Redis Released | AI workshop

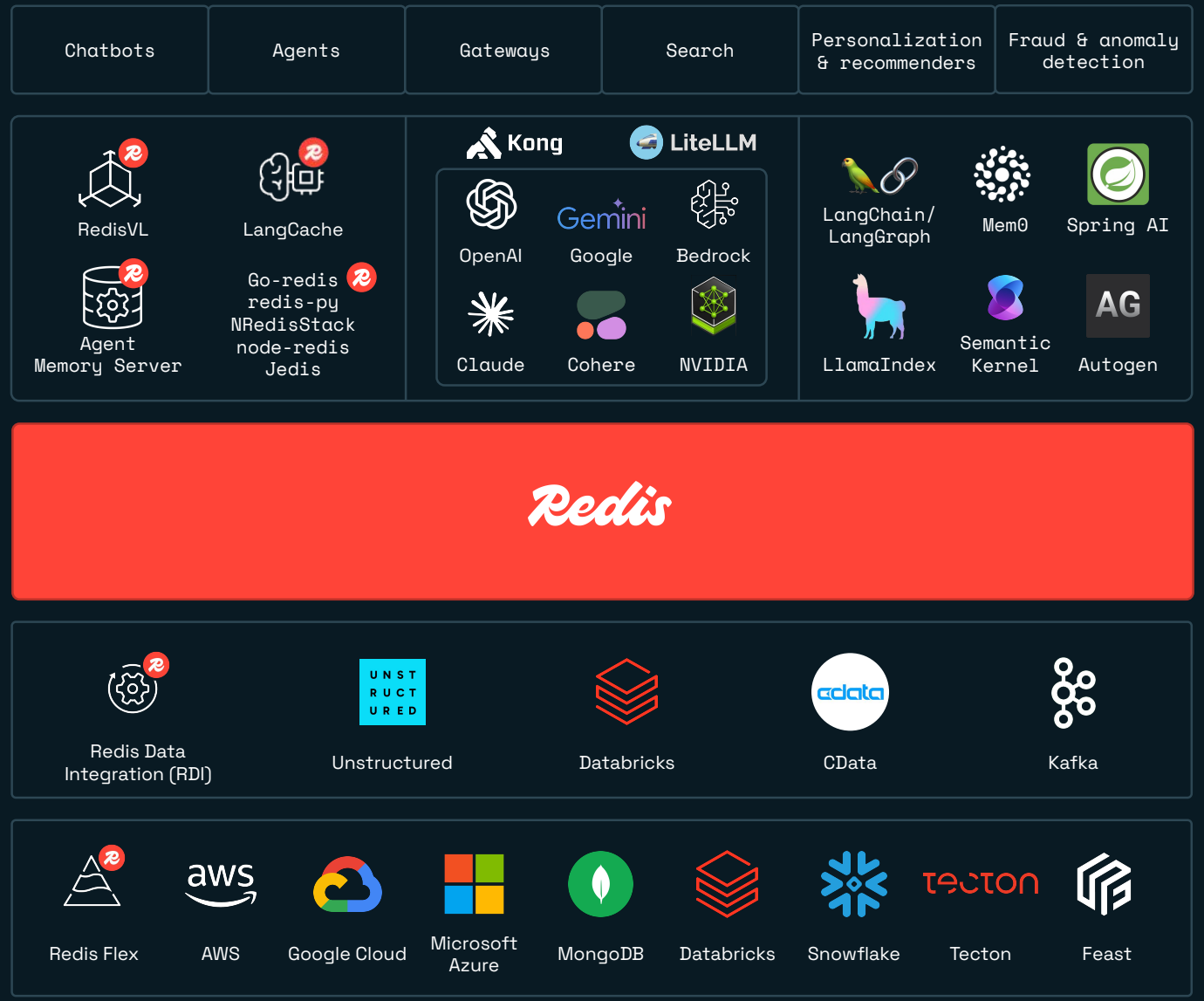
Agenda

- **Vectors and searching with Redis**
  - Understand how vectors fit into GenAI apps
  - Use the Redis vector library to find relevant data to improve your prompts
  - **Lab 1: Vector embedding and search**
- **Redis and Retrieval Augmented Generation (RAG)**
  - Describe RAG and how it works
  - Develop a RAG chatbot using RedisVL
  - **Lab 2: Build a RAG chatbot**
- **Production ready**
  - Use Semantic Caching to reduce time and cost
  - Use **relevant** Long-term Memory to provide better context (and reduce cost)
  - **Lab 3: Production-ready RAG**
- **Applications with agents**
- **General Q&A**

Vector Search

Redis real-time data platform

Redis for AI



The real-time data platform

What you can build

- Chatbots
- AI agents
- Responsive user sessions
- App modernization
- App performance optimization
- Personalization & recommendations

How you build it

CACHING
- Cache
- Semantic cache

AI
- Vector database
- LLM memory
- Feature store

DATABASE
- NoSQL database
- In-memory database

ANALYTICS
- Event stream
- Time series

DATA SYNC
- Redis Data Integration (RDI)
- RIOT

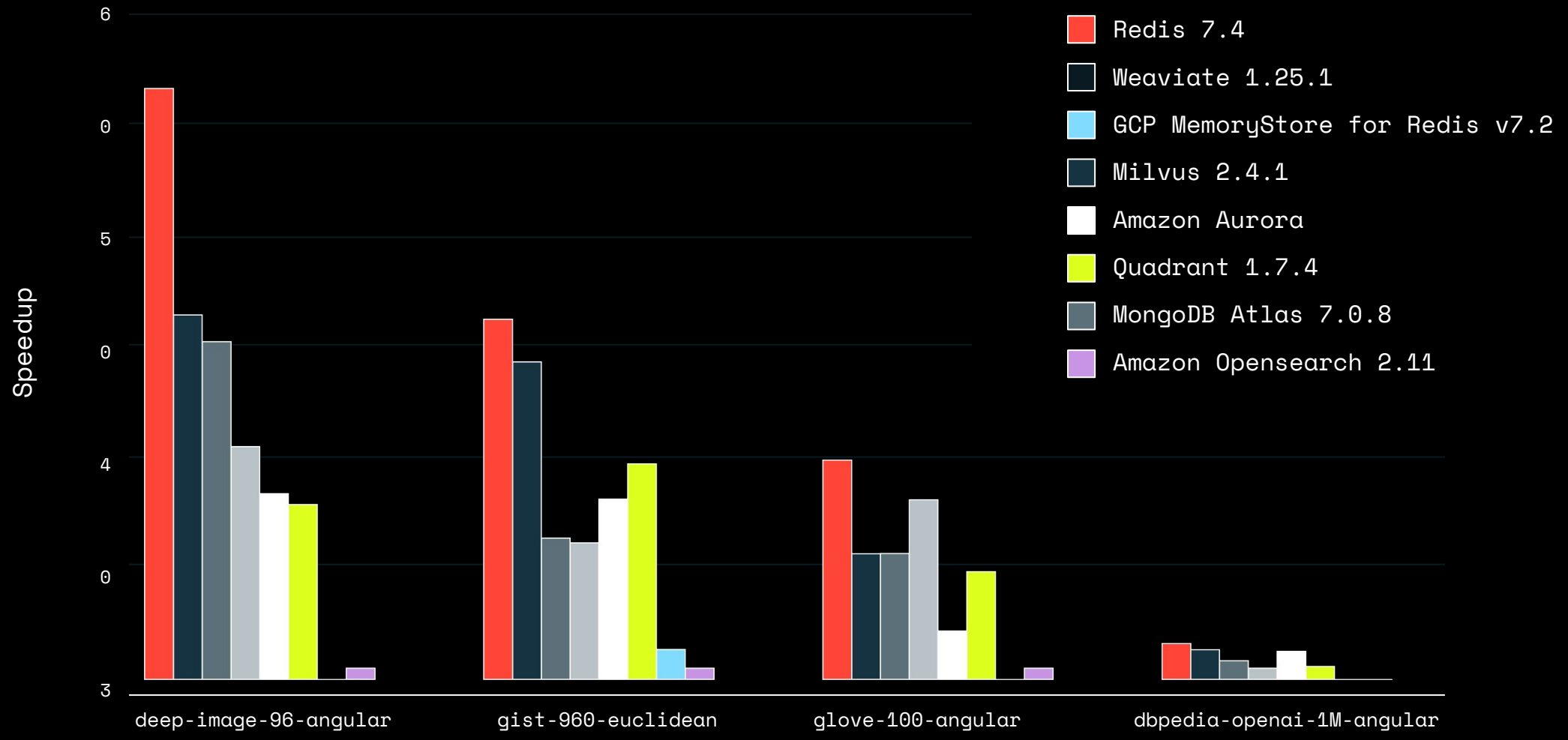
DEVEX
- Redis Insight
- Copilot
- Official clients
- Redis Command API
- RESP protocol

INTEGRATIONS
- LlamaIndex
- LangChain
- LangGraph

What it's built on

GLOBAL DEPLOYMENT LAYER Flexibility & security	Hybrid & multi-cloud . Infrastructure as code . Security
HYPERSCALE LAYER Fast at scale	99.999% availability with multi-region Active-Active Multi-tenancy . Redis Flex
REAL-TIME CACHE ENGINE Redis Open Source	Redis Query Engine . Multi-model . Persistence

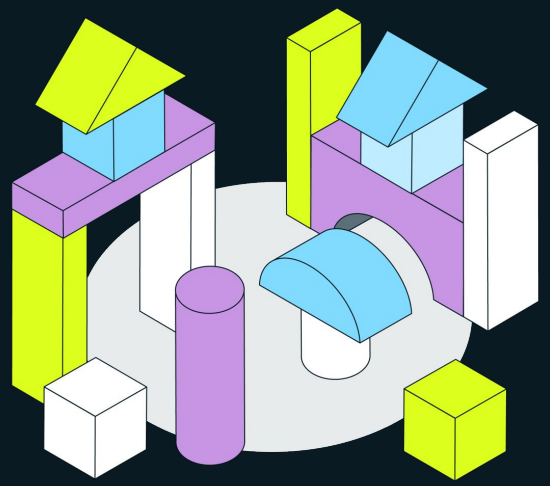
Faster than every other vector database. Period.

