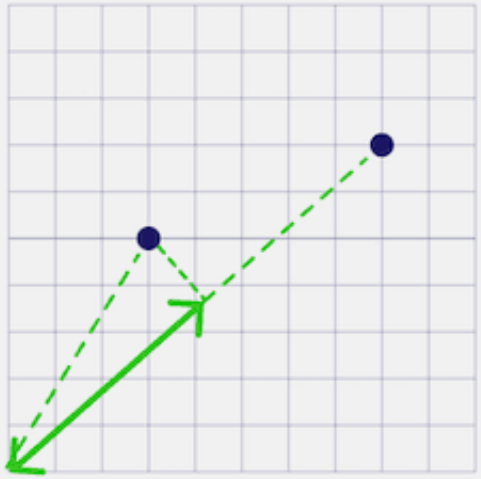
Cosine Distance

$$1 - \frac{A \cdot B}{||A|| \quad ||B||}$$



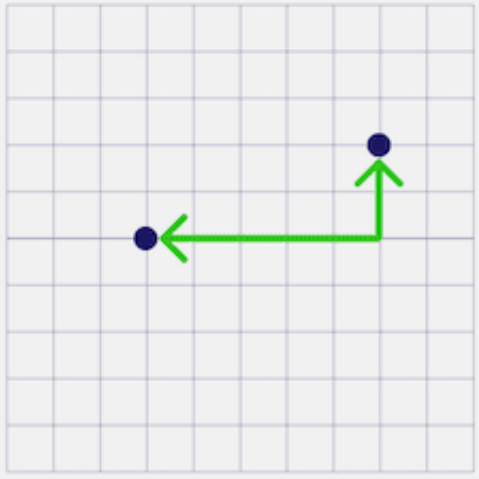
Squared Euclidean (L2 Squared)

$$\sum_{i=1}^{n} (x_i - y_i)^2$$



Dot Product

$$A \cdot B = \sum_{i=1}^{n} A_i B_i$$



Manhattan (L1)

$$\sum_{i=1}^{n} |x_i - y_i|$$

From Words to Documents

How to get from words to document embeddings?

01

Word Level / Token Level

Individual words or subword tokens

02

Sentence Level

Meaningful sentence representations

03

Paragraph Level

Coherent passage embeddings

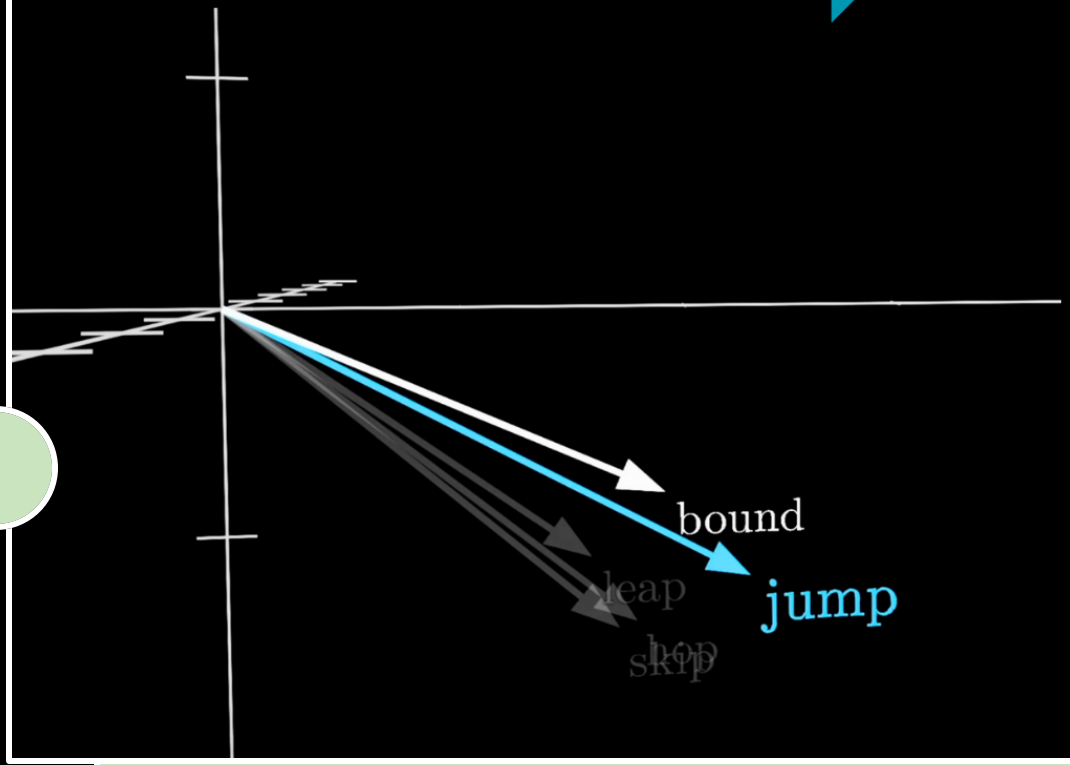
04

Document Level

Complete document representations

Most RAG systems use **sentence/paragraph** embeddings for optimal retrieval.

Tokens assigned to Vectors



To	date	.	the	cle	ve	rest	thin	of		time
5.4	7.8	9.7	2.6	3.6	5.6	1.6	9.7		3.2	6.7
7.1	5.2	7.9	7.7	4.3	4.3	1.1	4.6		6.6	2.7
6.0	5.6	4.6	4.5	6.9	9.8	6.5	9.7		1.3	7.3
5.4	9.2	7.7	5.6	0.6	1.0	1.4	6.0		7.1	9.5
4.2	0.7	1.2	0.2	6.6	2.1	1.9	7.3	...	2.9	2.5
6.4	0.9	6.3	6.1	6.6	1.6	3.7	0.4		1.8	5.7
4.3	0.2	1.4	6.1	2.1	6.5	8.1	2.8		5.8	5.9
8.8	8.2	9.4	6.1	1.3	2.5	1.0	1.2		0.2	5.7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮		⋮	⋮
3.8	8.6	4.1	6.8	3.6	2.4	1.0	1.2		0.0	9.4

Words with similar meanings have close vecto

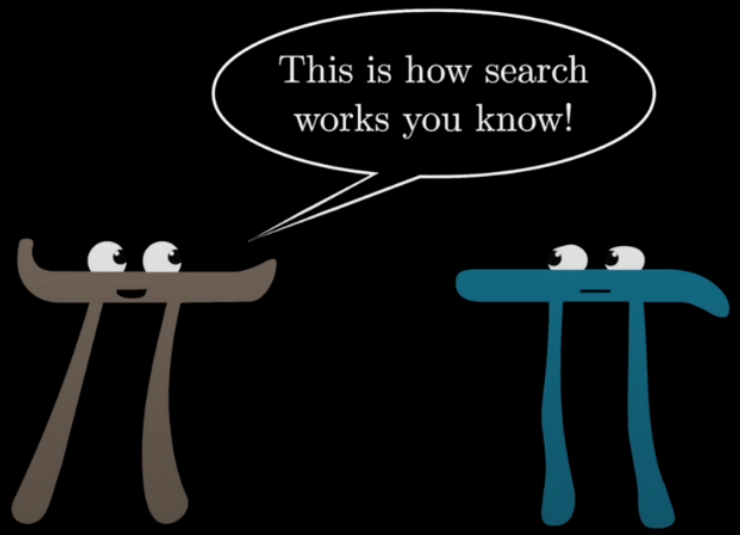
All words, $\sim$ 50k

aah	aardvark	aardwolf	aargh	ab	aback	abacterial	abacus	abalone	abandon	⋯	zygoid	zygomatic	zygomorphic	zygosis	zygote	zygotic	zyme	zymogen	zymosis	zzz
−8.6	−1.5	−4.8	+6.9	−9.2	+9.1	−2.9	−2.8	−9.6	−6.2	⋯	−2.0	+8.5	−7.9	+8.8	+7.3	−0.9	−3.4	−5.3	+2.3	−9.2
−9.6	−1.4	−8.5	−4.9	−5.5	−4.9	−7.3	−9.7	−7.6	+2.3	⋯	+9.4	+9.7	−1.8	−6.7	+2.7	−0.2	+9.7	−8.6	+5.6	−4.2
−5.1	+3.2	−5.0	+3.3	+0.3	−1.5	+1.1	−4.2	+4.1	−1.7	⋯	−2.8	+6.5	+8.4	−9.0	−5.3	−3.0	+6.2	+9.6	+9.3	+8.0
−4.0	+9.7	−5.0	−7.8	+8.9	−5.3	+3.8	−8.7	+4.6	+7.6	⋯	−4.5	−2.4	−2.5	+4.9	−5.2	−6.5	−1.0	−3.9	+6.7	−5.2
+0.0	+8.8	+2.7	+7.3	+8.7	+5.0	+3.9	+9.3	+9.8	−0.9	⋯	−8.5	−4.1	−6.9	−1.6	−7.3	+2.1	−2.3	+7.8	+9.3	+0.9
−4.5	+1.8	+7.8	−1.9	+1.0	−4.5	−0.9	−1.9	−5.0	+0.1	⋯	−3.8	−2.5	+0.5	+5.0	−3.3	+8.4	+7.2	−8.9	−4.9	−1.1
−7.8	−3.0	+4.7	+3.6	+2.4	+4.2	−5.8	−3.1	+3.5	+7.5	⋯	+0.9	−4.3	−9.3	+4.2	−9.7	−2.5	+0.6	+8.3	−8.1	−1.9
−9.4	−3.1	+2.4	−4.4	−5.7	−7.6	+1.5	+3.9	+3.4	+8.9	⋯	−9.8	+2.9	+2.0	+1.8	+9.2	−9.6	+3.9	+6.2	+0.2	−3.3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
+5.8	−8.0	−1.1	+0.4	+3.8	−8.1	−5.4	−1.8	+2.4	+7.7	⋯	+2.4	−7.3	+9.5	+7.4	+0.1	+8.4	+0.8	+8.4	+6.5	+9.3

$W_E =$

Embedding matrix

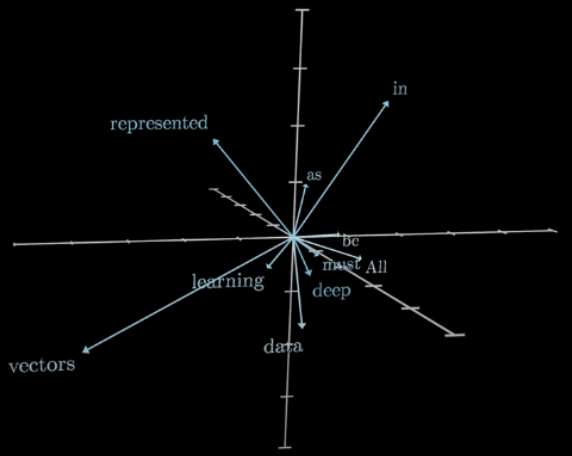
Embedding Matrix



Embedding Process

Words → "Embedding" → Vectors

All
data
in
deep
learning
must
be
represented
as
vectors



Visualizing Embeddings

High-dimensional vectors are difficult to visualize

Dimensionality reduction techniques:

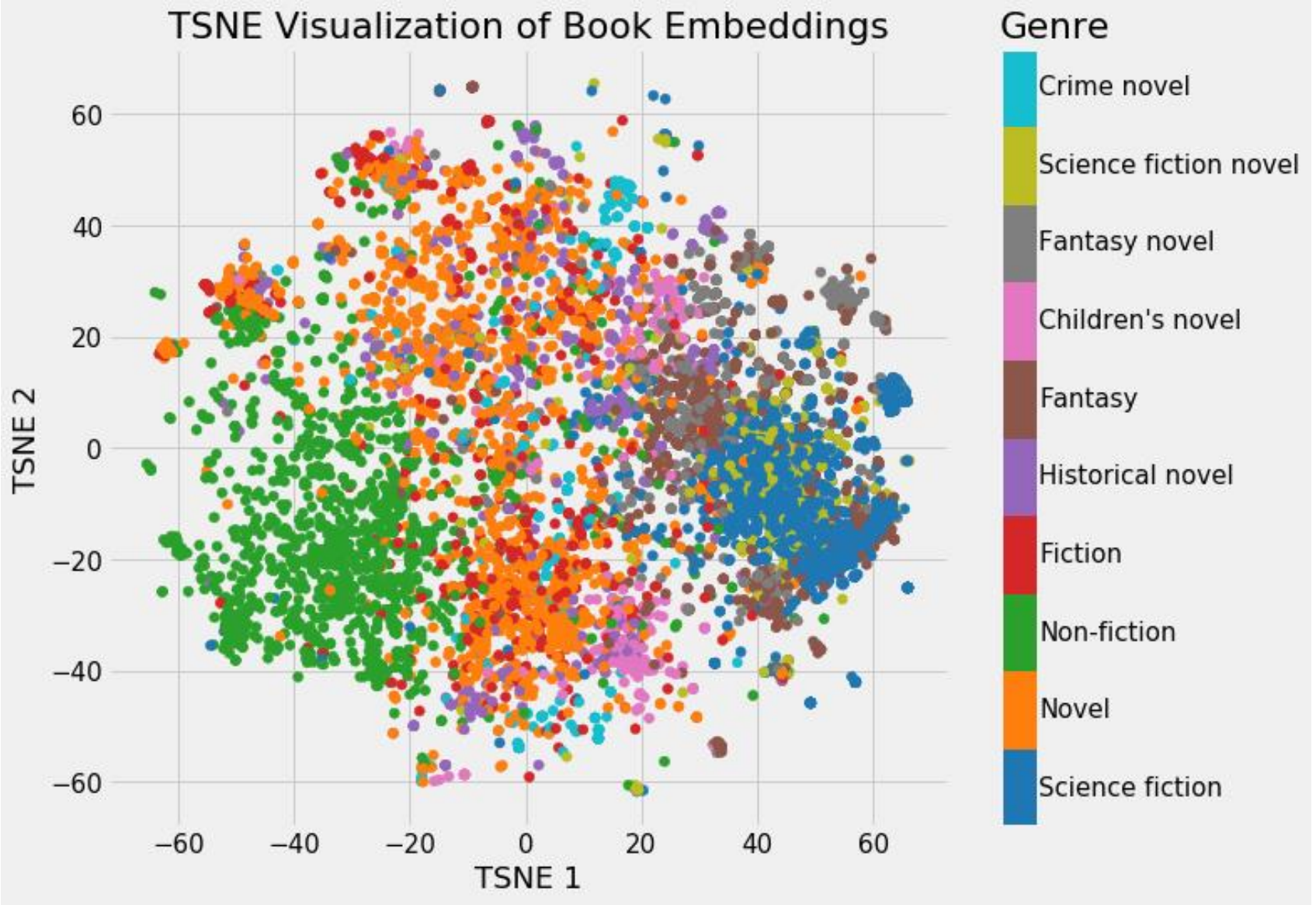
PCA: Principal Component Analysis (linear)	**t-SNE**: t-Distributed Stochastic Neighbor Embedding (non-linear)	**UMAP**: Uniform Manifold Approximation and Projection (preserves more global structure)

Word Embedding Visualization - Demos

https://koaning.github.io/cluestar/

https://projector.tensorflow.org/

https://jasonkessler.github.io/demo_compact.html


TSNE Visualization of Book Embeddings
TSNE 2
TSNE 1
60
40
20
0
−20
−40
−60
−60
−40
−20
0
20
40
60
Genre
Crime novel
Science fiction novel
Fantasy novel
Children's novel
Fantasy
Historical novel
Fiction
Non-fiction
Novel
Science fiction

Embedding Best Practices for RAG

Choose	Choose the right granularity: Sentence or paragraph level works best
Apply	Consistent processing: Apply the same preprocessing to queries and documents
Fine-tune	Domain adaptation: Fine-tune embeddings for specialized domains when possible
Ensure	Context preservation: Ensure chunks have enough context for understanding
Model	Model selection tradeoffs: Larger models aren't always better!