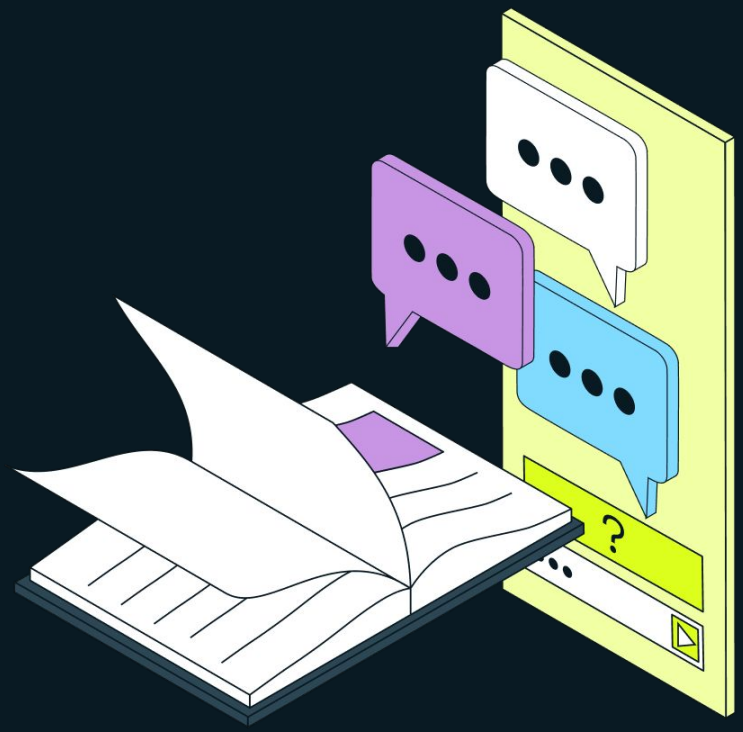


https://redis.io/blog/benchmarking-results-for-vector-databases/

Where we're going today

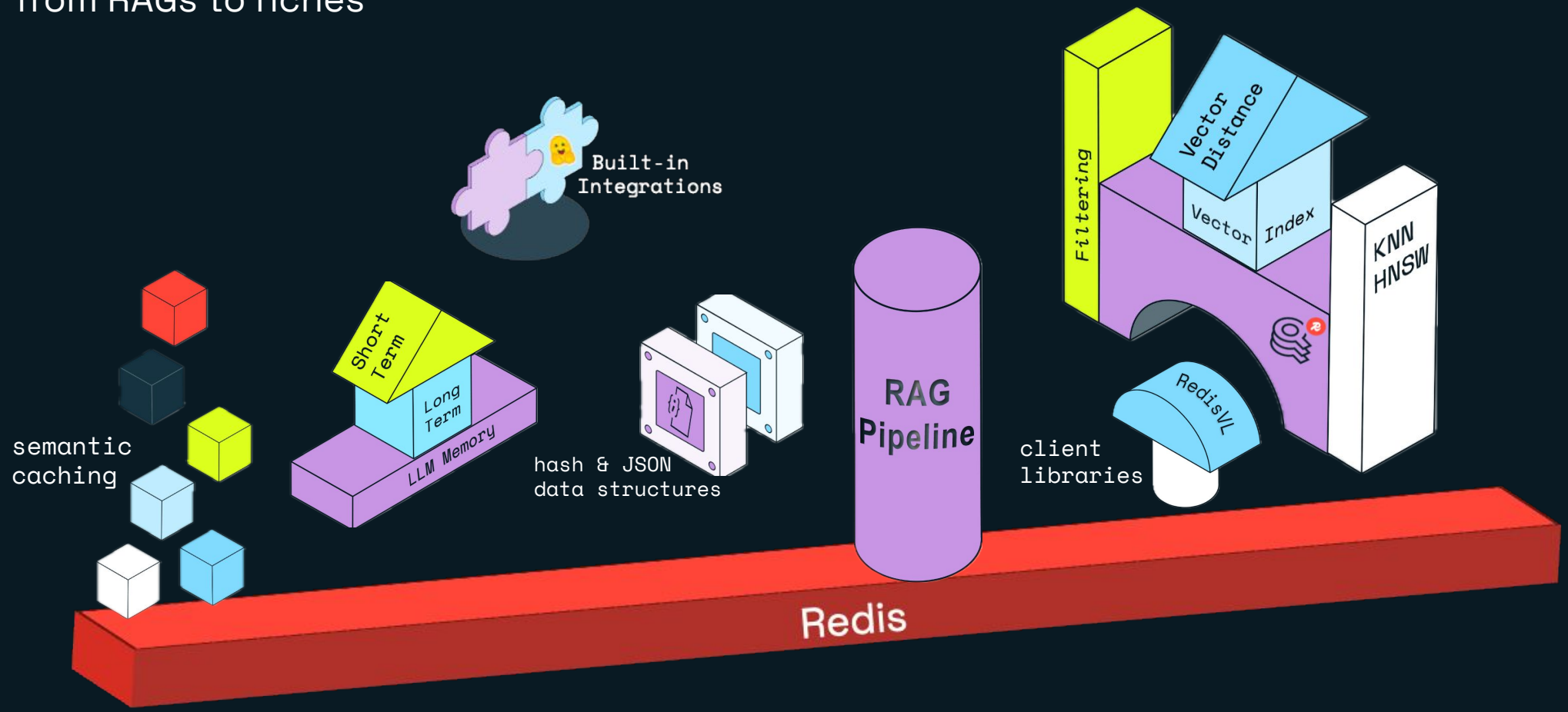
From building blocks to product ready RAG



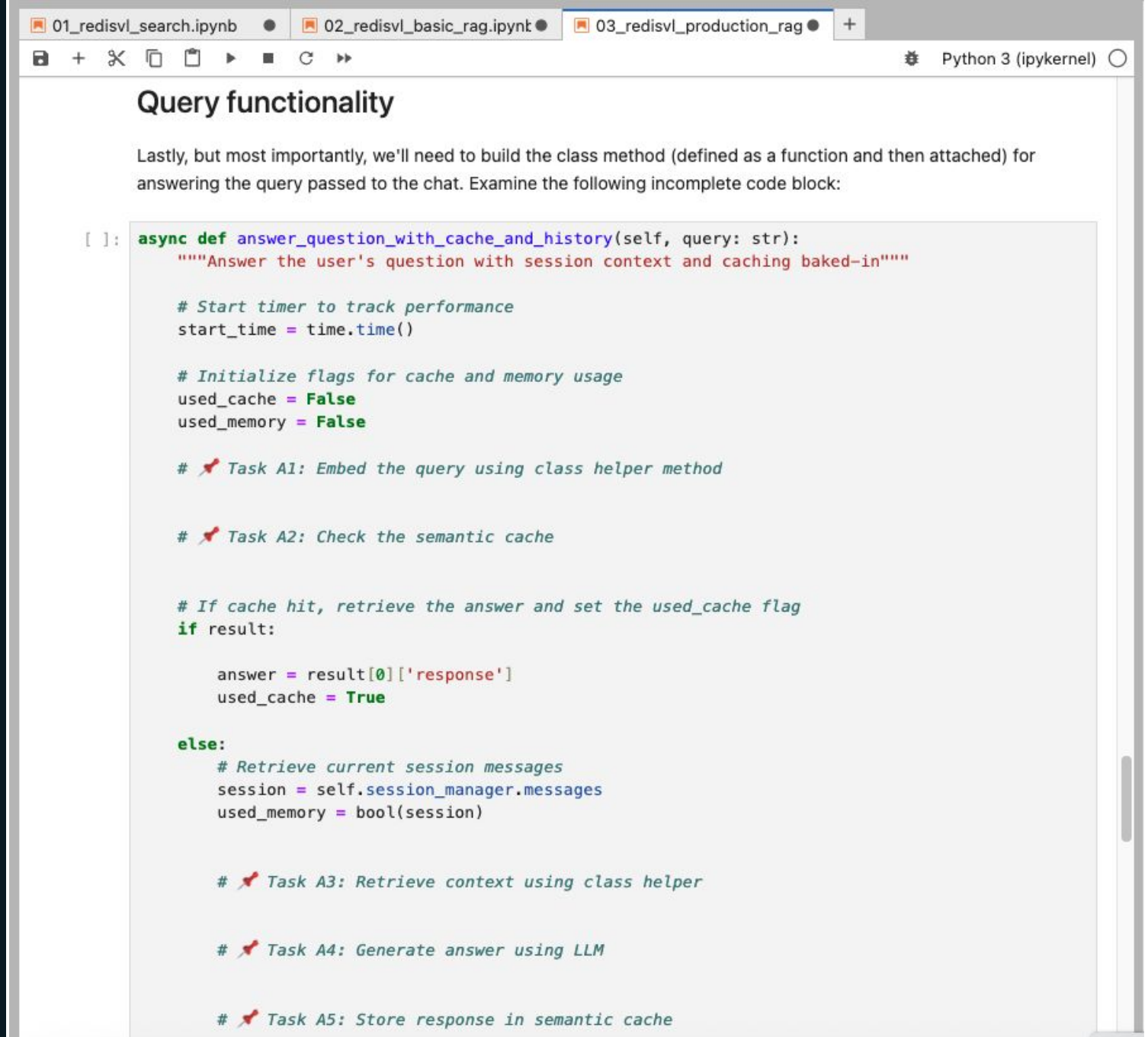


A production-ready RAG pipeline

Or – “from RAGs to riches”



Lab environments provided



Vector data structures and search

Vector as a data type

Stored as a field in a **Hash** or **JSON**

Hash

- Hash structure only supports the string and byte-array types.
- Vector data is stored as a byte-array, for example:

```
HSET docs:01 doc_embedding
myByteArray category sports
```

Your client code would convert the vector to a byte-array before saving it.

JSON

- JSON supports arrays of numbers and arrays of arrays.
- Vector data is stored as a numeric array for example:

```
JSON.SET docs:01 $
'{"doc_embedding": myFloatArray,
"category": "sports"}'
```

Hash and JSON fields can be indexed for vector search

Hash example:

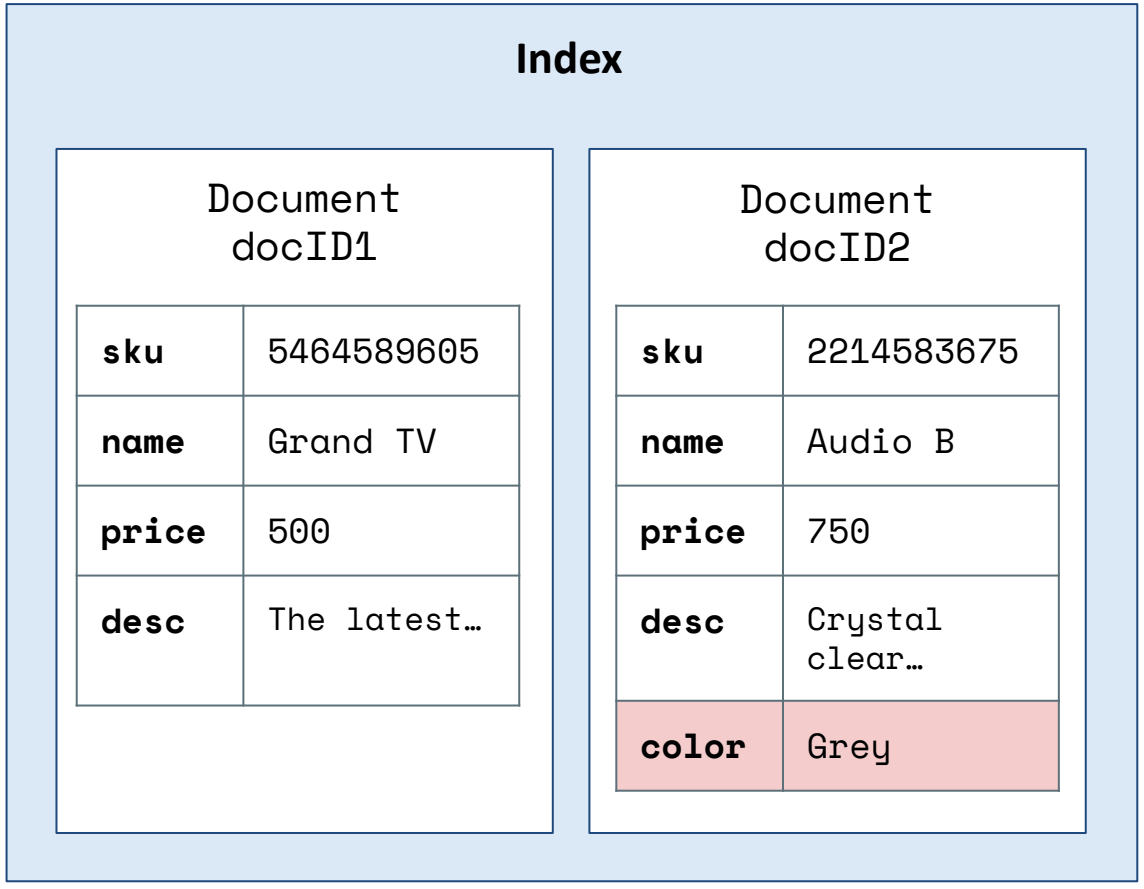
```
"description_embedding" VECTOR HNSW 6
  TYPE FLOAT32 DIM 384 DISTANCE_METRIC COSINE
```

JSON example:

```
"$.description_embedding" AS "desc_embedding" VECTOR HNSW 6
   TYPE FLOAT32 DIM 384 DISTANCE_METRIC L2
```

Indexing concepts

- A collection of documents makes up an **index**
- Documents are identifiable by a **document id**
- A document consists of multiple fields
- Not all documents need to have the same fields
- Not all fields are relevant in a search; thus, not all fields have to be indexed
- An index structure is defined by the **schema,** which specifies how the fields are indexed



Vectors can be indexed.

Creating a search schema

Creating a search schema starts with modeling the data to be searched.

We'll start by assuming this is the type of data we have.

```
{
  "docID":"8739a20b81",
  "author":"Anna Conda",
  "title":"Investing in Constricting Markets",
  "department":"finserve",
  "pub_date":"20240627",
  "summary":"Lorem ipsum dolor sit amet, consecte tur adipis
cing elit, sed do eiusmod tempor incidi dunt ut labore et dol
ore magna aliqua.",
  "summary_embedding":[0.04231967,-0.03875972,-0.0152787,0.01
0385, 0.01830176, -0.07294598,...]
  "content_embedding":[ 0.00631137,0.005189,-0. 03774299,-0.0
9026785,0.01094836,-0.05783698,0.0521009,... ]
}
```

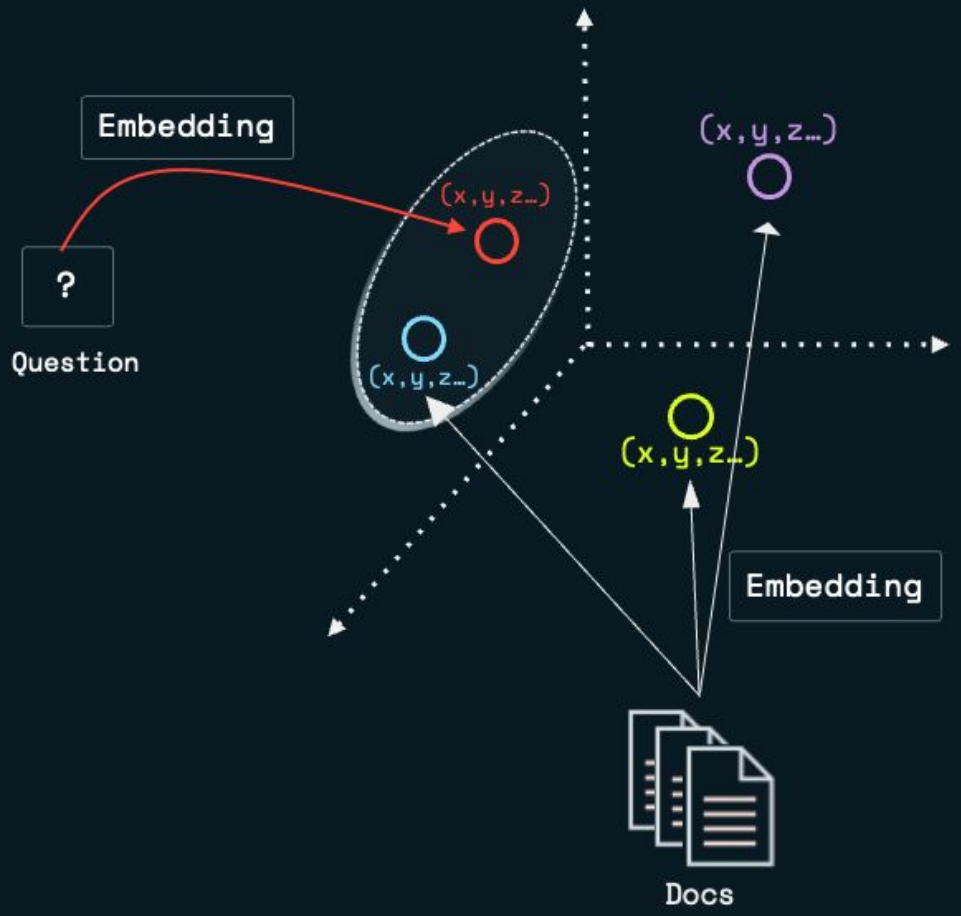
Defining a schema (JSON)

```
FT.CREATE knowledge-base
ON JSON
    PREFIX 1 kb:
SCHEMA
    "$.author" AS "author" TEXT NOSTEM WEIGHT 1.0
    "$.department" AS "department" TAG
    "$.pub_date" AS "pub_date" NUMERIC SORTABLE
    "$.title" AS "title" TEXT WEIGHT 1.0 PHONETIC dm:en
    "$.summary" AS "summary" TEXT WEIGHT 0.8
    "$.summary_embedding" AS "summary_embedding" VECTOR HNSW 6
        TYPE FLOAT32 DIM 384 DISTANCE_METRIC L2
    "$.content_embedding" AS "content_embedding" VECTOR HNSW 6
        TYPE FLOAT16 DIM 384 DISTANCE_METRIC COSINE
```

VECTORS ARE NOT NEW

Overview of Vectors

What is vector search?



- Vectors **are** lists of numbers:
  [-0.987, 0.345, 0.917, 0.370, 0.221, -0.478, …]
- Vectors **represent** data and encode meaning/semantics.
- Vectors **power** content discovery, search, recommendations, anomaly detection and more.

What's vector embedding?

The vector is stored as an array of numbers

- 3-Dimensional array:

```
[0.1316, 0.7251,
0.4055]
```

- 10-Dimensional array:

```
[0.4780, 0.1477,
0.4081, 0.9849,
0.1972, 0.4159,
0.3875, 0.1459,
0.4795, 0.2231]
```

- 20-Dimensional array:

```
[0.7535, 0.1160,
0.4554, 0.6150,
0.3500, 0.1385,
0.4322, 0.7321,
0.9004, 0.4046,
0.6093, 0.4340,
0.3621, 0.7382,
0.6386, 0.6286,
0.3858, 0.3591,
0.2211, 0.3060]
```

In a hash, the vector is stored as a byte-array.

Understand vector search

A visual representation

Embed

A model starts by converting source documents or objects into vectors encoded with semantic meaning.

Query

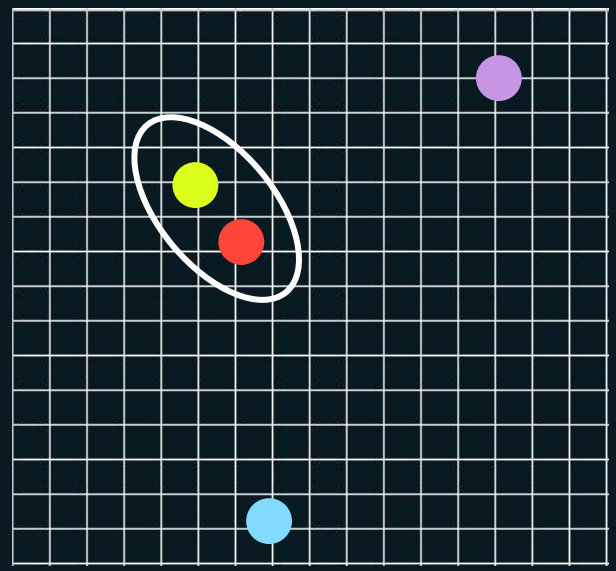
User input is also encoded, and the app retrieves information related to the input.

The cat sat on the mat
The dog sat on the frog
The fox sat in the street

Where is the cat?

The cat is on the mat

Vector space



Why is this a big deal?

Vectors are the **ultimate** machine-readable language.

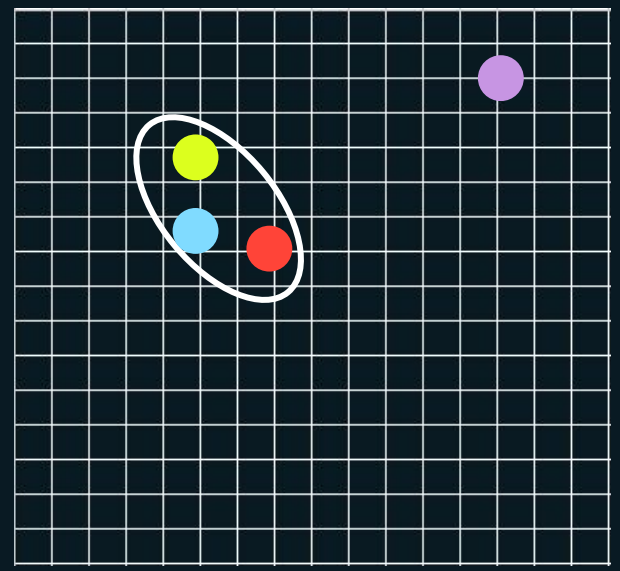
They provide a means of relating **any data** that can be embedded.

Redis offers the **fastest and most flexible database** for enabling the next level of insight.

<image> of a cat
<image> of truck
<image> of a fox
<image> of a dog

Which image is an outlier?

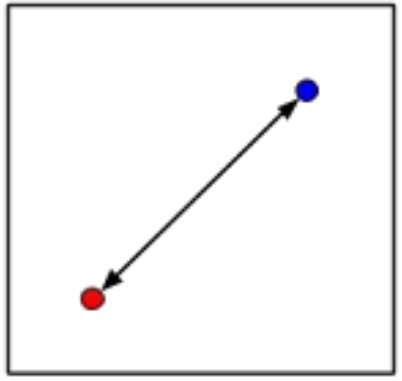
Vector space



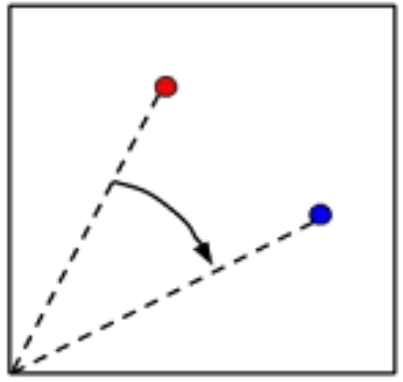
Introduction to vector embeddings

Distance metrics

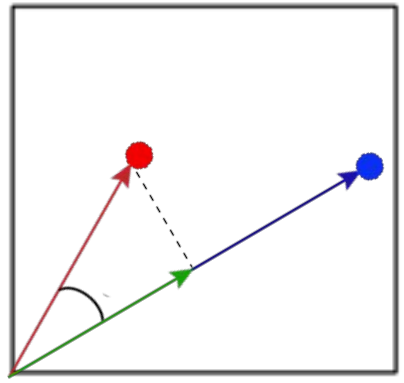
VS focuses on finding out how alike or different two vectors are. To achieve this reliably and measurably, we need a specific type of score that can be calculated and compared objectively. These scores are known as **distance metrics**.