Let's Generate Embeddings!

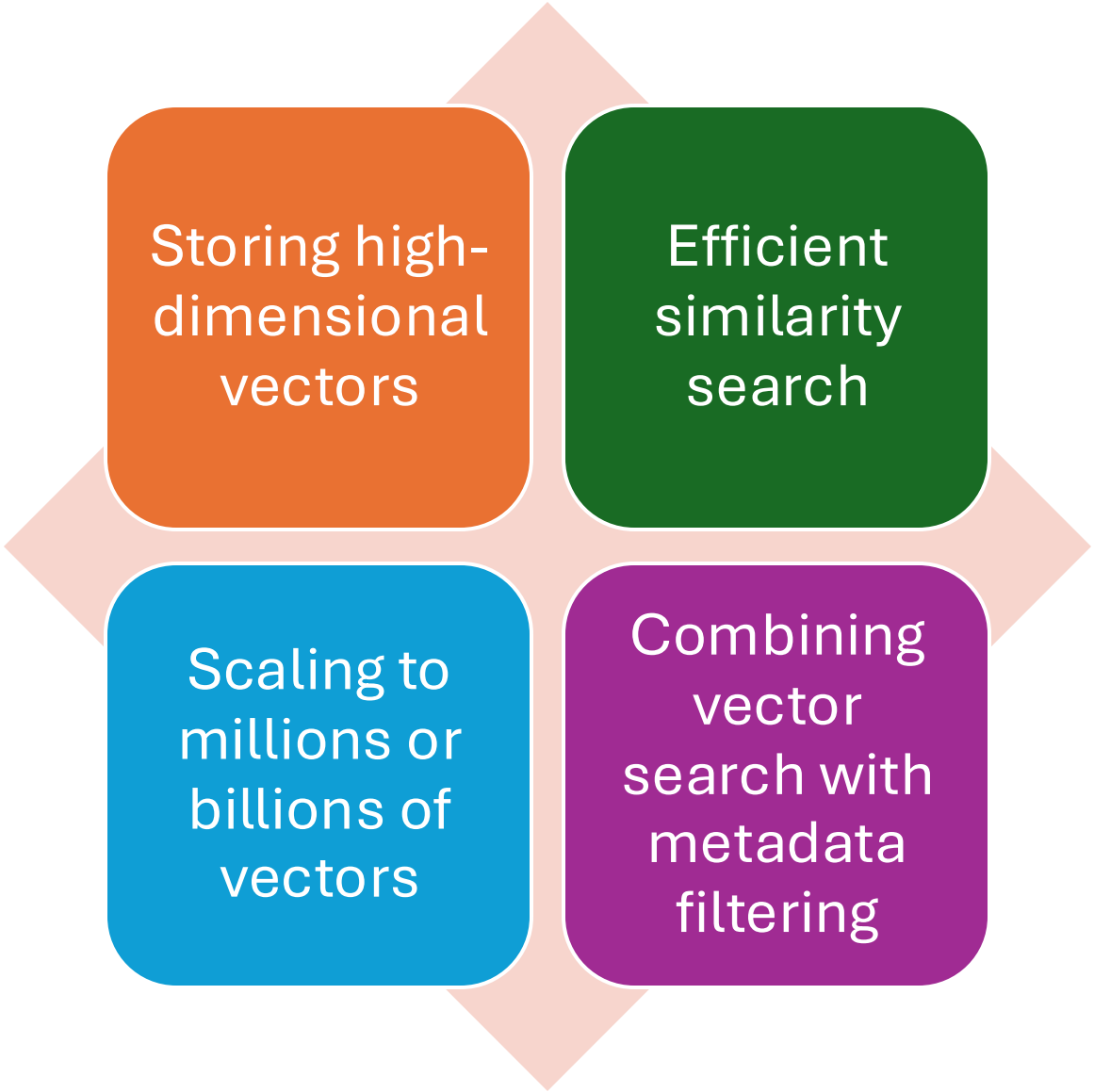
- Text preprocessing pipeline
- Embedding generation
- Vector visualization
- Simple similarity search

Time to switch to Colab for embedding implementation!

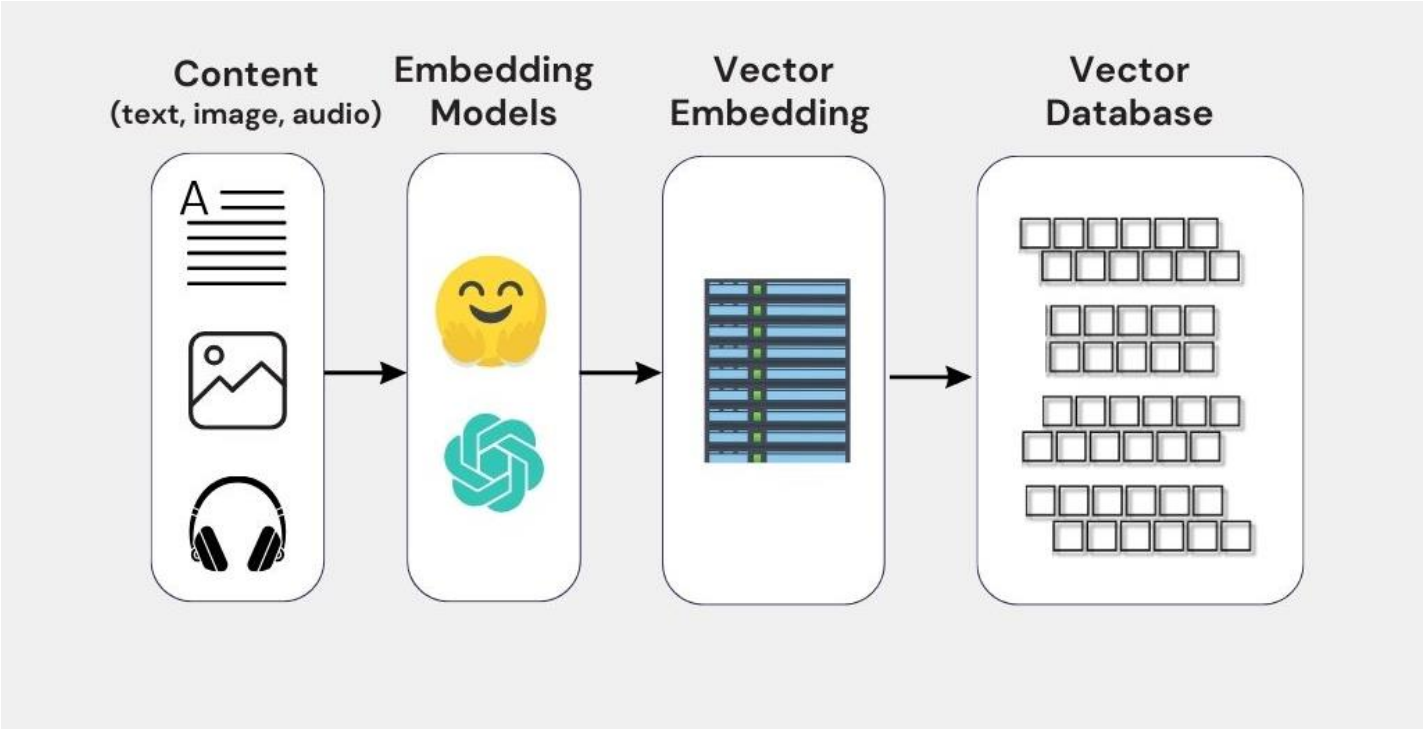
Cloud Vector Database Setup

What is a Vector Database?

A specialized database optimized for



Vector Database



Vector Search Problem

Finding similar vectors is challenging at scale

Exact search (brute force): O(n) comparisons

For 1M vectors: 1,000,000 comparisons per query!

Need specialized algorithms for efficiency

How Vector Search Works

Approximate Nearest Neighbors (ANN)

Makes vector search scalable through approximation

Tradeoff: Search speed vs. result accuracy

Graph

HNSW

(Hierarchical Navigable Small World)

Graph-based navigation

Cluster

IVF

(Inverted File Index)

Cluster-based coarse quantization

Vector

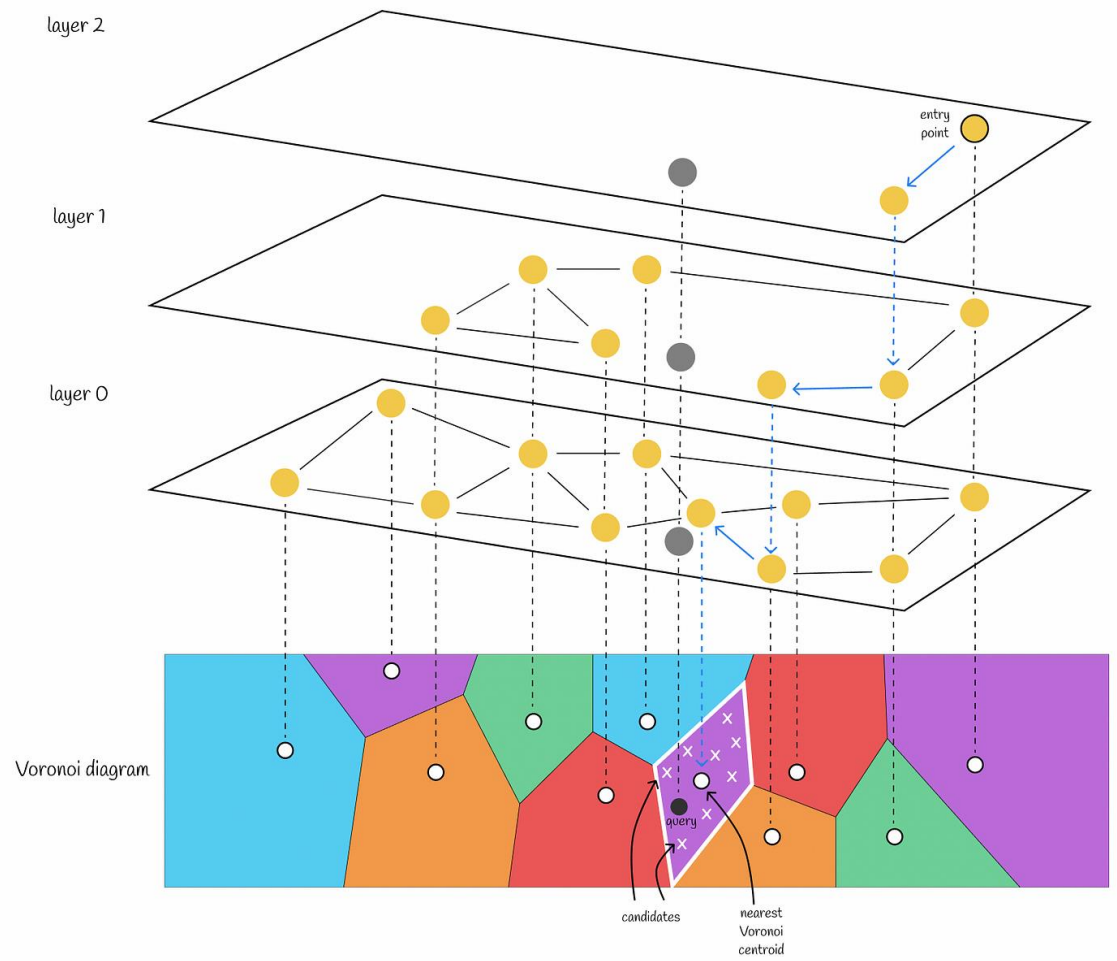
PQ

(Product Quantization)

Vector compression

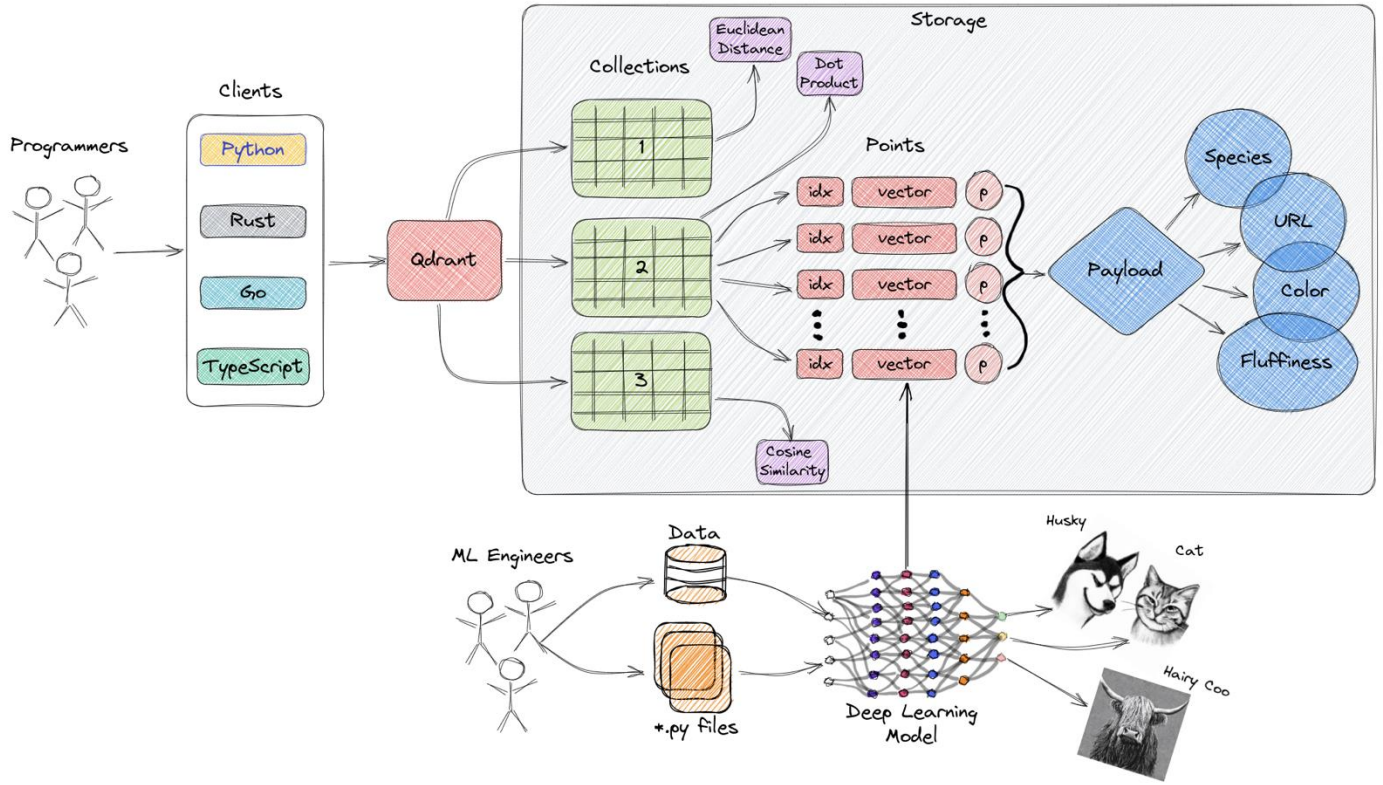
Inside HNSW Graph

- Multiple layers of connectivity
- Dense connections in bottom layer
- Sparse connections in upper layers
- Navigation from top to bottom Logarithmic search complexity

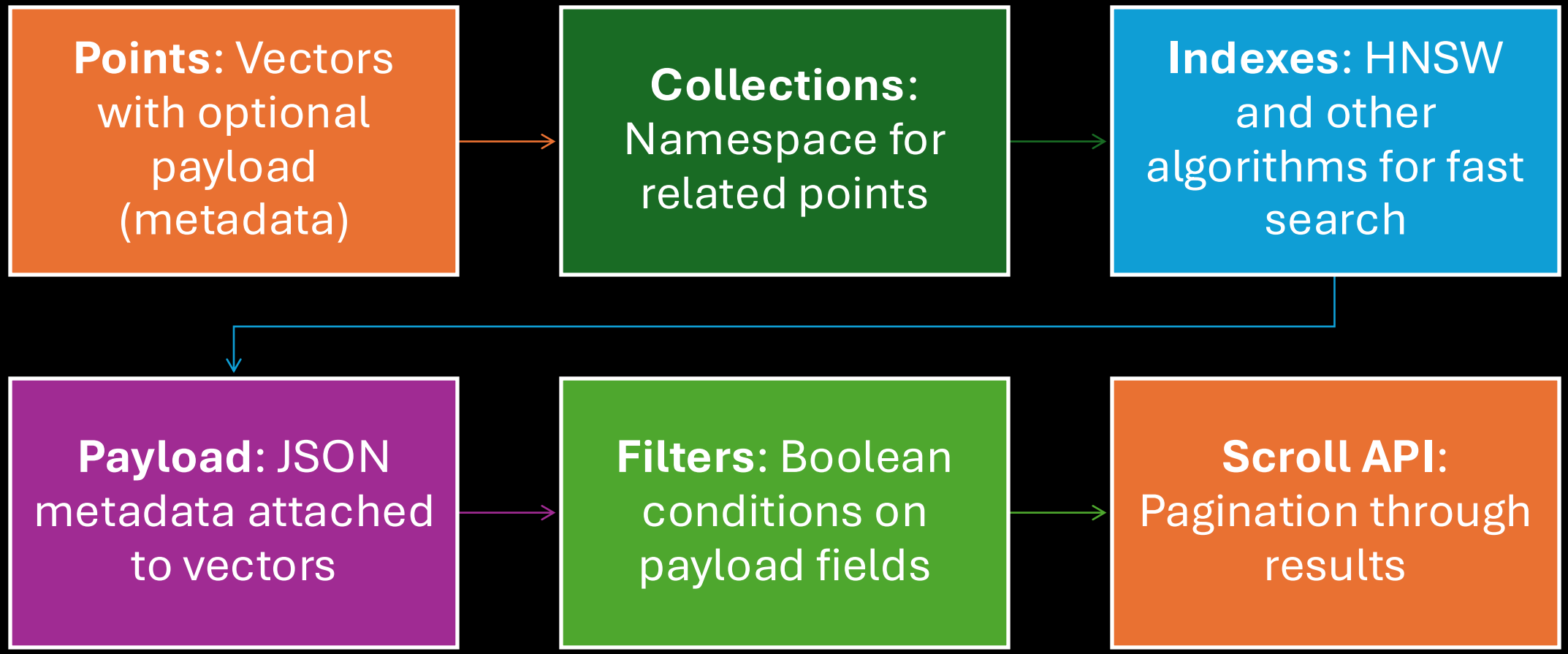


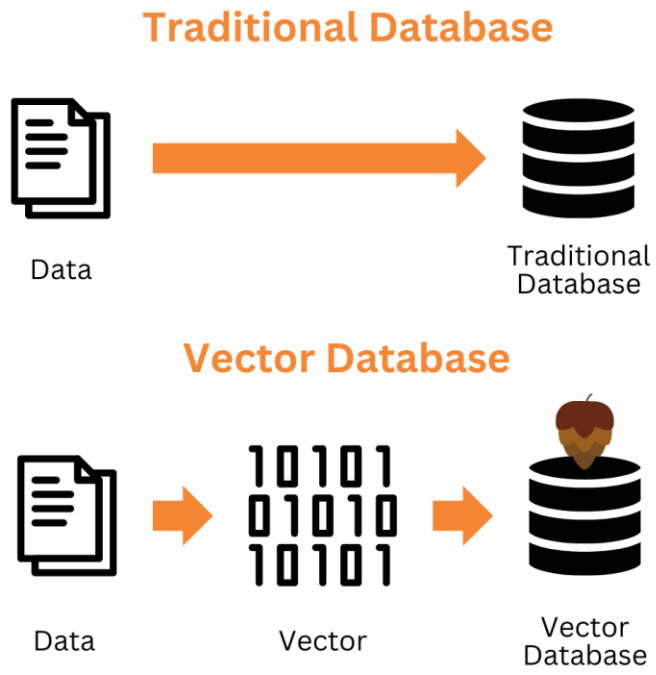
Qdrant Vector Database

- **High Performance**: Optimized for vector search
- **Rich Filtering**: Combine vector and metadata filtering
- **Cloud-Native**: Built for distributed environments
- **Payload Storage**: Store vectors with associated metadata
- **Multi-Tenancy**: Collections for different use cases



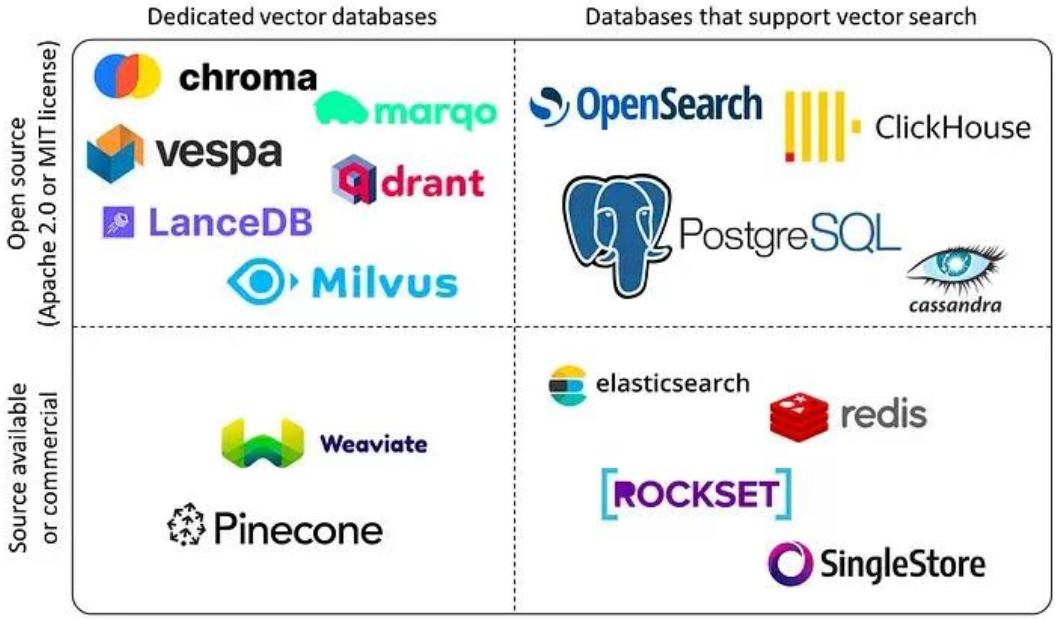
Qdrant Key Concepts





Feature	Vector Database	Traditional DB
Data Type	Vectors (embeddings)	Structured records
Search	Similarity-based	Exact match/range
Indexing	ANN algorithms (HNSW)	B-trees, hash indexes
Query Language	Vector + filters	SQL/NoSQL queries
Use Case	Semantic search	Structured data retrieval

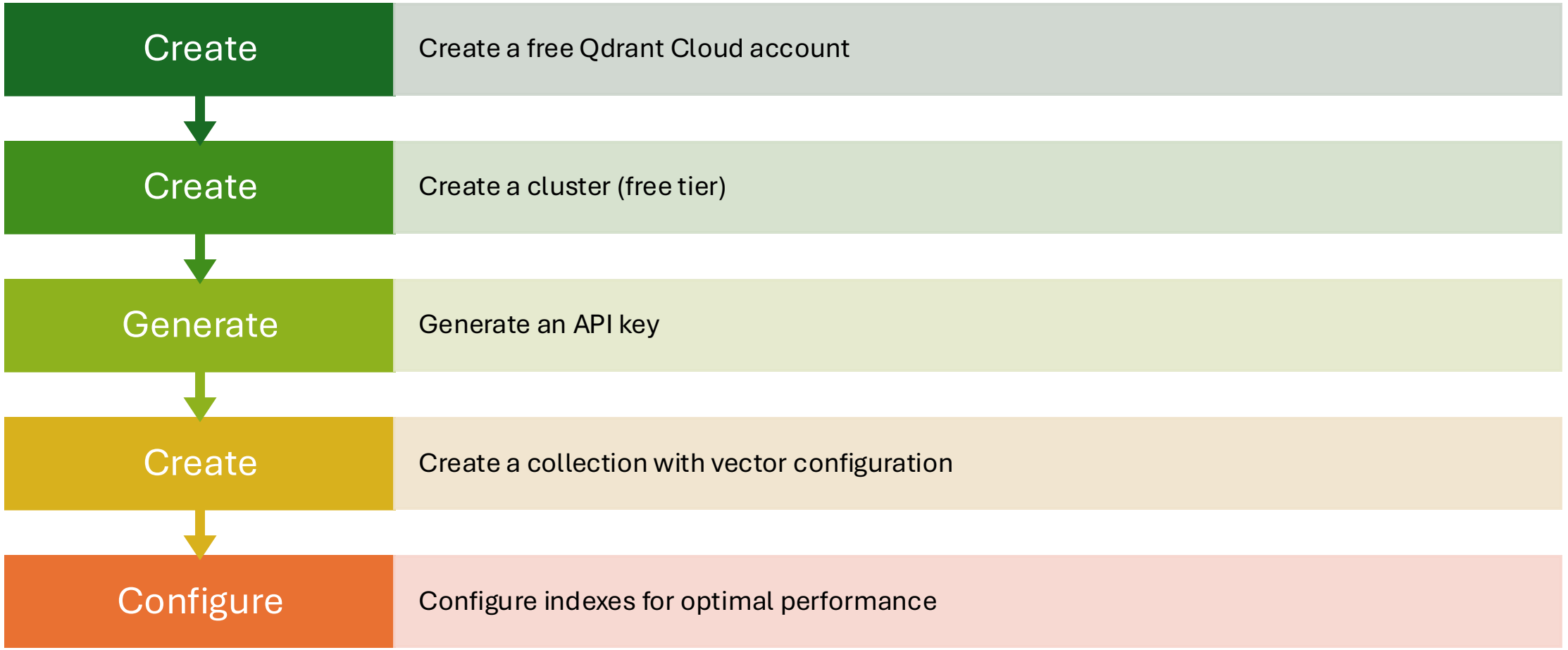
Vector DB vs. Traditional DB



Database	Type	Key Features	Free Tier
Qdrant	Dedicated Vector DB	Strong filtering, HNSW	1GB, 3 indices
Pinecone	Dedicated Vector DB	Serverless, namespaces	Limited pods
Weaviate	Multi-modal DB	Multi-modal, modules	1GB storage
Milvus	Distributed Vector DB	Scalable, hybrid	Self-hosted only
Supabase pgvector	PostgreSQL extension	SQL integration	Included in Supabase

Cloud Vector Database Options

Setting Up Qdrant Cloud



Qdrant Collection Configuration

Vector size: Must match embedding dimension

Distance: Cosine, Euclid, or Dot Product

HNSW parameters:

- m: Max connections per node (16-64)
- ef_construct: Index build quality (100-500)
- ef_search: Search quality (higher = more accurate)

Let's Set Up Our Vector Database!