Euclidean



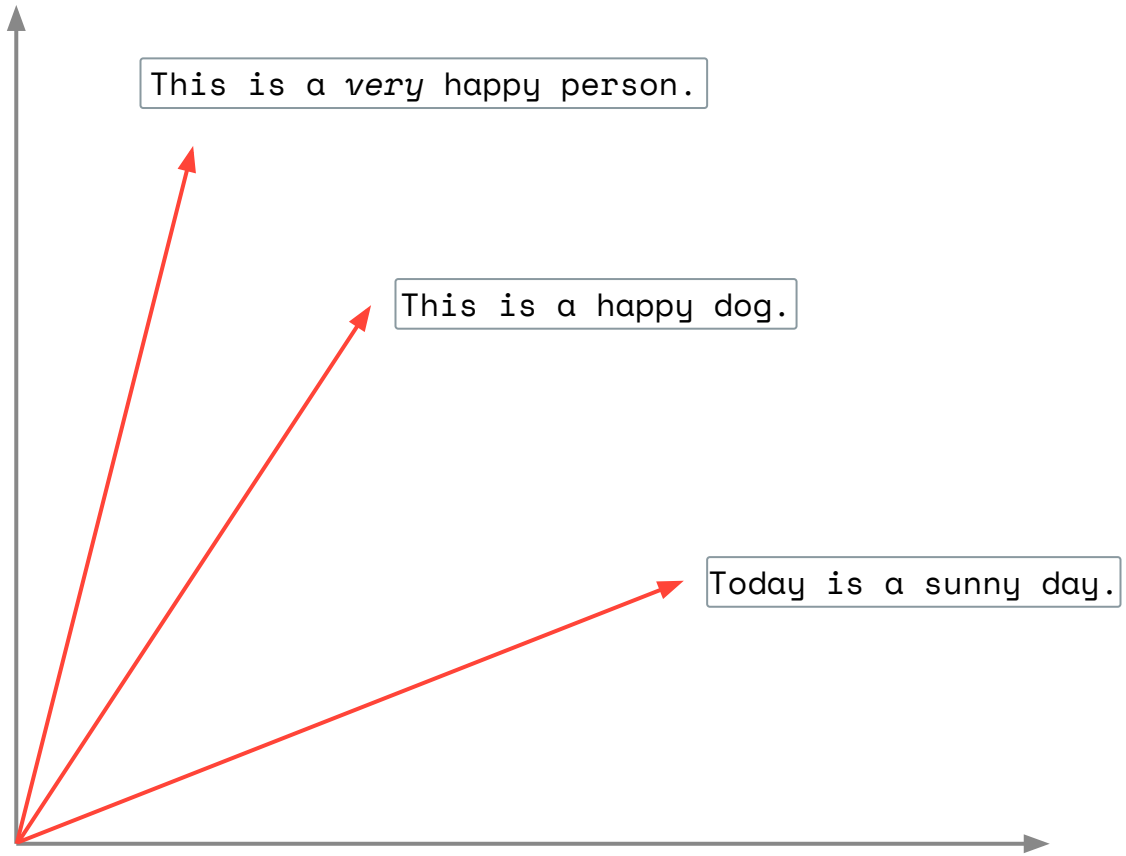
Cosine Similarity



Inner Product

Vector distance - cosine

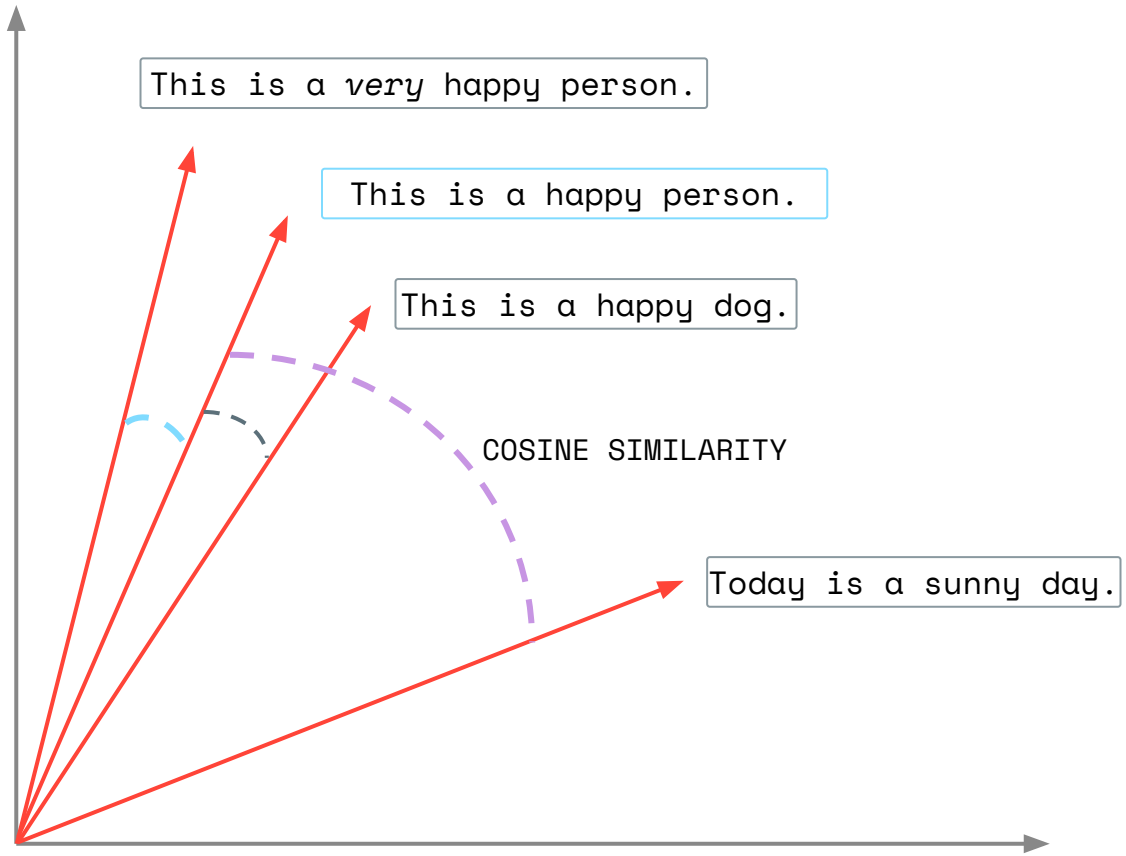
Search with cosine similarity distance



- Embeddings translate data semantic features to the vector space
- Unstructured data can be vectorized
- Particularly useful for data such as long texts, images, audio and video files
- Vectors belong to a multi-dimensional space

Vector distance - cosine

Search with cosine similarity distance



- The **degree of similarity** is expressed by the **distance between vectors**.
- **Cosine similarity** measures the cosine of the angle between vectors, indicating their ‘similarity.’
- Cosine similarity does **not** consider the vectors’ modulus (i.e., magnitude).

Vector search provides these functionalities

Realtime vector update/delete, triggering an update of the index.

Realtime vector indexing supporting **two indexing methods**:

- **FLAT** - ”Brute-force” index
- **HNSW** - an implementation of approximate nearest neighbor (ANN) search using Hierarchical Navigable Small World graphs.

K-nearest neighbors (KNN) search *and* **range** search, supporting three distance metrics:

- **L2** - Euclidean distance between two vectors
- **IP** - Inner product of two vectors
- **COSINE** - Cosine distance of two vectors

The two indexing methods

FLAT

Queries are **O(n * d)**

Query Process:

1. For a query vector, compute the distance to *every* vector in the dataset
2. Sort the distances to find the top-k nearest neighbors

HNSW

Queries are **O(log n * d)**

Query Process:

1. Start at a random node in the top layer
2. Navigate through the graph by moving to closer neighbors across layers
3. In lower layers, refine the search to find the approximate nearest neighbors
4. Return the top-k closest vectors.

n = number of documents d = number of dimensions

Bottom line

FLAT

Queries are **O(n * d)**

Best when:

1. Exact results are required.
2. You can tolerate a bit more latency.
3. Number of documents is relatively small.
4. You have the CPU or GPU for it.

HNSW

Queries are **O(log n * d)**

Best when:

1. You can settle for ≳95% certainty
2. You need speed.
3. Number of documents is huge.
4. You have the RAM for it.

Redis Vector Database Capabilities

Storage & indexing	Hash	JSON	Flat (KNN)	HSNW (ANN)
Distance metrics	Euclidean	Cosine	Inner product	
Search techniques	KNN/ANN	Filtered queries	Range queries	Full text search
Client libraries	Redis Vector Library	redisOM	.NET Core	

The native Redis Vector Library

RedisVL

RedisVL

Redis Vector Library

A powerful, dedicated Python client library for using Redis as a vector database.

- **Free to use**, just pip install redisvl
- **Easy to deploy** GenAI solutions with simple abstractions
- **Built on Redis** so you benefit from all our speed and ecosystem integrations

Vector database

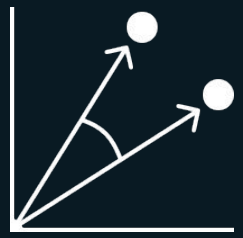
Semantic cache

LLM memory

Semantic routing

RedisVL client library

Key features



Vector Search: Efficiently find nearest neighbors in high-dimensional spaces using algorithms like HNSW

Integration with AI Frameworks: Works seamlessly with popular frameworks such as TensorFlow, PyTorch, and Hugging Face, making it easy to deploy AI models

Link to RedisVL docs

RedisVL client library

Key features

Scalable and Fast: Leveraging Redis' in-memory architecture, provides low-latency access to vector data, even at scale

Bridging the gap between data storage and AI model deployment, RedisVL empowers developers to build intelligent, real-time applications with **minimal infrastructure complexity**

Link to RedisVL docs

RedisVL examples

RedisVL simplifies the process of storing, retrieving, and performing semantic searches on vectors in Redis.

Index example

```
index:
  name: user_index
  storage_type: hash
  prefix: users

fields:
  # define tag fields
  tag:
  - name: user
  - name: job
  - name: credit_store
  # define numeric fields
  numeric:
  - name: age
  # define vector fields
  vector:
  - name: user_embedding
    algorithm: hnsw
    distance_metric: cosine
```

Query example

```
query = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "age", "job", "credit_score"],
    num_results=3,
)
results = index.query(query)
```

Vector search syntax

Creating a schema for vectors

```
FT.CREATE knowledge-base
ON JSON
    PREFIX 1 kb:
SCHEMA
   Etc. …
    "$.content_embeddings" AS "content_embeddings" VECTOR HNSW 12
      TYPE FLOAT16
      DIM 384
      DISTANCE_METRIC COSINE
      M 40
      EF_CONSTRUCTION 250
      EF_RUNTIME 16
```

Required input

Optional input

Schema creation in RedisVL

There are two options for creating a schema

YAML Definition

1. Define schema in a .yaml file
2. Create the schema with this syntax

```
from redisvl.index import SearchIndex

index =
SearchIndex.from_yaml('schema_file.yaml')

# assumes this yaml has a valid schema
```

Python Dictionary

1. Define schema in Python code
2. Create the schema with this syntax

```
from redisvl.index import SearchIndex

index = SearchIndex.from_dict(myschema)

# assumes you have previously defined a
# dictionary named myschema
```

Schema creation in RedisVL

Sample schema **yaml**

- Give the index a name
- Specify the key prefix and data type (`hash | json`)
- Define the fields and their attributes. This example shows only two fields:
  - `user` – a tag
  - `user_embedding` – a vector

```yaml
index:
  name: user_simple
  prefix: user_simple_docs
  storage_type: json

fields:
    - name: user
      type: tag
    - name: user_embedding
      type: vector
      attrs:
        algorithm: flat
        dims: 384
        distance_metric: cosine
        datatype: float32
```

Schema creation in RedisVL

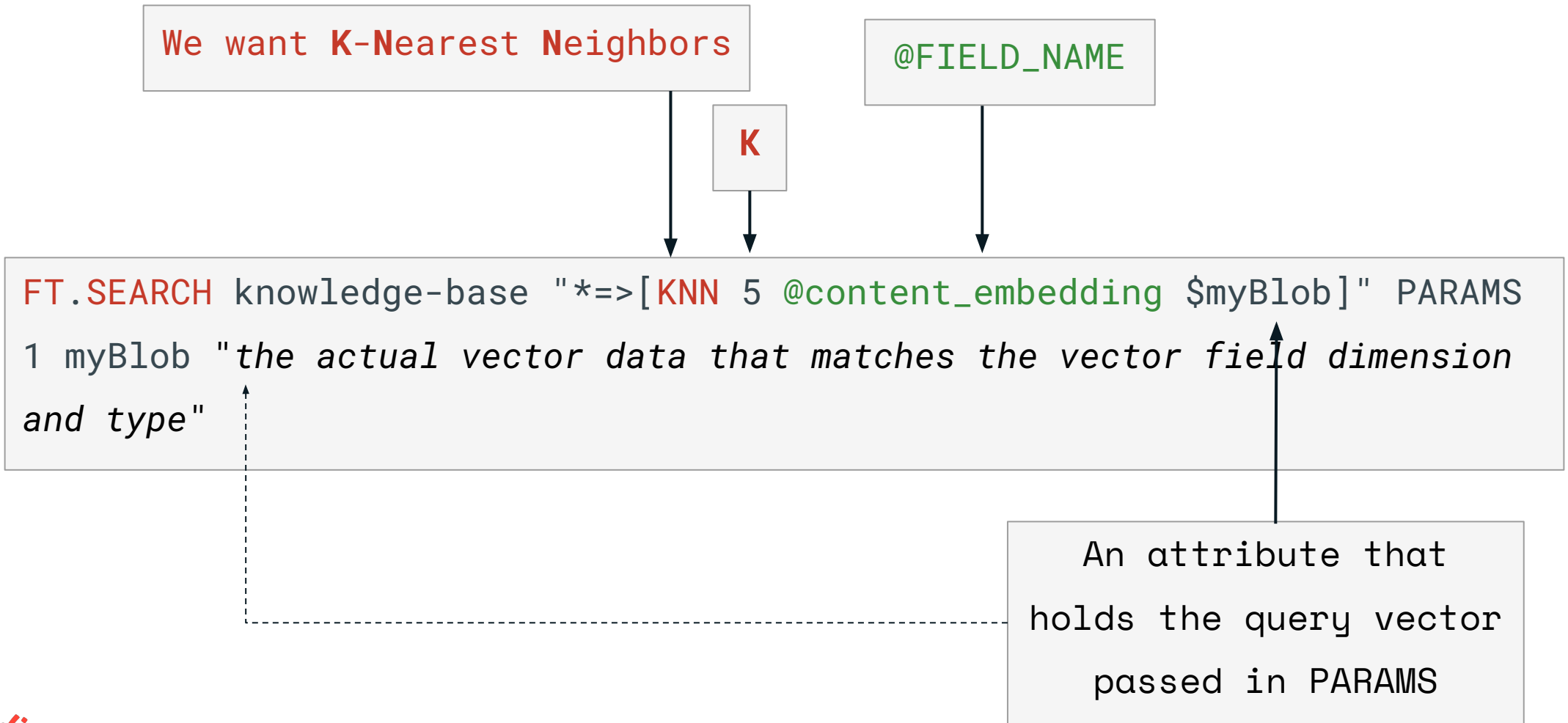
Sample schema **dictionary**

- Give the index a name
- Specify the key prefix and data type (`hash|json`)
- Define the fields and their attributes. This example shows only two fields:
  - `user` – a tag
  - `embedding` – a vector

```
myschema = IndexSchema.from_dict({
    "index": {
        "name": "user_simple",
        "prefix": "user_simple_docs",
        "storage_type": "json",
    },
    "fields": [
        {"name": "user", "type": "tag"},
        {
            "name": "embedding",
            "type": "vector",
            "attrs": {
                "algorithm": "flat",
                "dims": 384,
                "distance_metric": "cosine",
                "datatype": "float32"
            }
        }
    ]
})
```