Now we'll implement

01

Connect to Qdrant Cloud

02

Create a properly configured collection

03

Insert and retrieve vectors

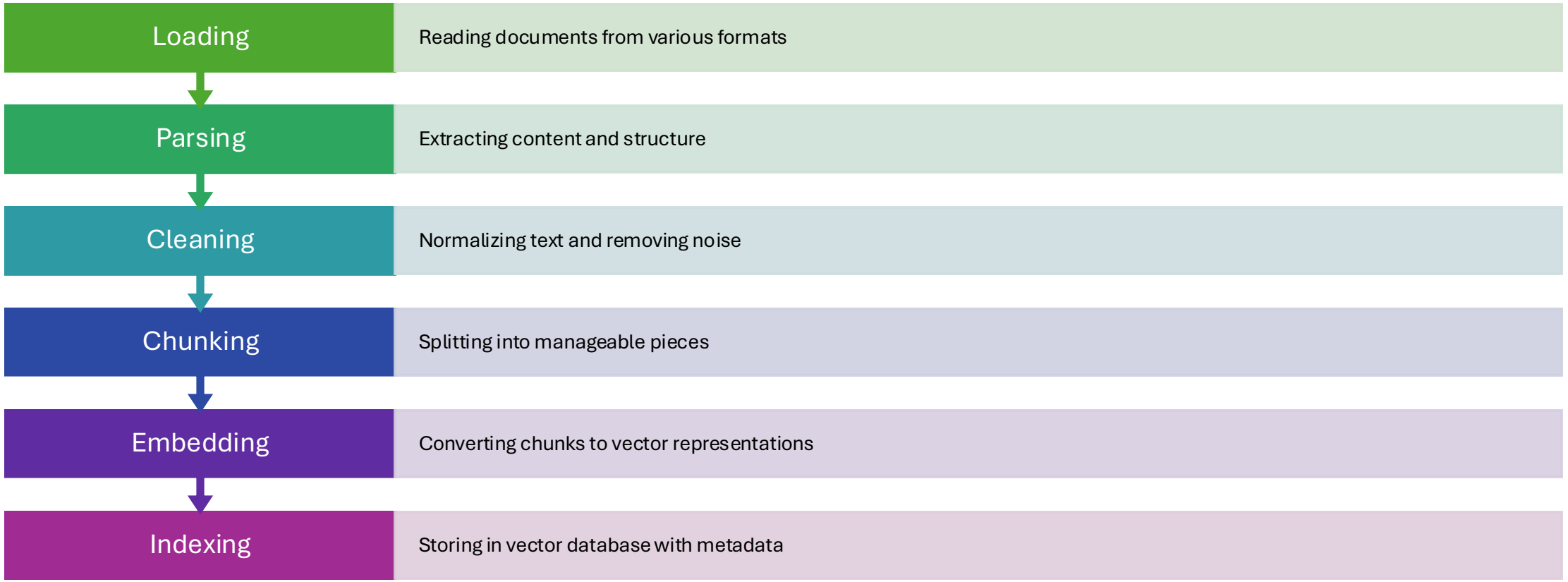
04

Test basic search functionality

Time to switch to Colab for Qdrant implementation!

Document Processing Pipeline

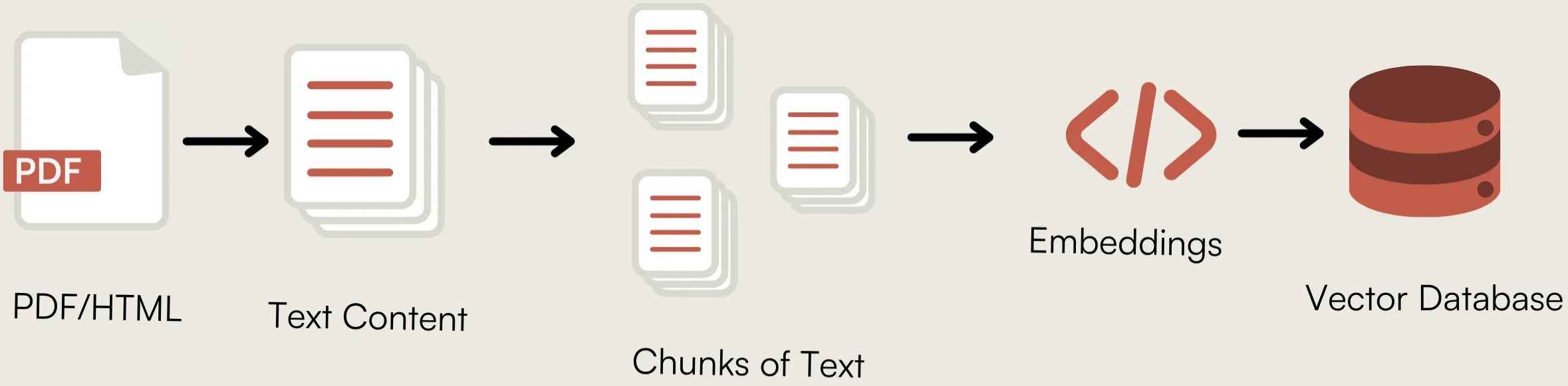
Document Processing Pipeline



Document Formats & Extraction

- **PDFs**: PyPDF2, pdf2image + OCR, pdfplumber
- **HTML**: BeautifulSoup, html2text
- **Word/Office**: python-docx, mammoth
- **Images**: OCR (Tesseract, Google Vision)
- **Audio/Video**: Transcription APIs

Chunking for RAG



The Chunking Problem

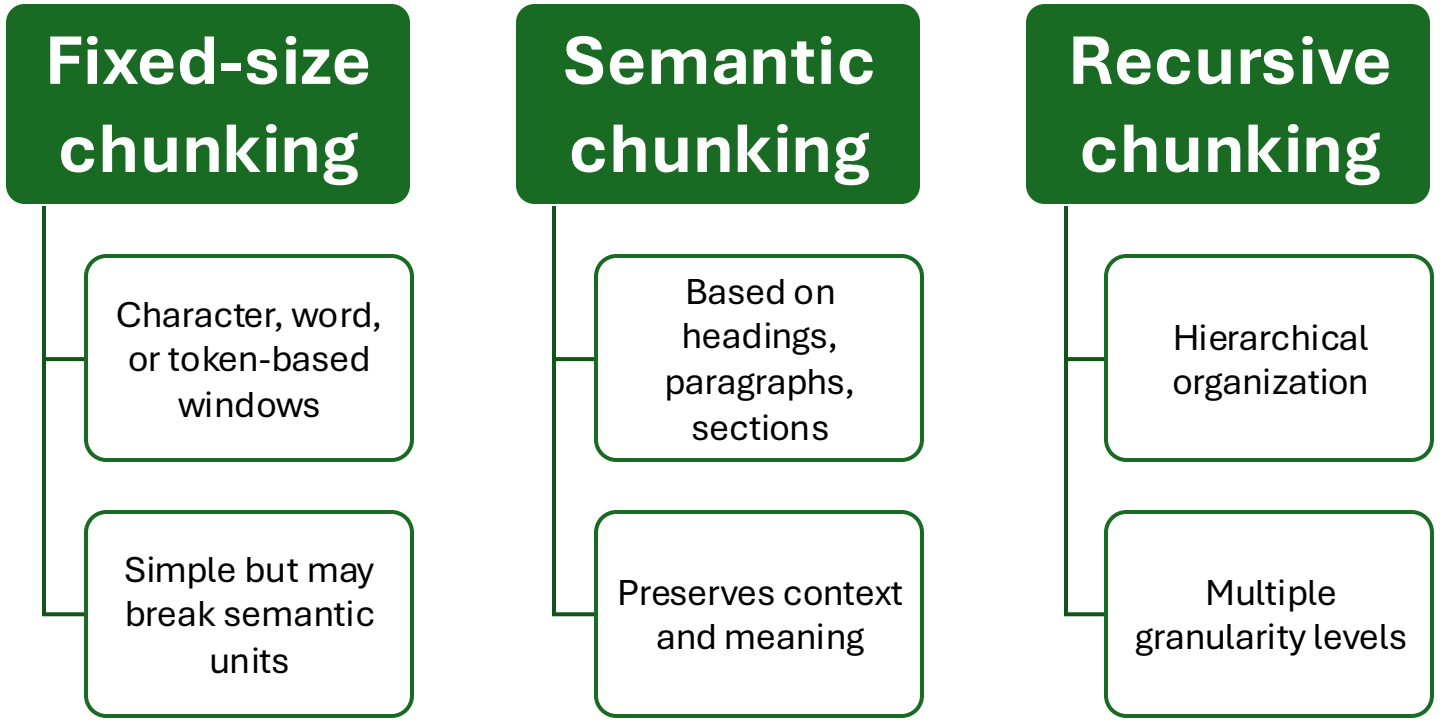
Why chunking matters:

- LLM context windows are limited (4K-32K tokens)
- Retrieval needs focused, relevant passages
- Too small: lacks context; Too large: dilutes relevance

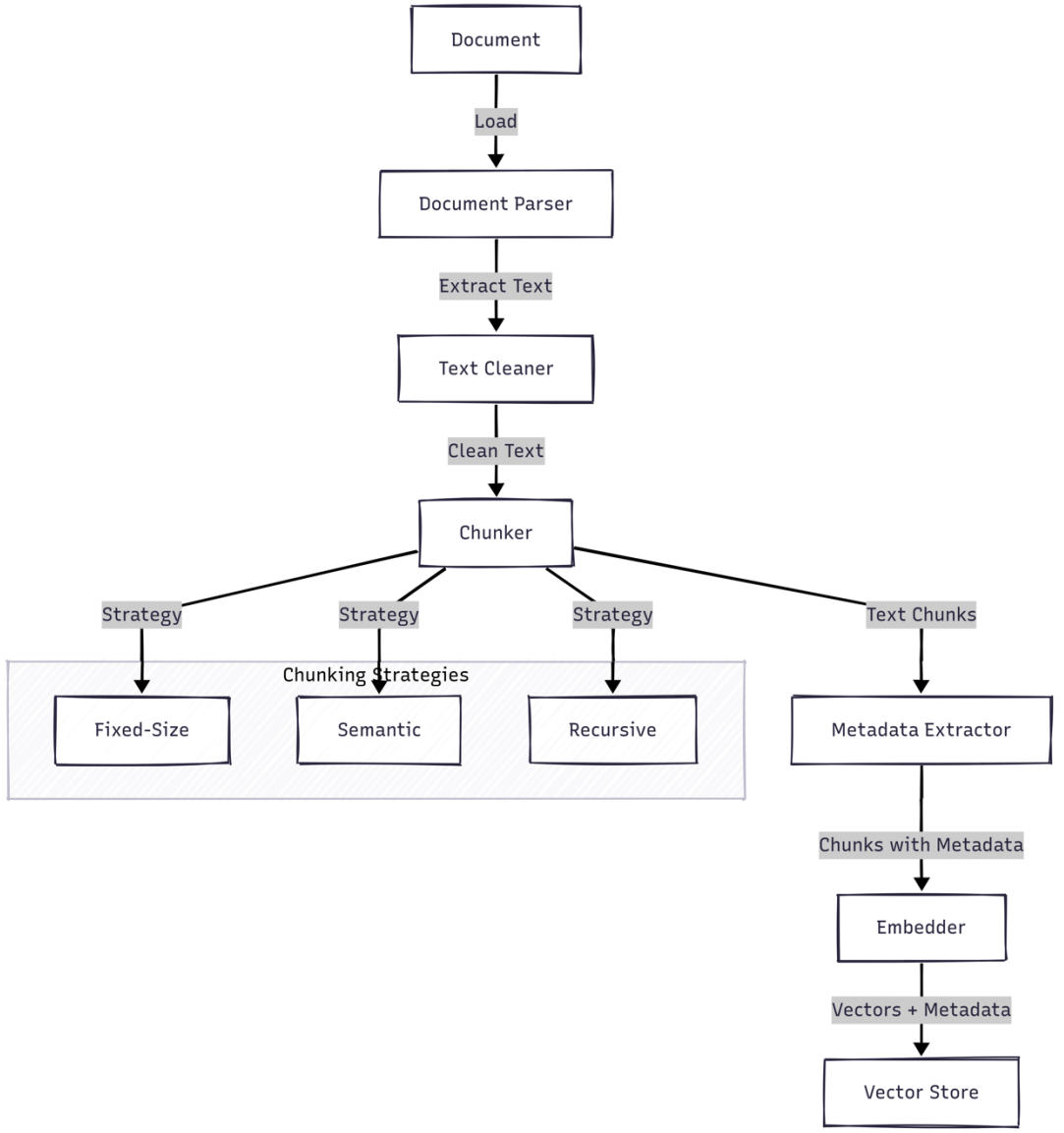
Chunks must be:

- **Self-contained**: Makes sense in isolation
- **Properly sized**: Fits retrieval and LLM needs
- **Semantically coherent**: Preserves meaning

Chunking Strategies



Chunking Strategies



Chunking Best Practices

Overlap between chunks (10-20%)

- Prevents context loss at boundaries
- Enables cross-reference between chunks

Preserve document structure

- Headers with content
- Keep lists together
- Maintain code blocks intact

Consistent sizing

- Token-based (not character/word)
- Optimized for embedding model and LLM

Metadata Extraction & Utilization

Source

Document name, URL, path

Position

Location within document

Structure

Section, chapter, heading

Dates

Creation, modification dates

Authors

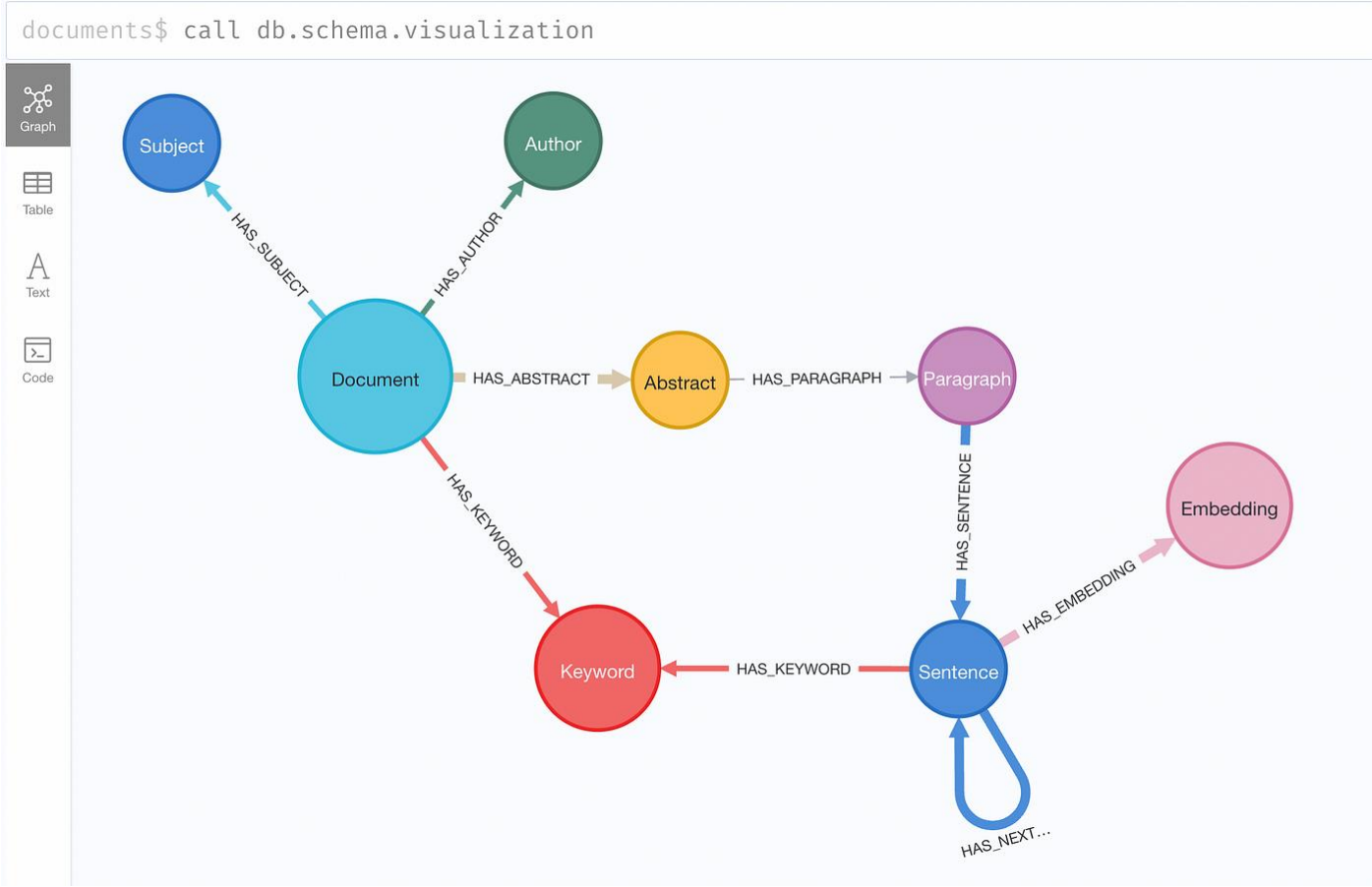
Creator information

Domain-specific

Categories, tags, etc.

Document Graph Relationships

- **Sequential**: Next/previous chunks
- **Hierarchical**: Parent/child sections
- **References**: Citations, links between documents
- **Semantic**: Related by topic or concept



Token Counting & Optimization

Why token counting matters:

- LLM pricing is per token
- Context windows are token-limited
- Different tokenizers produce different counts

Tokenization approaches:

- GPT Tokenizers (tiktoken)
- Sentence-piece
- WordPiece

Advanced Processing Techniques

1

Text classification

- Categorize documents/chunks by topic
- Identify document types

2

Entity extraction

- Find people, organizations, locations, dates
- Extract key concepts

3

Summary generation

- Create summaries for long documents
- Use as additional context

Let's Build a Document Processor!

- Document loading from various formats
- Chunking strategies
- Metadata extraction
- Embedding and storage in Qdrant

Time to switch to Colab for document processing implementation!

LLM Integration with Cloud Providers

Language Model Landscape

- **Proprietary**: GPT-4, Claude, PaLM, Gemini
- **Open Source**: Llama 2/3, Mistral, Falcon
- **Specialized**: Code models, multilingual, domain-specific

Hosted LLM Providers

- **OpenAI**: GPT-3.5/4 models, high quality but costly
- **Anthropic**: Claude models, long context
- **OpenRouter**: Multi-provider gateway, simplified access
- **TogetherAI**: Open models, cost-effective
- **Cohere**: Specialized for enterprise, languages
- **Model hosting services**: Replicate, HuggingFace, etc.

OpenRouter: Multi-Provider Gateway

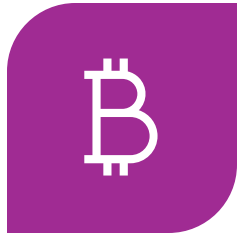
- **Single API**: Access multiple providers
- **Model discovery**: Browse available models
- **Fallbacks**: Automatic failover between providers
- **Cost optimization**: Route to cheaper providers
- **Free models**: Free models available for testing

Model Selection Factors

CONTEXT WINDOW: HOW MUCH TEXT CAN BE PROCESSED (4K-128K TOKENS)

QUALITY: REASONING, INSTRUCTION FOLLOWING, ACCURACY

LATENCY: RESPONSE TIME REQUIREMENTS



COST: PRICE PER 1K TOKENS (INPUT/OUTPUT)

SPECIALIZATION: DOMAIN EXPERTISE (CODE, MEDICAL, LEGAL)

RAG-Specific Model Requirements

Context utilization: Efficiently use retrieved information

Citation ability: Reference sources correctly

Focused responses: Stay on topic with retrieved facts

Admission of ignorance: Say "I don't know" when appropriate

Hallucination resistance: Avoid making up facts

Prompt Engineering for RAG

- **System instructions**: RAG-specific guidelines
- **Retrieved context**: Formatted context passages
- **User query**: Original question
- **Format instructions**: Output structure requirements
- **Reasoning guidance**: Step-by-step thinking instructions

RAG Prompt Templates

Basic template

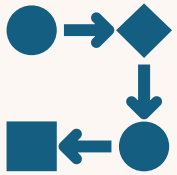
Context + Question + Answer format

Simple but may not use context effectively

Instruction-heavy template

Explicit rules for using context

Better for preventing hallucinations



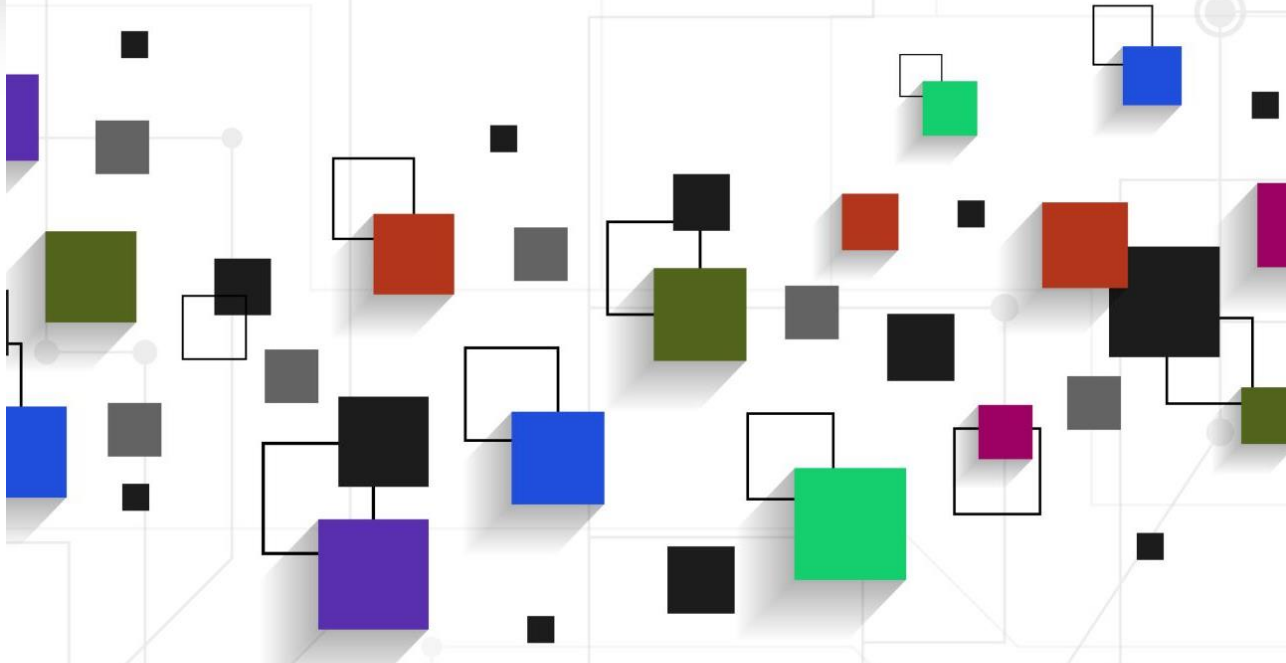
Chain-of-thought template

Guides the model through reasoning steps

Better for complex questions

Context Formatting Techniques

- **Chunked format**: Separate numbered passages
- **Metadata inclusion**: Source, date, author with each chunk
- **Relevance markers**: Indicate passage relevance scores
- **Hierarchical**: Organize by relevance or topic



LLM API Integration

Authentication: API keys, tokens

Request formatting: JSON structure

Streaming: Real-time token streaming vs. complete responses

Error handling: Timeouts, rate limits, content policy rejections

Cost tracking: Token counting and expense monitoring

Let's Integrate LLMs!

API connections to OpenRouter and TogetherAI

Effective RAG prompt templates

Response handling and formatting

Error handling and fallbacks

Time to switch to Colab for document processing implementation!

Building the Retrieval Component

Retrieval System Architecture

Query Processor: Prepare and expand queries

Retriever: Find relevant documents/passages

Ranker: Score and order retrieved results

Filter: Apply metadata constraints

Result Processor: Format for LLM consumption

The Retrieval Problem

Vocabulary mismatch: Different words, same meaning

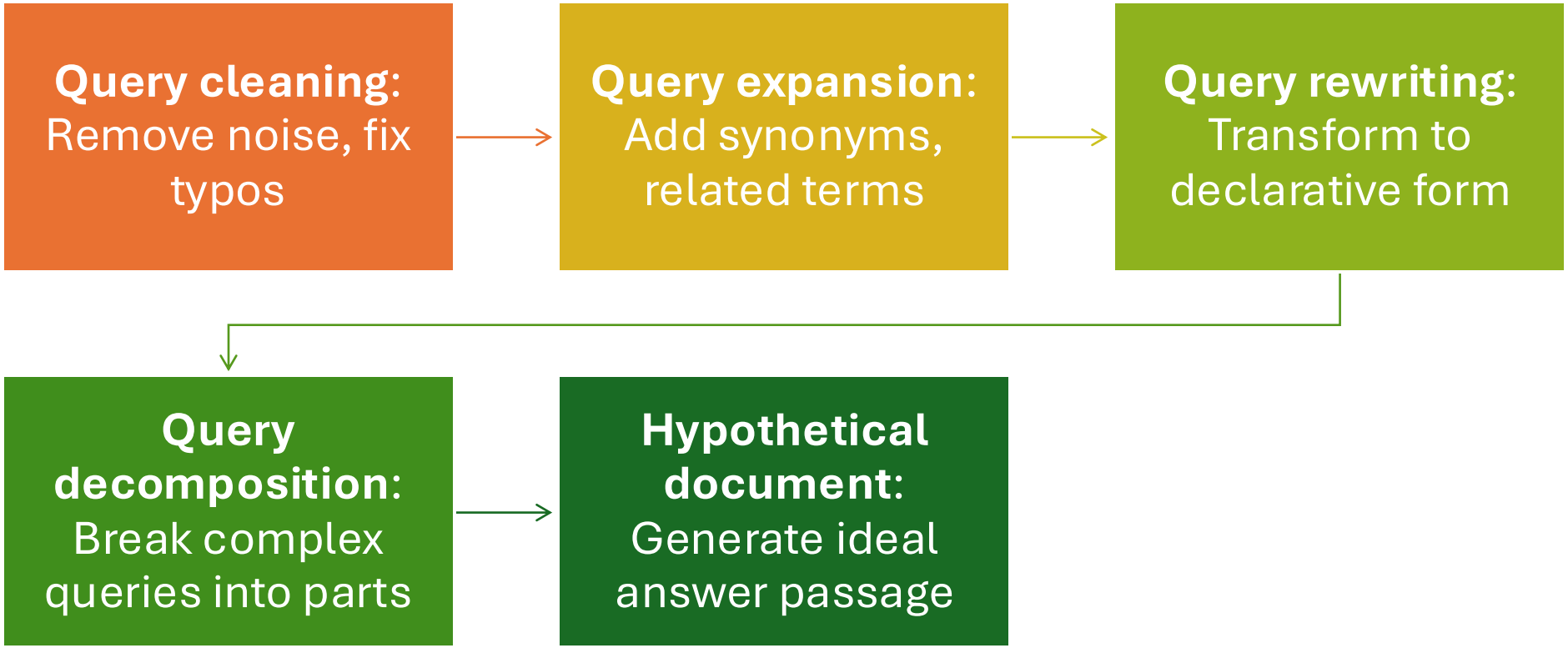
Intent understanding: What is the user really asking?