Vector search syntax

Vector search queries: KNN



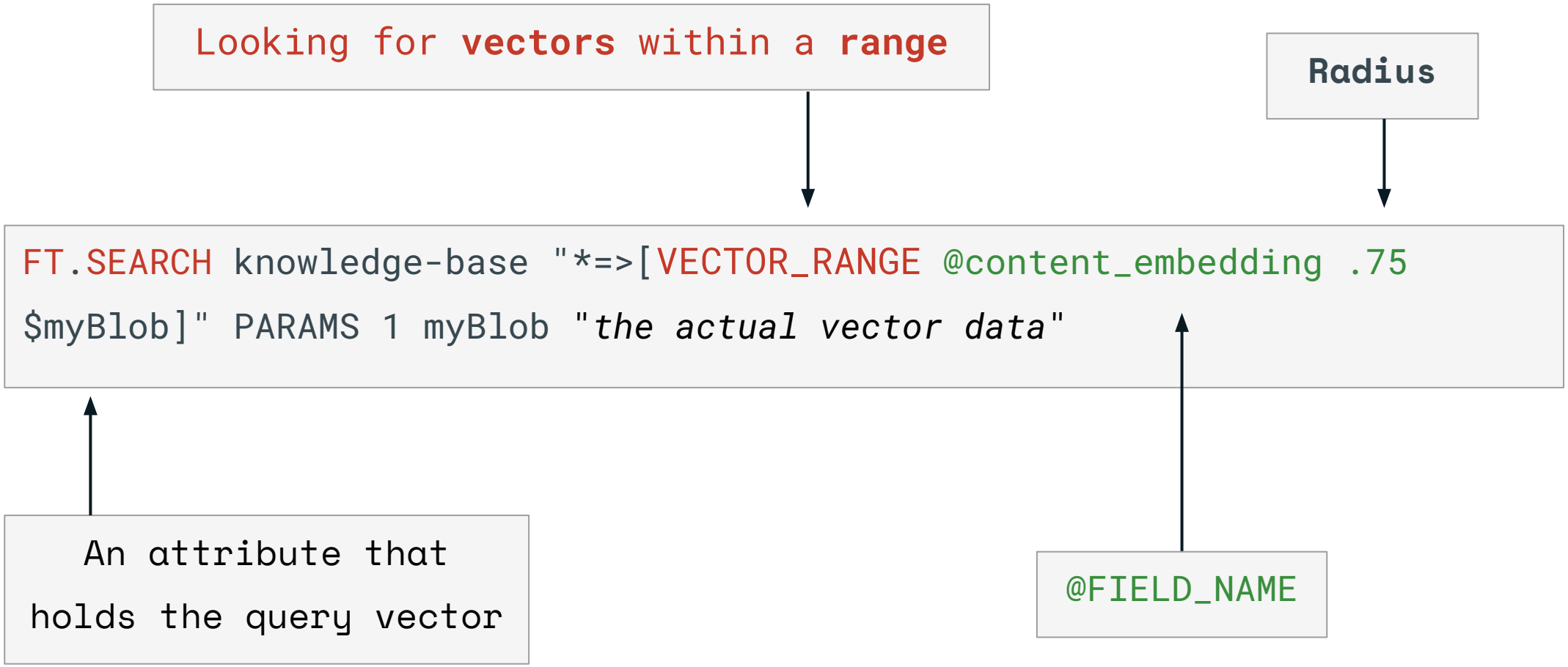
KNN search syntax in RedisVL

- Define the VectorQuery and provide these attributes:
  - `vector` – the search vector
  - `vector_field_name`
  - `num_results` – the value for K (defaults to 10)
  - `return_fields` – defaults to `"none"` if you only want the doc IDs
- Run the query
- Additional, optional parameters are described in the RedisVL API doc

```
vq = VectorQuery(
    vector=my_search_vector,

vector_field_name="user_embedding",
    num_results=K
    return_fields=["user",
      "some_attribute",
      "another_attribute"]
)

results = index.query(vq)
```

Example 1: Creating a KNN query

Vector search queries: Range



Range search syntax in RedisVL

Like KNN but using VectorRangeQuery

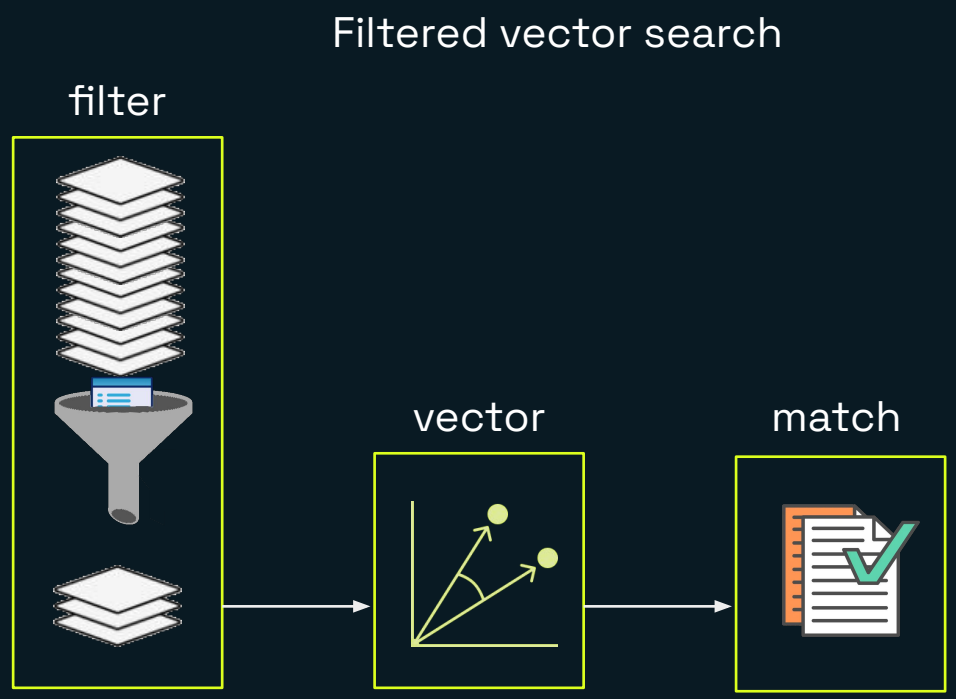
- Define the `VectorRangeQuery` and provide these attributes:
  - `vector` – the search vector
  - `vector_field_name`
  - `distance_threshold` – threshold for vector distance. (smaller threshold indicates a stricter semantic search; defaults to 0.2)
  - `return_fields`
- Run the query
- Additional, optional parameters are described in the [RedisVL API doc]

```
vq = VectorRangeQuery(
   vector=my_search_vector,

vector_field_name="user_embedding",
   distance_threshold=myDistance
   return_fields=["user",
     "some_attribute",
     "another_attribute"]
)

results = index.query(vq)
```

Example 1: Creating a range query

Filter the search

You can limit the vector search with numeric, text, or tag filters.



Redis allows you to combine filter searches on fields within the index object allowing you to create more specific searches.
For example:

- Recent articles (publication date > spec)
- Movie ratings (rating > 7.0)
- Tagged articles (tag == “GenAI” or “vector”)
- “Liked” products

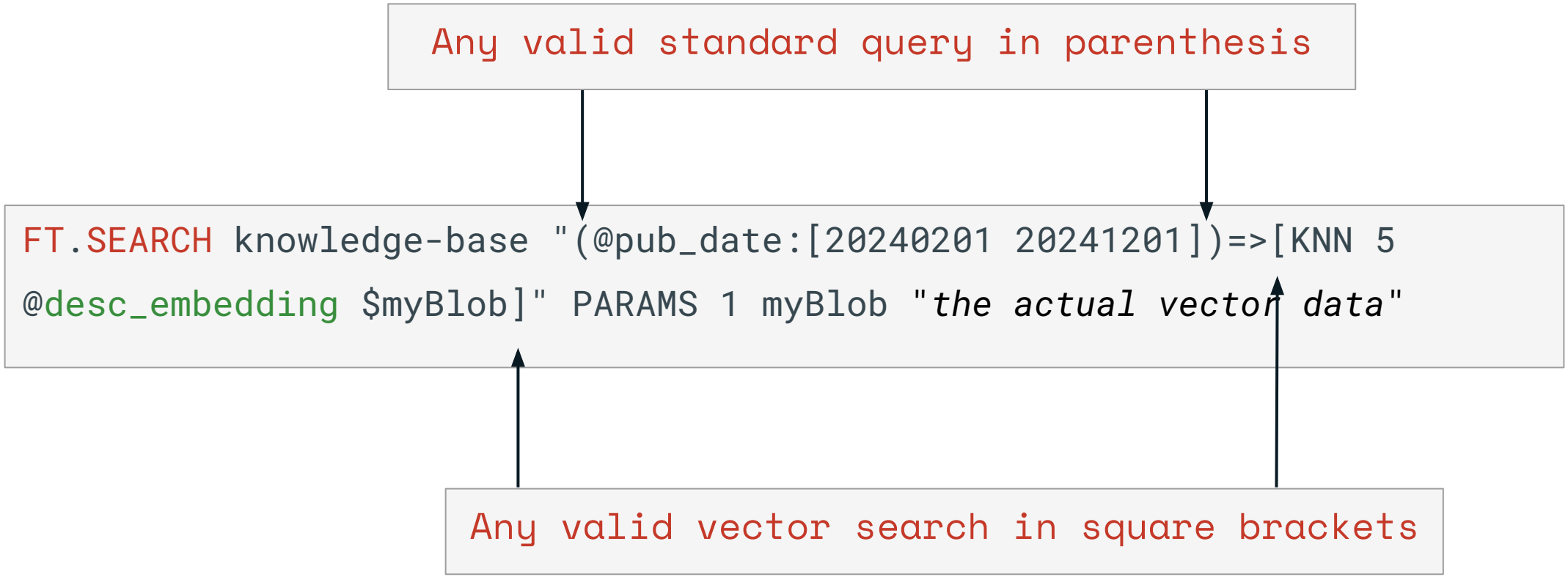
Filter query example

Command line syntax

```
FT.SEARCH knowledge-base "@pub_date:[20240101 20241231]"=>[vector search]
```

- The “filter” is run first
- The vector search is performed against the result set
- This is typically a faster search because it minimizes the vector burden
  - By indexing the metadata, like title and publication date, you can pre-filter searches using more efficient search features.
  - Additionally, hybrid search is especially effective for RAG and Custom Ranking use cases.

Filtered vector search queries



Filtered search in RedisVL

- Write the `VectorQuery` the same as before but **add `filter_expression`**
- The filter, and any other options, need *not* be set during instantiation. The class has functions to set them.

For example:

```
myFilter= Tag(color) == “red”

vq.set_filter(myFilter)
```

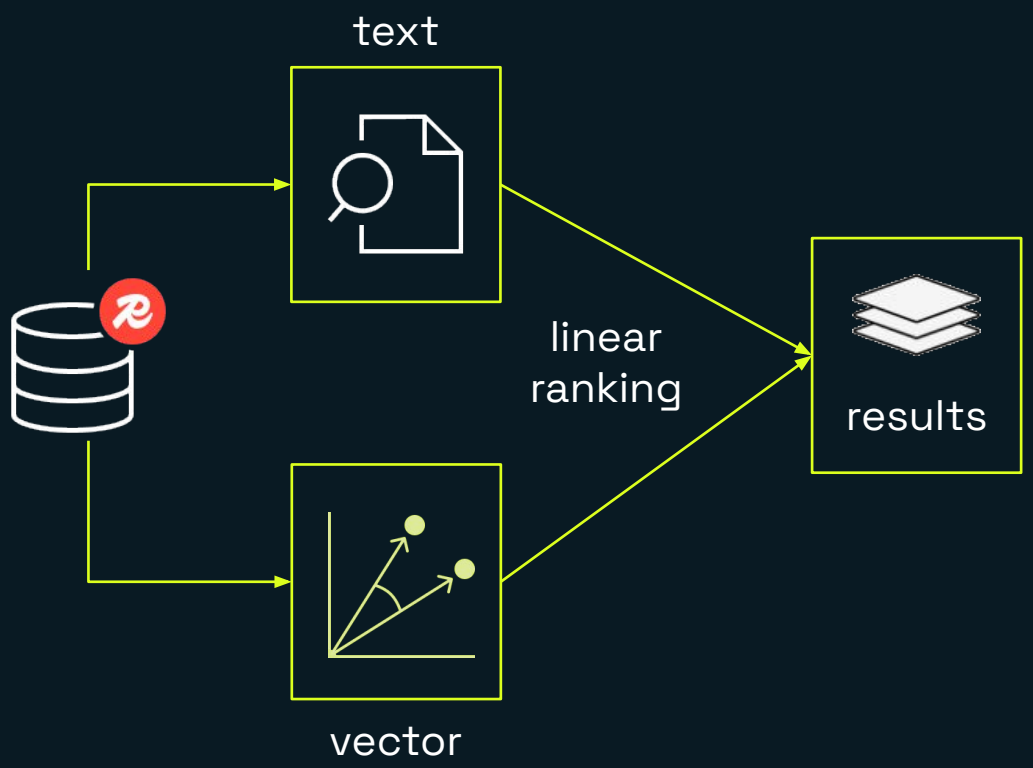
Example 1: Adding a filter to a previously defined query

```
myFilter= Tag(color) == “red”

vq = VectorQuery(
  vector=my_search_vector,
  vector_field_name="desc_embedding",
  num_results=5,
  return_fields=[”name”,”sku","price"],
  filter_expression=myFilter
)

results = index.query(vq)
```

Example 2: Creating a query with a filter in it

Hybrid search

Combine the results of two techniques



Use RedisVL to combine the results of a full text search and a vector search. Order the results by a weighted linear score.

You specify the alpha factor ($0 < \alpha < 1$).

```
doc_score = (vector_doc_score * α)
+ (text_doc_score * (1-α))
```

Hybrid Search: Linear combination

Using the HybridQuery class in RedisVL

```
class HybridQuery(text, text_field_name, vector, vector_field_name,
text_scorer, filter_expression, alpha, dtype, num_results,
return_fields, stopwords, dialect)
```

- HybridQuery combines text and vector search.
- It scores documents based on a weighted combination of text and vector similarity.
- The alpha value is the linear combination factor.

Note: Redis CLI does not have separate syntax for linear combination. You would need to write the code to do each search independently and then combine the results.

Sample code

```
index = SearchIndex.from_yaml("path/to/index.yaml")

query = HybridQuery( text = my_text_search_string,
   text_field_name = "indexed_text_field",
   vector = my_search_vector,
   vector_field_name = "indexed_vector_field",
   text_scorer = "BM25",                    # optional
   alpha = 0.7,
   dtype = "float32",                       # optional
   num_results = 10,                        # optional
   return_fields = ["field1", "field2"],    # optional
   )
results = index.query(query)
```

Meaning of less obvious fields

- **text_scorer** – The text scorer to use. Options are {TFIDF, TFIDF.DOCNORM, BM25, DISMAX, DOCSCORE, BM25STD}. Defaults to “BM25STD”
- **alpha** – The weight of the vector similarity. Documents will be scored as: `hybrid_score = (alpha)*vector_score + (1-alpha)*text_score`.

  Defaults to 0.7

Hybrid: Custom reranker

Reranking provides a relevance boost to search results

The goal of reranking is to provide a more fine-grained quality improvement to initial search results. It is not exclusive to hybrid search scenarios.

The steps

1. Perform independent text and vector searches.*
2. Rank the results of each with a custom ranking algorithm.
3. Join the results using the calculated document ranks.

*Note: some rerankers let you combine vector search score and metadata search score automatically.

Reranking in RedisVL

Out of the box

RedisVL supports reranking through:

- `HFCrossEncoderReranker`
  - A re-ranker that uses pre-trained Cross-Encoders which can use models from Hugging Face cross encoder models (example: `BAAI/bge-reranker-base`).
- The `Cohere/rerank` API.
- The `VoyageAI/rerank` API.
- More to come ...

Lab 1: Vector Search

01_redisvl_search.ipynb

1. Instantiate a text vectorizer and use it to create embeddings for text
2. Create a search index that includes a vector field
3. Perform vector searches

Lab Debrief

- Lab 1

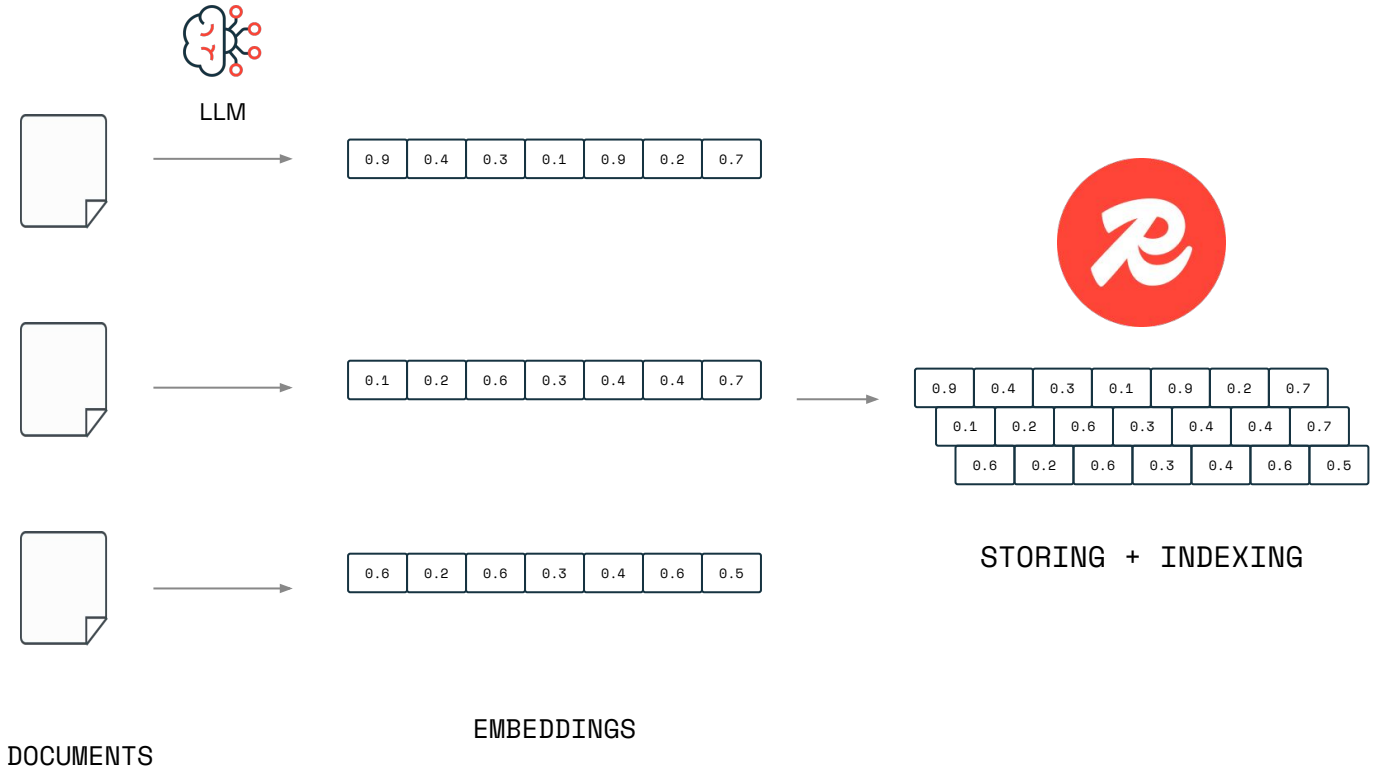
In this lab you:

- Instantiated a text vectorizer and used it to create embeddings for text
- Created a search index that included a vector field
- Performed these vector searches:
  - KNN vector query
  - KNN vector query with filters
  - Range query
  - A simple linear hybrid search

Vector database for LLM-based use cases

Text Semantic Search

Vectorize, store, and index your documents.



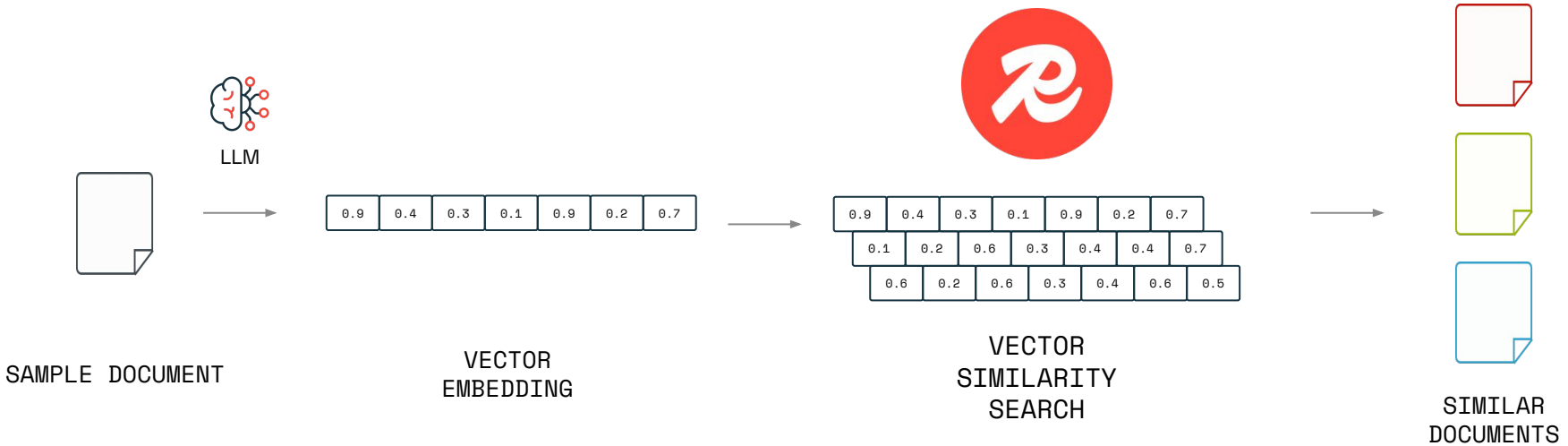
1. Embedding models consider documents **up to a supported number of words**
2. **Split the documents** into chunks if they exceed the supported length
3. **Store** the original documents and its metadata in a hash or JSON along with the embedding

JSON documents can store and index multiple embeddings.

Text semantic search

Search your documents.