Ranking relevance: Most important info first

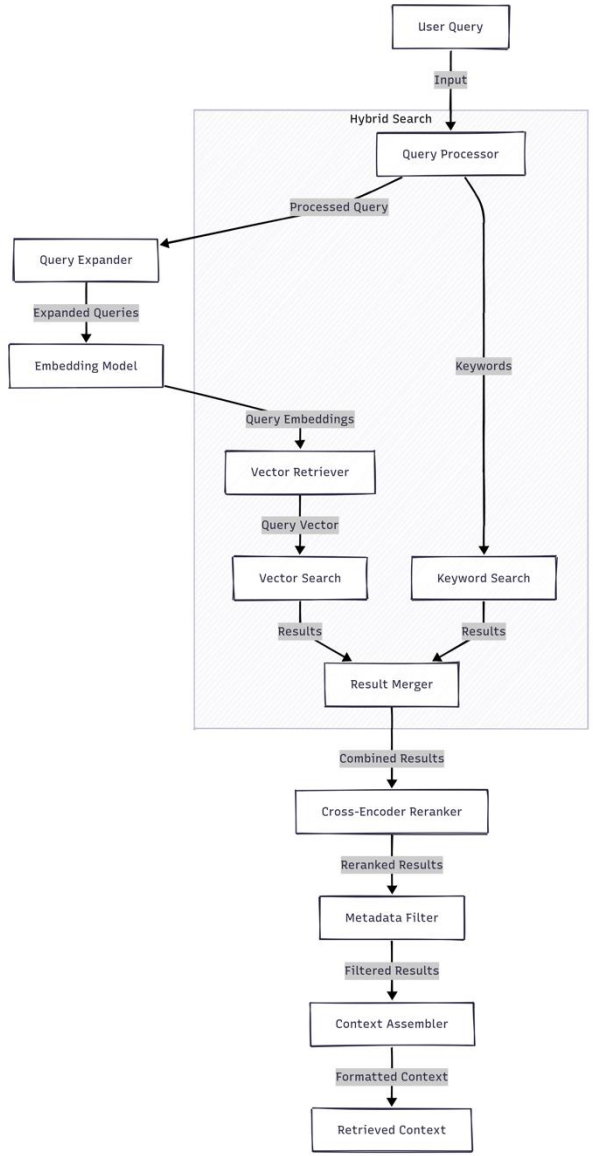
Contextual importance: What's relevant depends on context

Diversity vs. precision: Coverage vs. exactness tradeoff

Query Processing Techniques



Retrieval Component



Dense vs. Sparse Retrieval

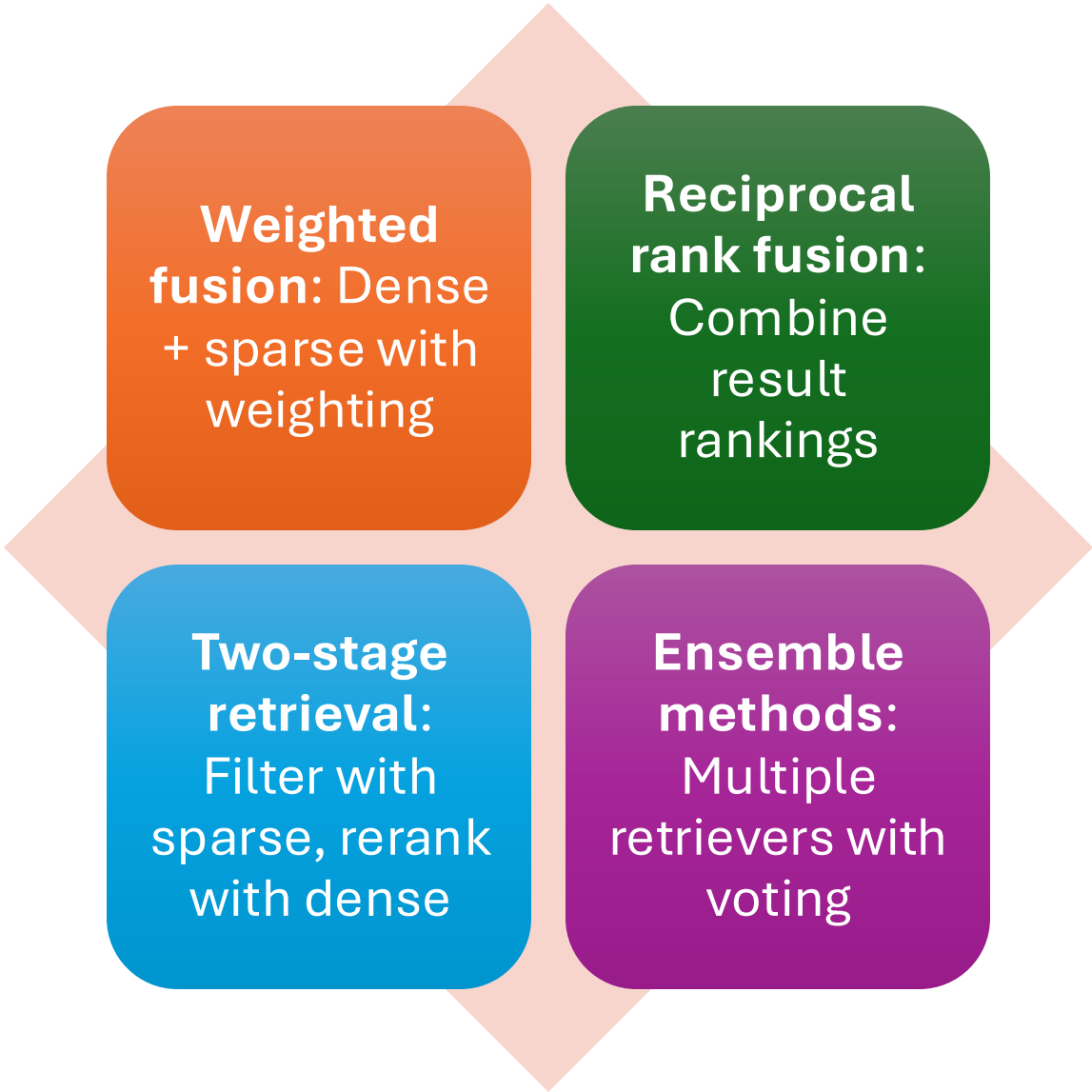
Sparse Retrieval (TF-IDF, BM25)

- Keyword-based, explicit term matching
- Good for exact terminology, proper nouns

Dense Retrieval (Embeddings)

- Semantic understanding, concept matching
- Good for conceptual relationships, synonyms

Hybrid Search Approaches



Cross-Encoder Reranking

Bi-encoder (embedding model): Query & document encoded separately

Cross-encoder: Processes query+document pair together

Much more accurate relevance scoring

Computationally expensive (can't precompute)

Applied to top-k initial results only

Advanced Filtering Techniques

- **Source filtering**: Specific documents, websites, authors
- **Recency filtering**: Time-based relevance
- **Format filtering**: Document types (PDF, code, etc.)
- **Hierarchical filtering**: Sections, chapters, categories
- **Domain-specific attributes**: Custom metadata fields

Optimizing Retrieval Parameters

- **k value**: Number of documents to retrieve
- **Similarity threshold**: Minimum relevance score
- **Hybrid weights**: Dense vs. sparse importance
- **Reranker confidence**: When to trust reranking
- **Context window utilization**: Optimal content amount

Contextual Compression

Document trimming: Remove irrelevant parts

Extractive summarization: Pull key sentences

Answer extraction: Isolate most relevant passages

Abstractive compression: Rewrite to be more concise

Let's Build an Advanced Retriever!

QUERY PREPROCESSING

HYBRID SEARCH WITH VECTOR + KEYWORD METHODS

CROSS-ENCODER RERANKING

METADATA FILTERING

Time to switch to Colab for retrieval implementation!

Full RAG Integration & Supabase Backend

Complete RAG System Architecture

User Interface: Query input and response display

Backend API: Request handling and orchestration

Retrieval Engine: Finding relevant documents

Generation Engine: Creating responses with LLMs

Monitoring & Logging: Performance tracking

Supabase as a Backend Solution

OPEN-SOURCE FIREBASE ALTERNATIVE

POSTGRESQL DATABASE WITH REST/GRAPHQL APIS

AUTHENTICATION, STORAGE, AND SERVERLESS FUNCTIONS

REAL-TIME CAPABILITIES

GENEROUS FREE TIER (500MB DATABASE, 1GB FILE STORAGE)

Supabase Key Features for RAG

Database: Store conversations, user preferences, logs

Authentication: Secure user management

Edge Functions: Serverless API endpoints

Storage: Document storage and management

Realtime: Live updates for responsive UI

pgvector: Native vector operations (alternative to Qdrant)

RAG Database Schema Design

- **Users**: User profiles and preferences
- **Conversations**: Conversation sessions
- **Messages**: Individual messages in conversations
- **Documents**: Uploaded document metadata
- **Cache**: Response and embedding cache
- **Feedback**: User feedback for answers

Conversation State Management

Conversation history: Previous Q&A pairs

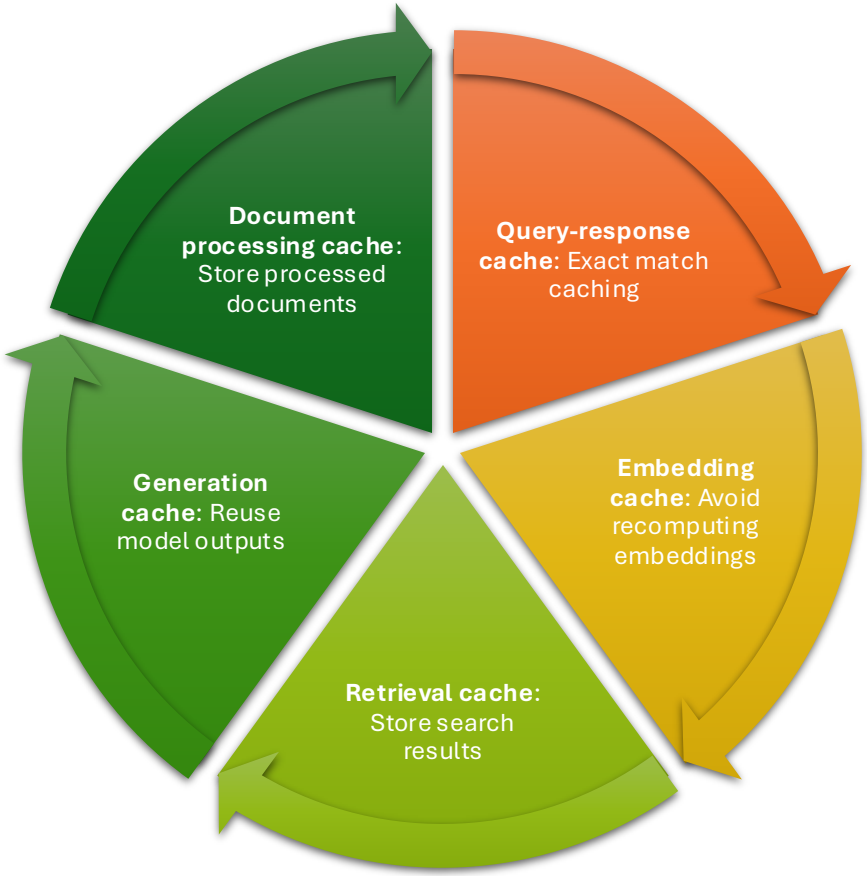
Retrieved context tracking: What's been shown to the model

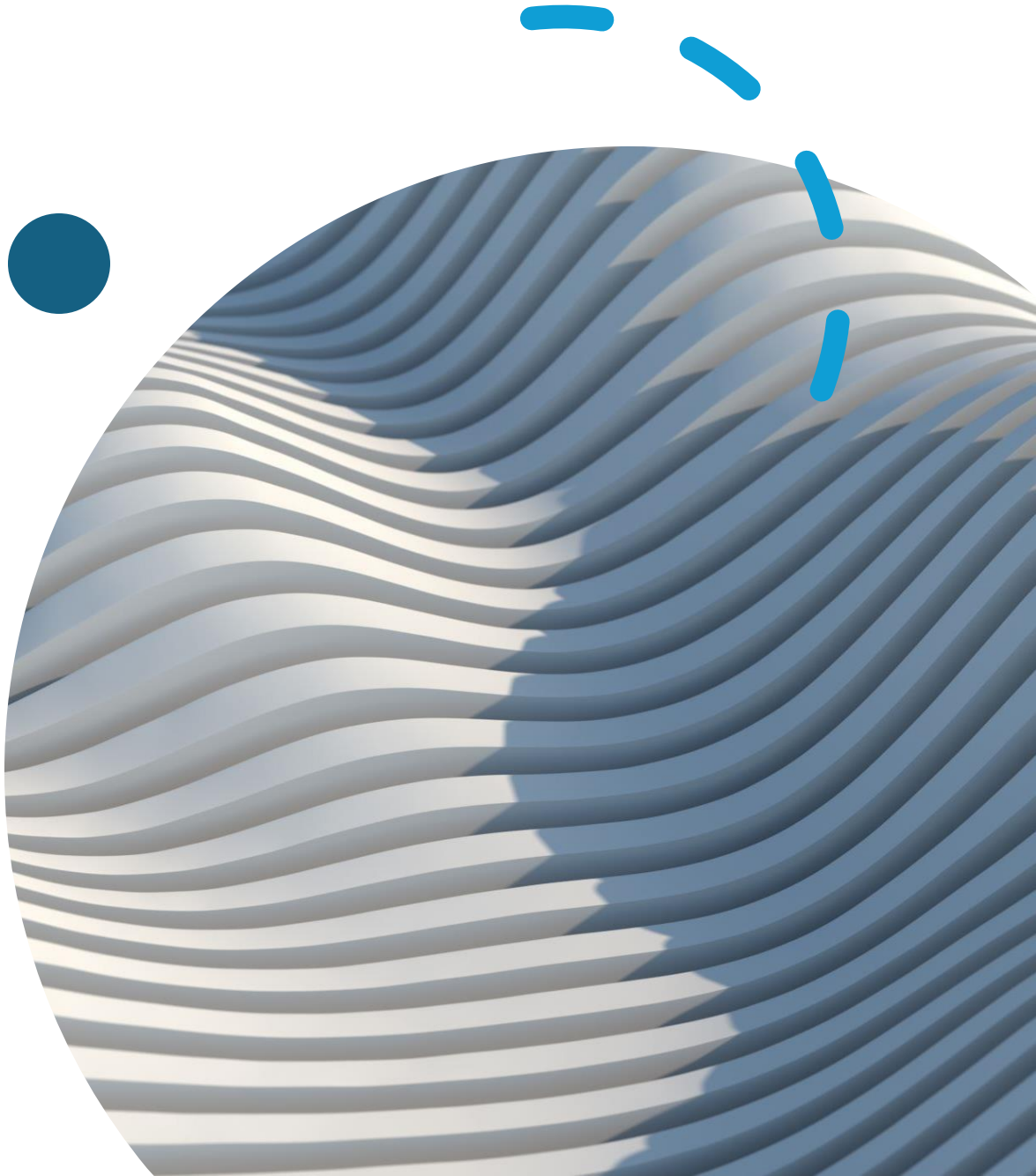
User preferences: Personalization options

System state: Mode, active features, etc.

Prior feedback: Learning from past interactions

Caching Strategies for RAG





Rate Limiting & Throttling

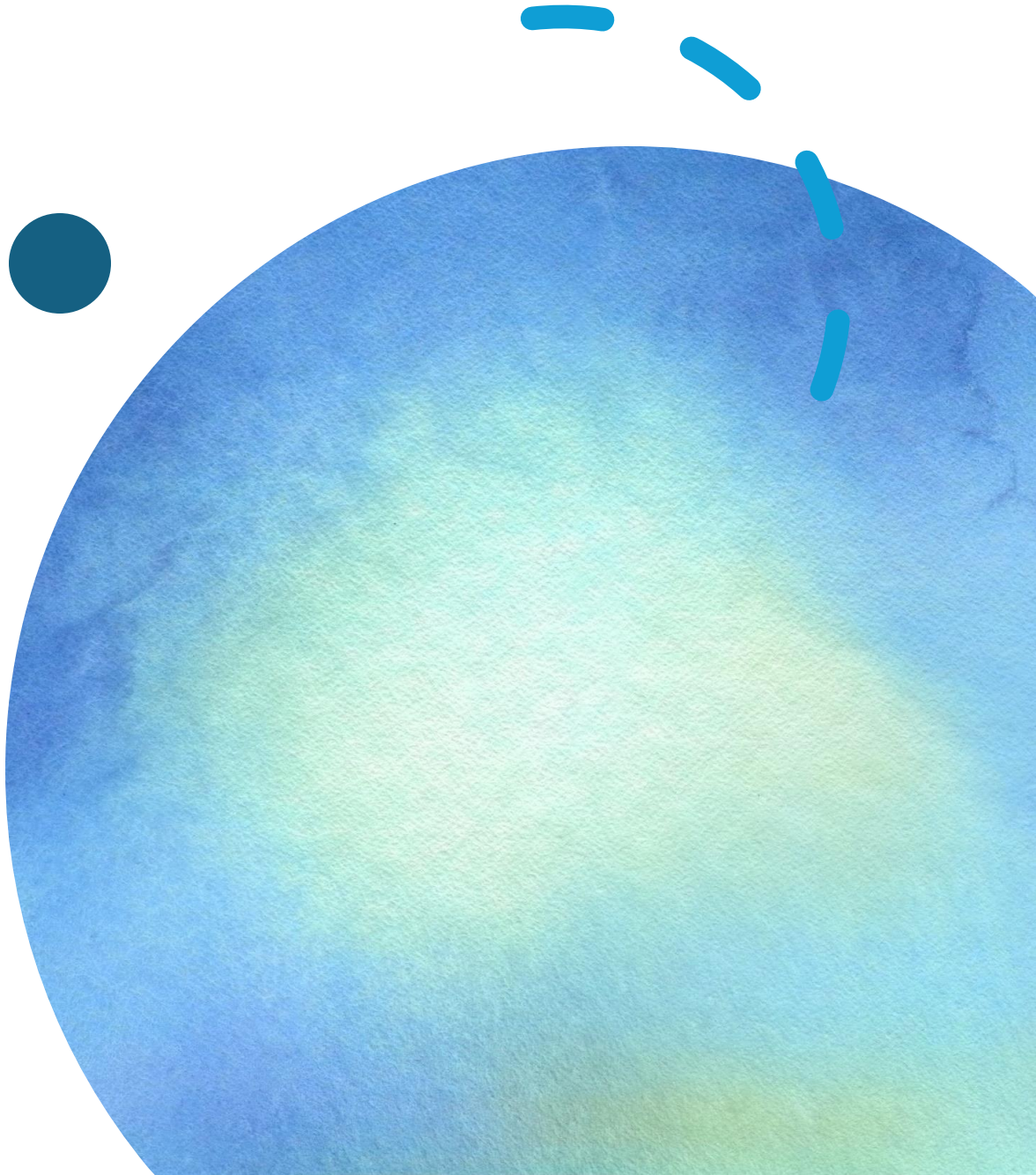
Request limits: Maximum requests per time period

Token budgeting: Allocation of LLM token usage

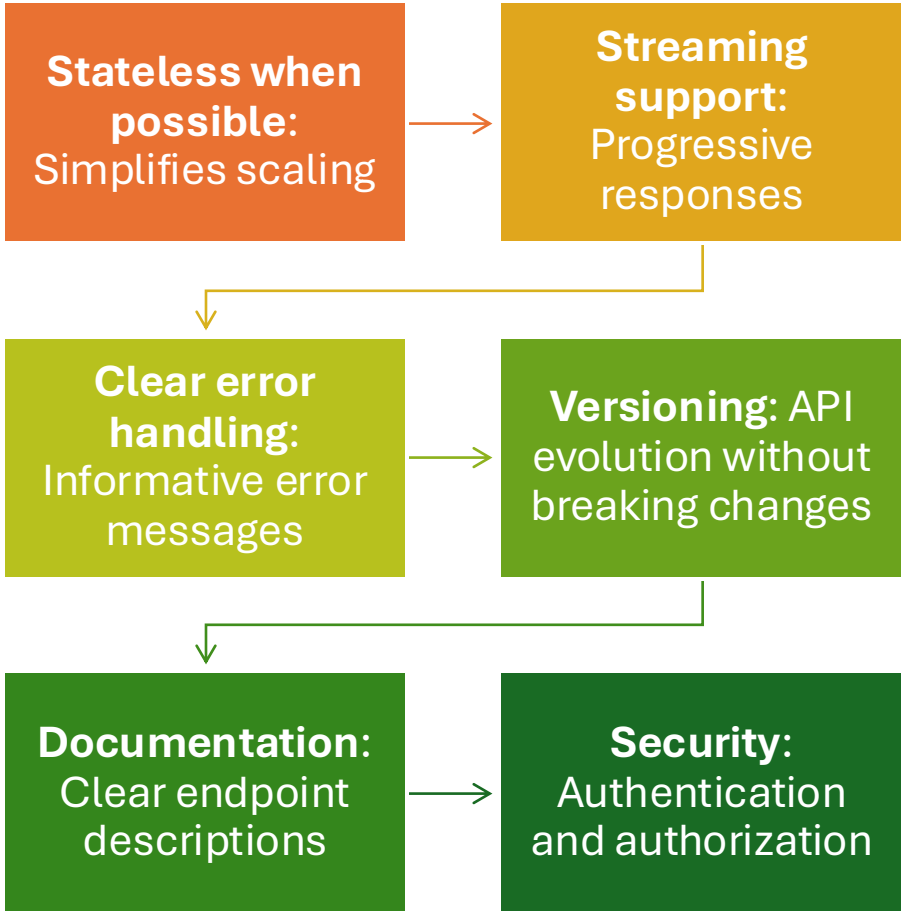
Prioritization: Handling concurrent requests

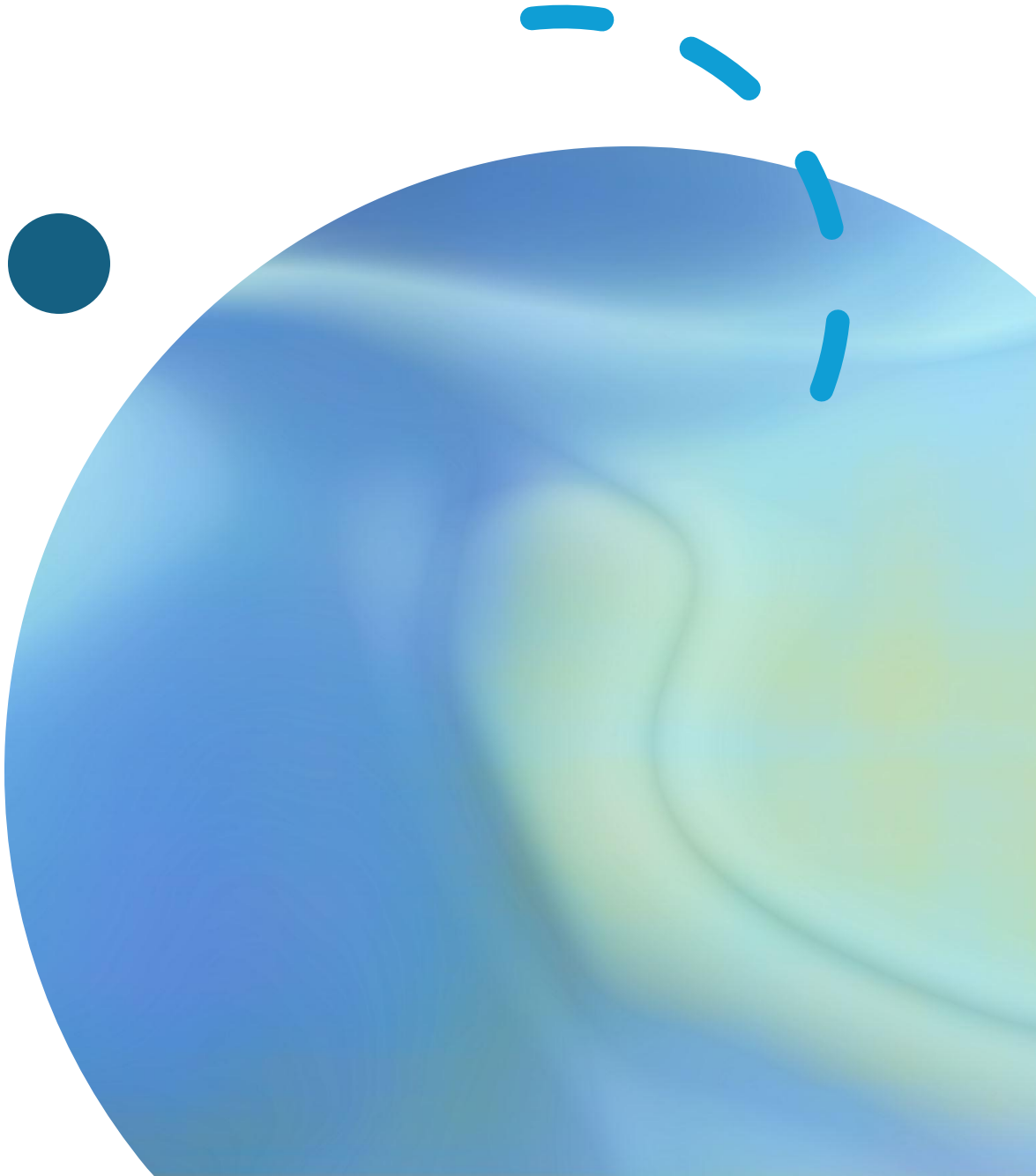
Graceful degradation: Fallbacks when limits reached

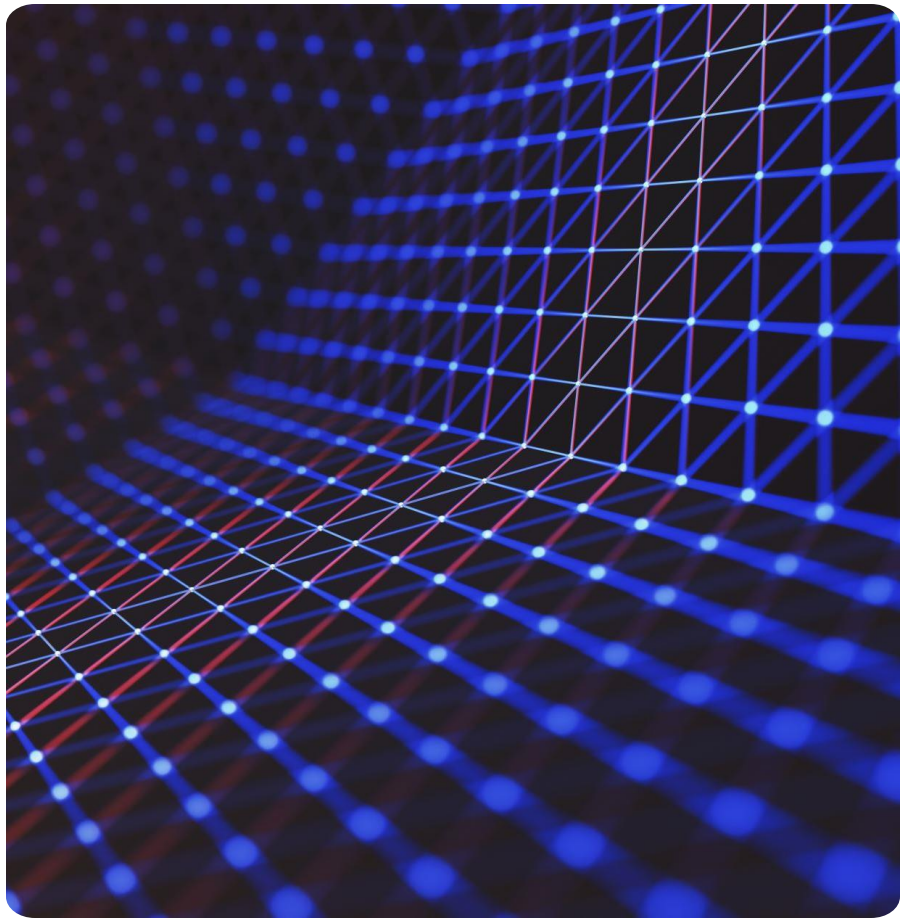
Cost awareness: Budget-based limitations