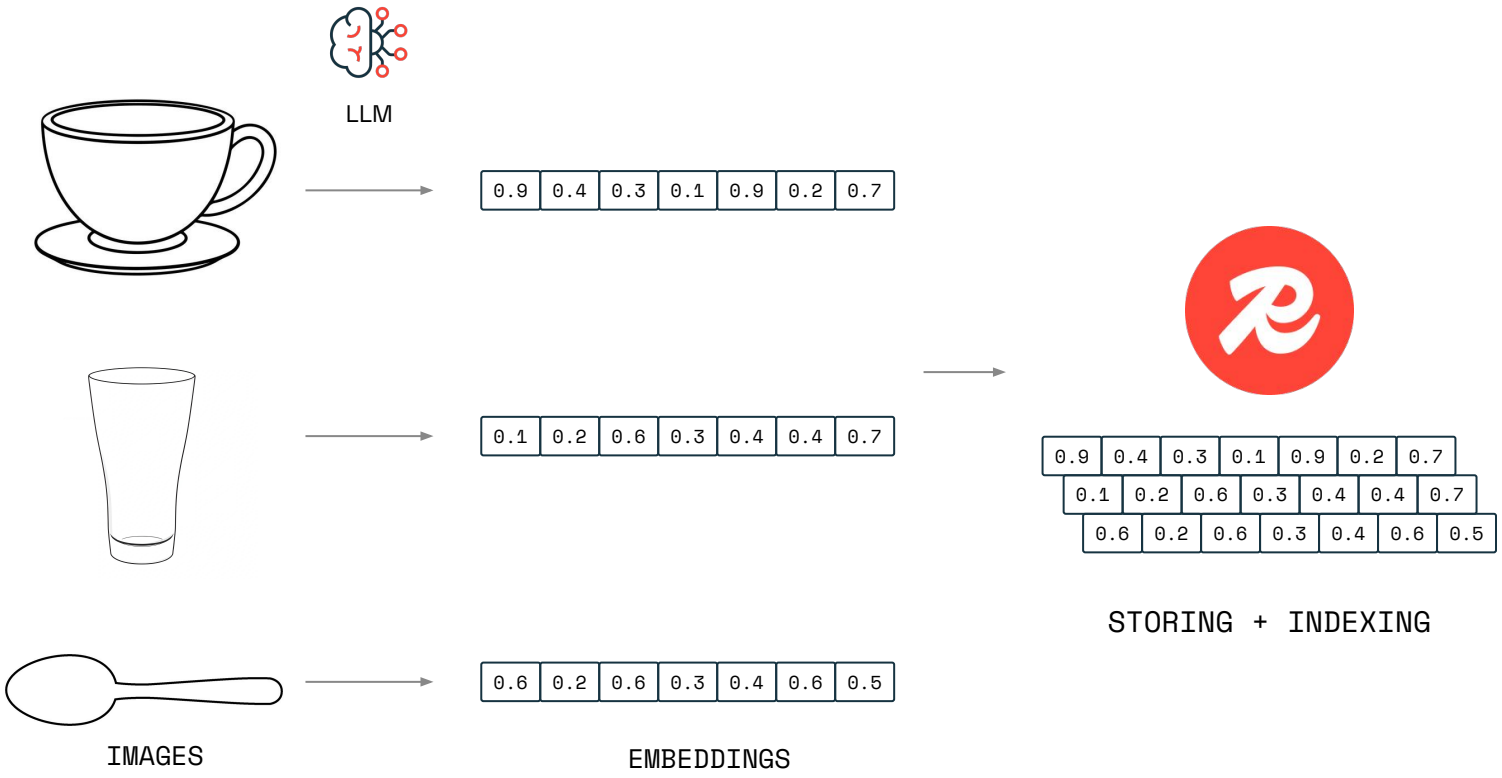
1. The document being read already has a vector embedding associated with it.
2. The embedding is compared to the rest of vector embeddings.
3. You can also filter the results, limit the number returned, and perform hybrid searches.



Based on my current document, I want to get a list of recommendations.

Visual search

Vectorize, store, and index your images.



JSON documents can store and index multiple embeddings.

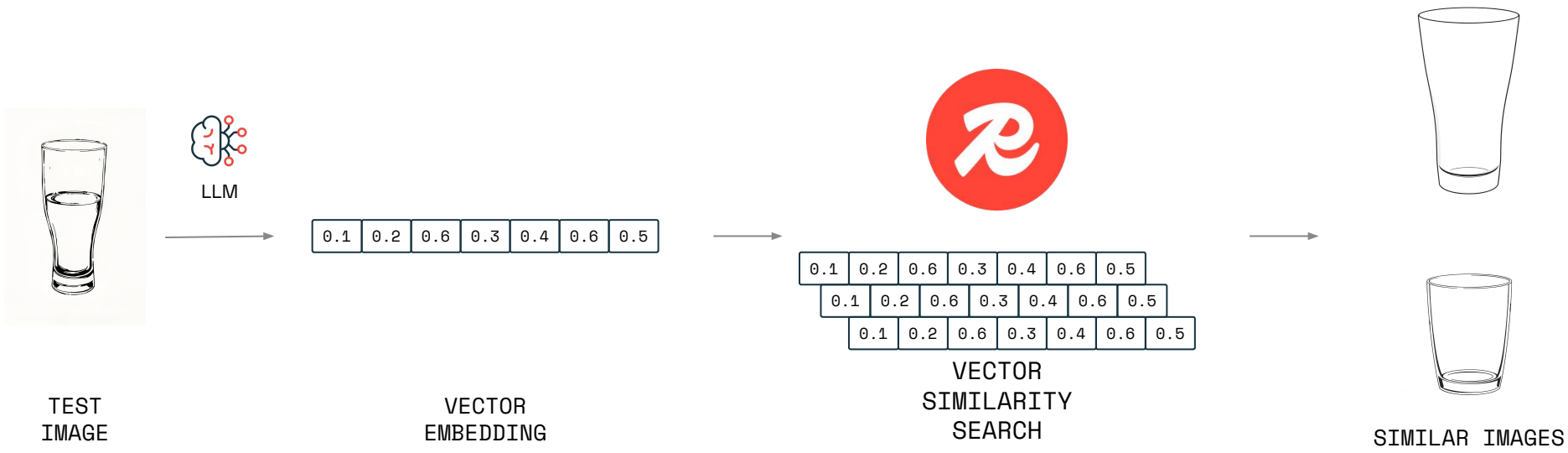
1. **Convert** the images to the vector embedding using a suitable model (Resnet, Densenet, etc.)

2. **Store** the embedding together with metadata. (Images are usually stored in a separate file system)

3. Store the embeddings and metadata in a **hash or JSON**

Visual search

1. Use the image processing service to **calculate the vector embedding** for the query image.
2. The embedding is **compared** to the rest of vector embeddings with VS.
3. You can also **filter** the results, **limit** the number returned, and perform **hybrid searches**.



Propose a list of similar products to the one the shopper has/wants.

Large Language Models (LLM)

What are they and what do they do?

LLM Providers

- LLMs are massive, general purpose neural networks, pre-trained on large amounts of text.
- Specifically focused on language understanding and generation (GPT, BERT, LLAMA).
- Commonly utilize **vector search** to retrieve information from external databases
- Use Cases:
  - Translation
  - Sentiment Analysis
  - Content Generation/Summarization
  - Answering Questions

LLMs have gaps.

Training data is stale

LLMs answer based on training, but can't access up-to-date or domain-specific knowledge.

Costly to scale

Larger models take more time to train and use, increasing cost and limiting availability.

No short-term memory

LLMs don't recall recent user actions. They only know their training data.

Slow responses

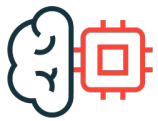
Models take seconds to generate answers, creating bad UX.

Dealing with the gaps

Common design patterns

Retrieval Augmented Generation

- Current documents provide context to LLM

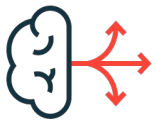


Semantic Caching

- Return cached responses; save time & money.

LLM Memory

- Include relevant history as context for LLM



Semantic Routing

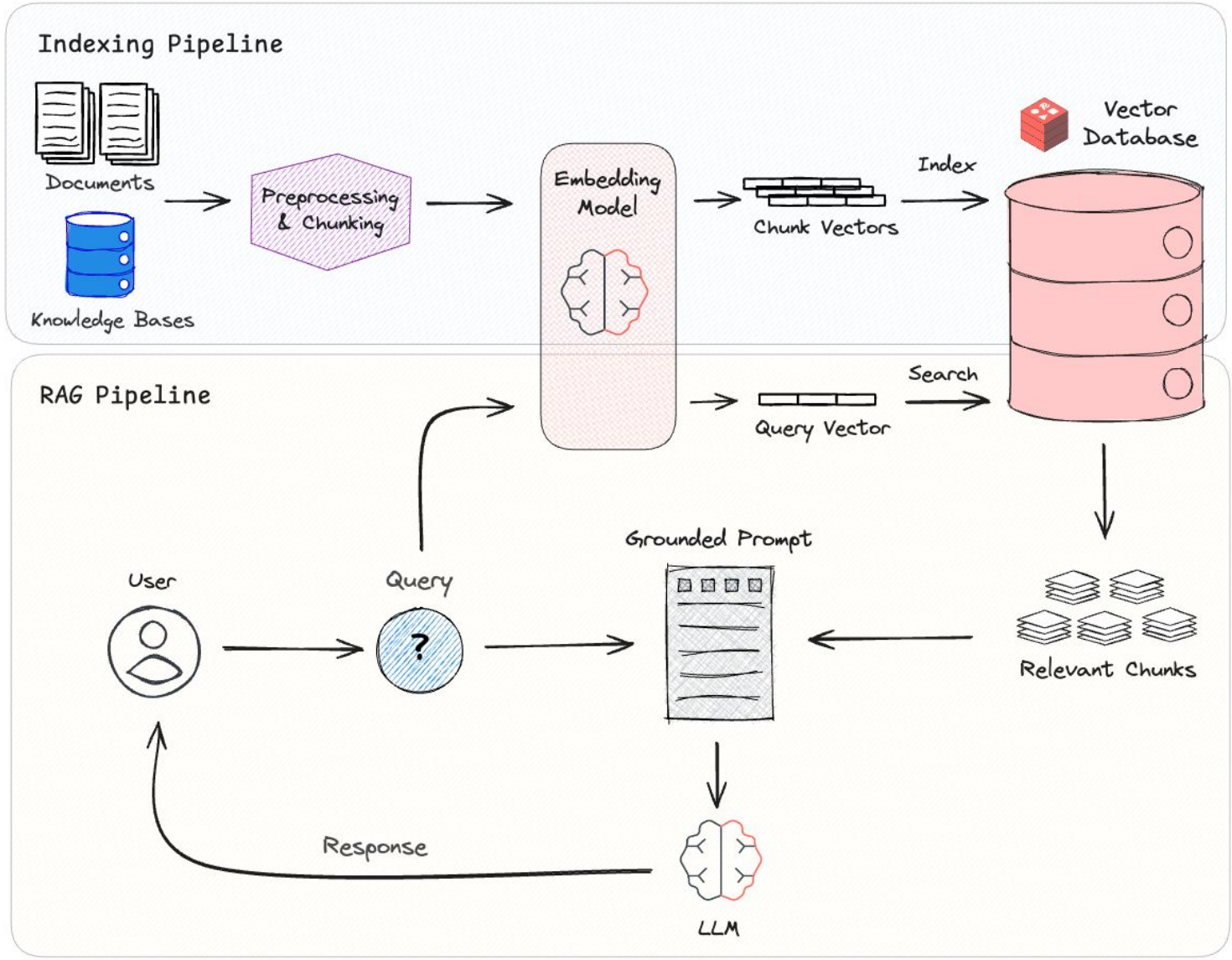
- Choose best processing for a request

RAG with Redis

Retrieval Augmented Generation

Retrieve context powered by vector search + hybrid search.