API Design Best Practices







Error Handling & Fallbacks

Service unavailability: Handle API downtime

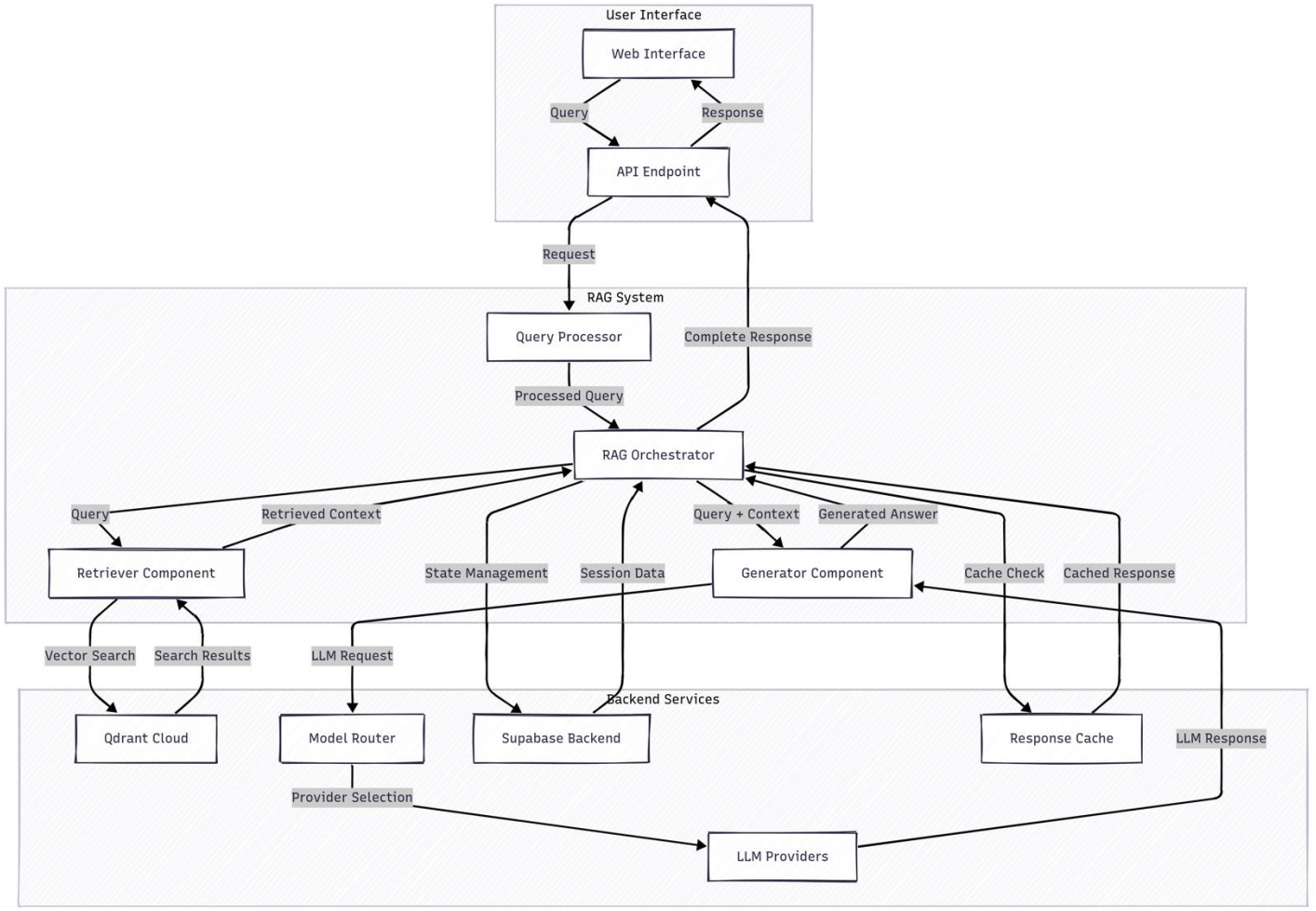
Rate limit exceeded: Graceful backoff strategies

Content moderation: Handle policy violations

Invalid queries: Detect and handle edge cases

Timeout management: Handle slow responses

Full RAG System Integration



Let's Integrate the Full RAG System!

- Supabase database setup
- End-to-end RAG pipeline
- Conversation management
- Caching and optimization

Time to switch to Colab for system integration!

RAG Evaluation & Optimization

Why Evaluate RAG Systems?

Retrieval quality: Finding the right documents

Answer accuracy: Generating correct responses

Relevance: Addressing the user's actual need

Hallucination rate: Fabricated or incorrect information

Performance: Latency, cost, and resource usage

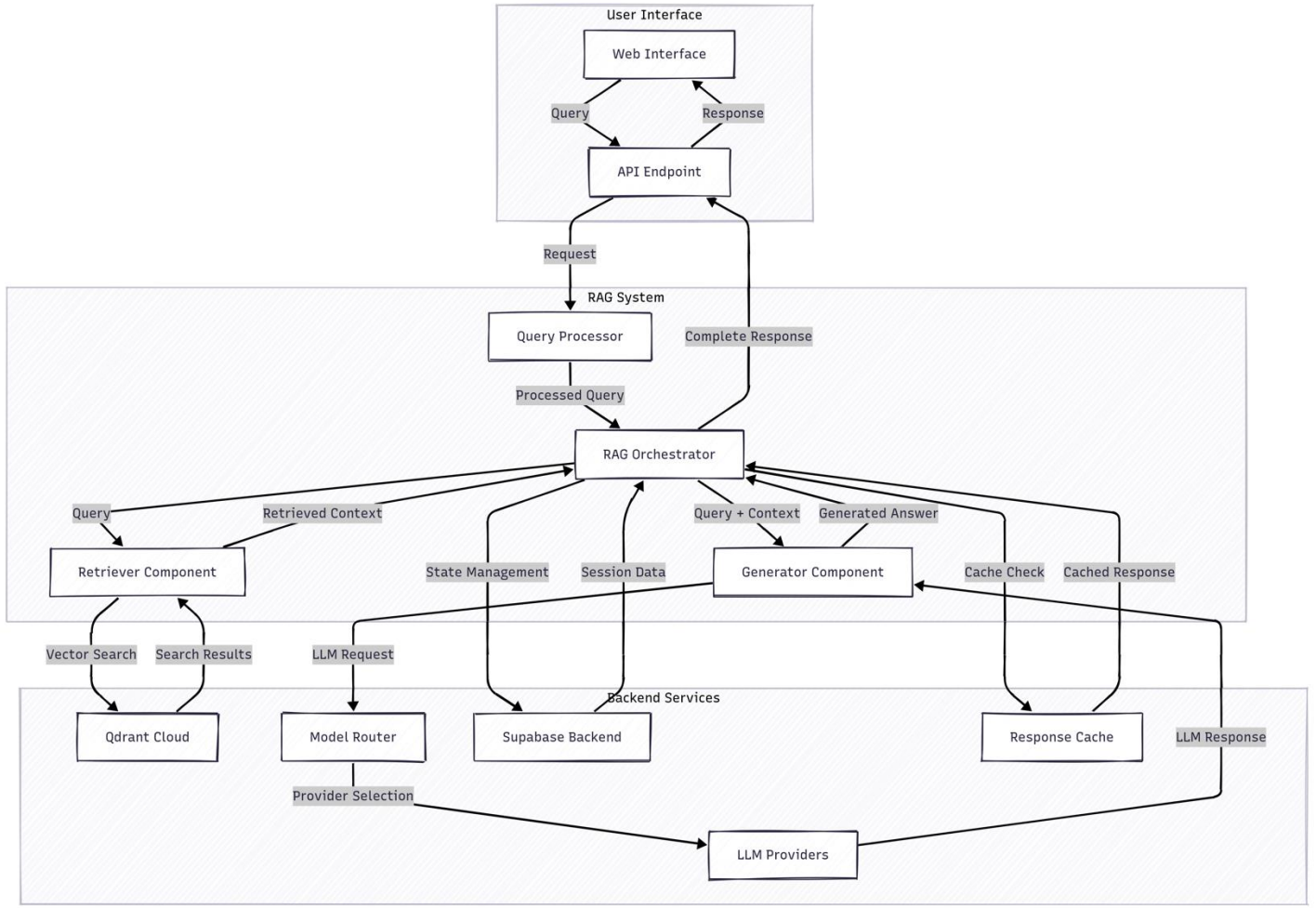
RAG Evaluation Framework

Retrieval evaluation:
Measuring document relevance

Generation evaluation:
Assessing answer quality

End-to-end evaluation:
Overall system performance

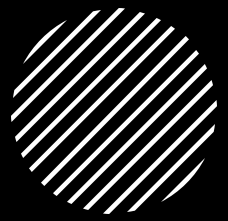
RAG Evaluation Framework



Retrieval Evaluation Metrics

- **Precision@k**: Fraction of retrieved documents that are relevant
- **Recall@k**: Fraction of relevant documents that are retrieved
- **Mean Reciprocal Rank (MRR)**: Position of first relevant result
- **Normalized Discounted Cumulative Gain (NDCG)**: Ranking quality
- **Mean Average Precision (MAP)**: Precision at each relevant result

Generation Evaluation Metrics



- **Faithfulness**: Factual accuracy and support from sources
- **Answer relevance**: Addresses the user's question
- **Coherence**: Logical flow and readability
- **Conciseness**: Appropriate length and focus
- **Hallucination detection**: Made-up information

RAG Assessment Framework

FAITHFULNESS: ANSWER ACCURACY BASED ON CONTEXT

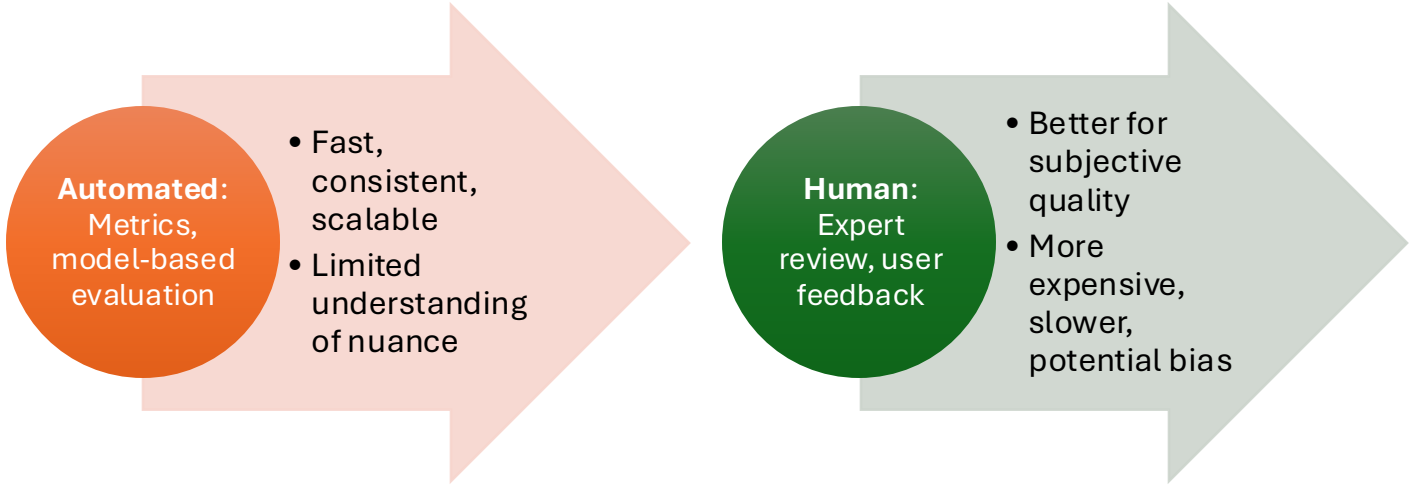
CONTEXT RELEVANCE: RETRIEVED CONTEXT QUALITY

CONTEXT RECALL: COVERAGE OF NECESSARY INFORMATION

ANSWER RELEVANCE: RELEVANCE TO THE QUESTION

ASPECT EVALUATION: DOMAIN-SPECIFIC METRICS

Automated vs. Human Evaluation



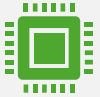
Performance Optimization Areas

Latency: Response time reduction

Cost: Token and API usage efficiency

Quality: Retrieval and generation improvements

Scalability: Handling more users/documents



Resource usage: Memory and computation needs

Latency Optimization Techniques

Caching: Store common queries and responses

Index optimization: Tune vector search parameters

Retrieval shortcuts: Tiered retrieval approaches

Smaller models: Balance quality vs. speed

Request parallelization: Concurrent operations

Streaming responses: Progressive output

Cost Optimization Strategies

Token efficiency: Optimal context formatting

Model selection: Cheaper models for simpler tasks

Caching: Reduce redundant API calls

Context pruning: Remove unnecessary information

Tiered approach: Use expensive models sparingly

Quality Improvement Techniques

- **Better chunking**: Improve document segmentation
- **Query reformulation**: Rewrite for better retrieval
- **Hybrid retrieval**: Combine multiple retrieval methods
- **Prompt optimization**: More effective LLM instructions
- **Feedback loops**: Learn from user interactions

Let's Evaluate and Optimize Our RAG System!

- Evaluation metrics and benchmarks
- Performance analysis
- System optimization techniques
- Final application improvements

Time to switch to Colab for evaluation and optimization!