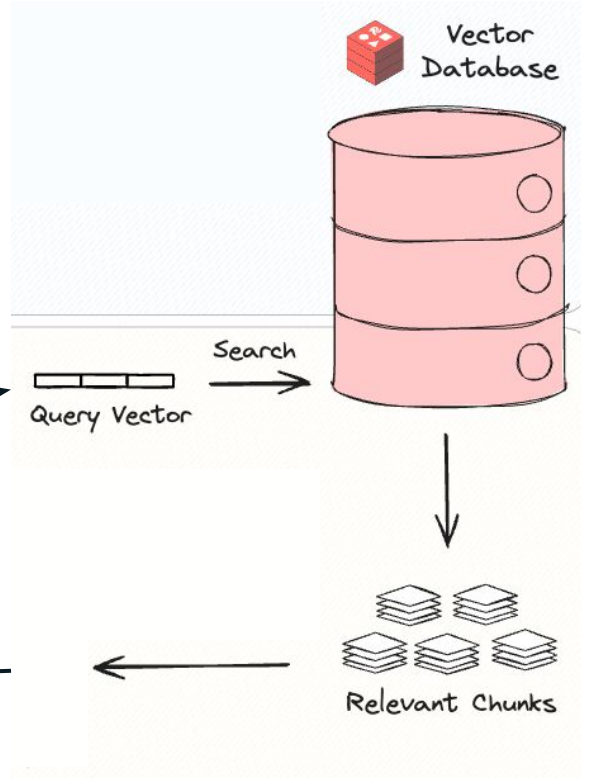
Augment LLM context and **Generate** a factual, relevant response.



Context retrieval for Retrieval Augmented Generation (RAG)

Grounded Prompt

```
def promptify(query: str, context: str) -> str:
    return f'''Use the provided context below
    derived from public financial documents to
    answer the user's question. If you can't
    answer the user's question, based on the
    context; do not guess. If there is no context
    at all, respond with "I don't know".
    User question:
    {query}
    Helpful context:
    {context}
    Answer:
    '''
```



Build with native clients libraries & supported ecosystem integrations

RedisVL

Powerful, dedicated Python client library for using Redis in AI native workloads

$ pip install redisvl

Ecosystem


Langchain


LlamaIndex


Haystack


Spring AI

AWS Bedrock

One out of four options to be integrated with AWS Bedrock Knowledge bases

....and more coming soon.

Lab 2: RAG from scratch

02_redisvl_basic_rag.ipynb

1. Instantiate a text splitter and use it to “chunk” a large document
2. Vectorize the chunks
3. Test your data with a basic vector search
4. Use vector search to create a RAG pipeline
5. Create an async search index and used it to retrieve context for an augmented prompt
6. Send the query using the OpenAI API

Lab Debrief

- Lab 2

In this lab you

- Instantiated a text splitter and used it to “chunk” a large document
- Vectorized the chunks
- Reviewed basic vector search syntax (KNN)
- Used that syntax to create a RAG pipeline
- Created an async search index and used it to retrieve context for an augmented prompt
- Sent the query using the OpenAI API

More use cases

For vector search and RAG

- **Fraud detection**
  - Vector search can be used to classify user behaviors when properly modeled as vectors
- **Personalization of product description**
  - Based on semantic matching, the user will read a product description that highlights aspects of the product matching user preferences
- **User segmentation**
  - Semantic matching enables the creation of categories of users to boost the relevance of recommendations
- **Contact center analytics**
  - Vector search helps retrieve historical tickets to assist with incoming tickets
- **Recommendations**
  - Semantic search helps find similar items (products, documents) based on the aspect or description

- Going beyond simple RAG to become…

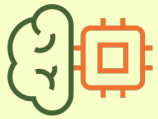
Production ready

Dealing with the gaps

Common design patterns

Retrieval Augmented Generation

- Current documents provide context to LLM

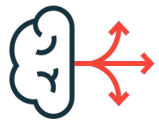


Semantic Caching

- Return cached responses; save time & money.

LLM Memory

- Include relevant history as context for LLM

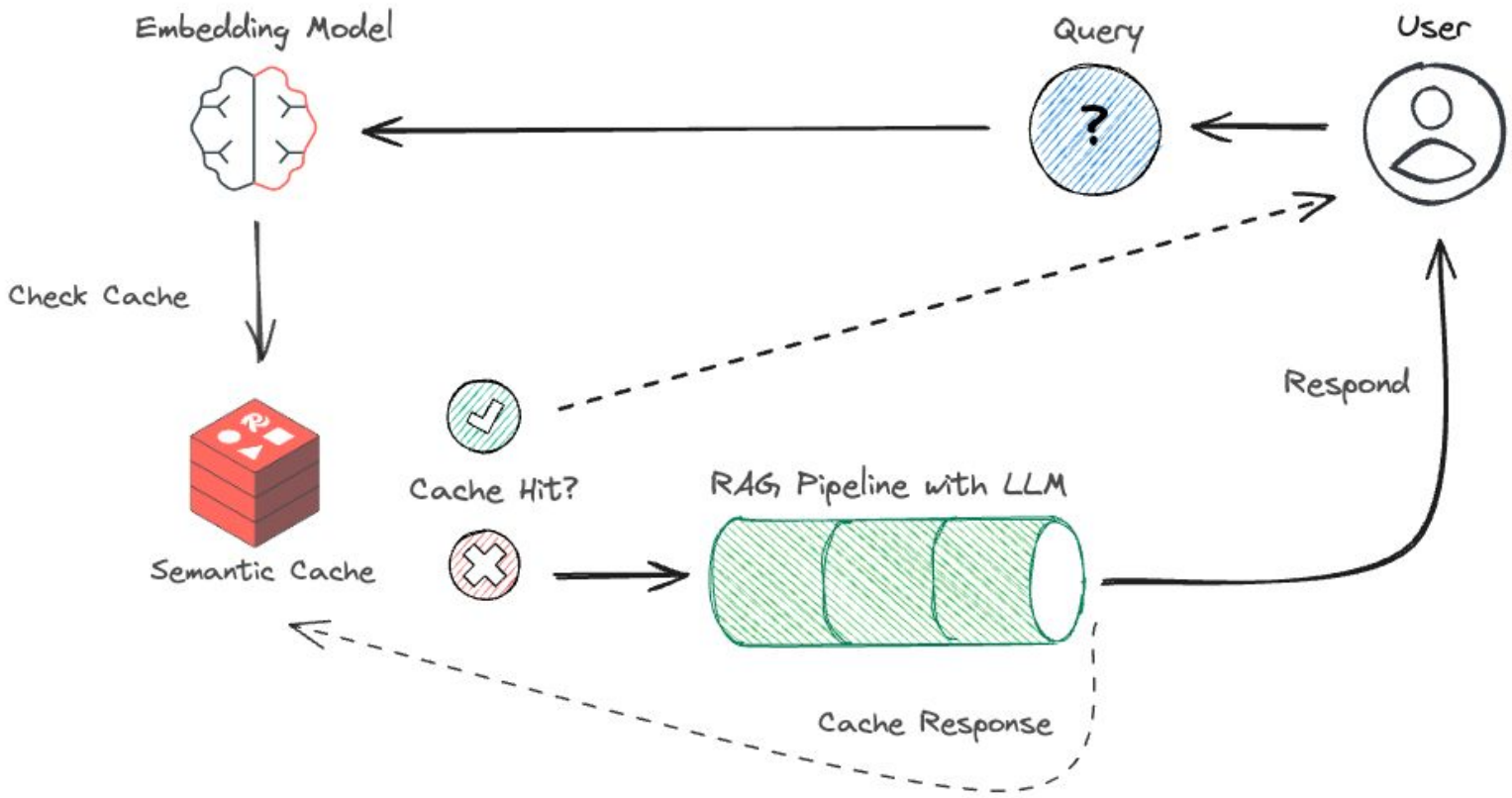


Semantic Routing

- Choose best processing for a request

Semantic caching

RAG systems often exhibit large numbers of repeated questions (FAQs).
Use semantic cache to re-use previously answered questions.
Powered by Redis caching + vector search features.



~30% of user's queries are similar to at least one previous query by the same user https://arxiv.org/pdf/2403.02694

Semantic Caching syntax in RedisVL

Using `class SemanticCache()`

Define the `SemanticCache` and provide these values:

- `vectorizer` – the vectorizer for the cache
- `ttl` – time-to-live for cached records
- `distance_threshold` – semantic threshold for the cache
- `filterable_fields` – optional list of RedisVL fields that can be used to customize cache retrieval with filters

```
llmcache = SemanticCache(
    name="llmcache",
    vectorizer=hf,          # optional
    redis_url=REDIS_URL,
    ttl=120,                # optional
    distance_threshold=0.2,# optional
)
```

Store a query in the Semantic Caching

Can be synchronous: `store()` or asynchronous: `astore()`

Invoke `store` or `astore` and provide these values:

- `prompt` – the prompt string
- `vector` – the vectorized prompt (optional) defaults to None, and the prompt vector is generated on demand
- `ttl` – optional ttl for this entry
- `metadata` – optional

```
llmcache.store(
    prompt=client_input
    response=llm_output,
    metadata={"user": "dent42”,
    "level": "basic"}
)
```

Check the Semantic Caching for a query

Can be synchronous: `check()` or asynchronous: `acheck()`

Invoke check or acheck and provide these values:

- `prompt` – the prompt to search for
- `vector` – the vectorized prompt to search for (optional)
- `num_results` – (optional) default is 1
- `distance_threshold` – to use (optional)

```
response = await llmcache.acheck(
    prompt=new_input
)
```

Semantic caching in RedisVL

```python
from redisvl.extensions.llmcache import SemanticCache
# choose the fields to use as filters
private_cache = SemanticCache(
    name="private_cache",                                  # search index name
    redis_url='redis://localhost:6379',                    # redis connection
    distance_threshold=0.2,                                # semantic distance threshold
    filterable_fields=[{"name": "user_id", "type": "tag"}]  # custom tags to add to docs
)
private_cache.store(
    prompt="What is the phone number linked to my account?",
    response="The number on file is 123-555-0000",
    filters={"user_id": "abc"}, # set the value of the field when inserting
)
private_cache.store(
    prompt="What's the phone number linked in my account?",
    response="The number on file is 123-555-1111",
    filters={"user_id": "xyz"}, # set the value of the field when inserting
)
```

```python
from redisvl.query.filter import Tag
# define a Tag filter on our filterable field
user_id_filter = Tag("user_id") == "abc"

response = private_cache.check(
    prompt="What is the phone number linked to my account?",
    filter_expression=user_id_filter,   # limit results to matching user_id
)
```

Advanced cache privacy controls (filters) prevent data leakage between users

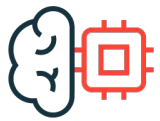
Filter logic enables customizable access levels

Dealing with the gaps

Common design patterns

Retrieval Augmented Generation

- Current documents provide context to LLM

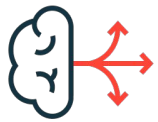


Semantic Caching

- Return cached responses; save time & money.

LLM Memory

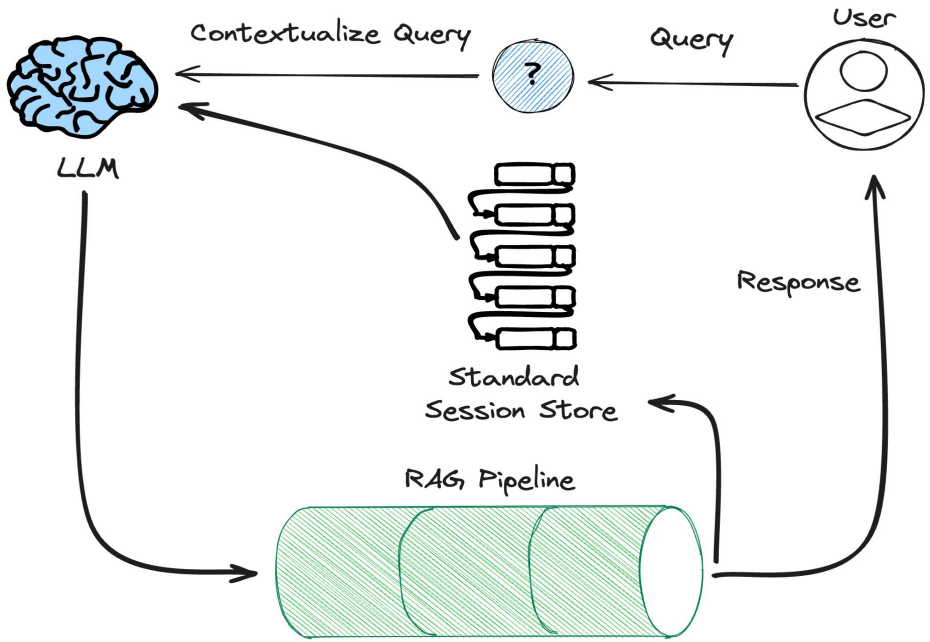
- Include relevant history as context for LLM

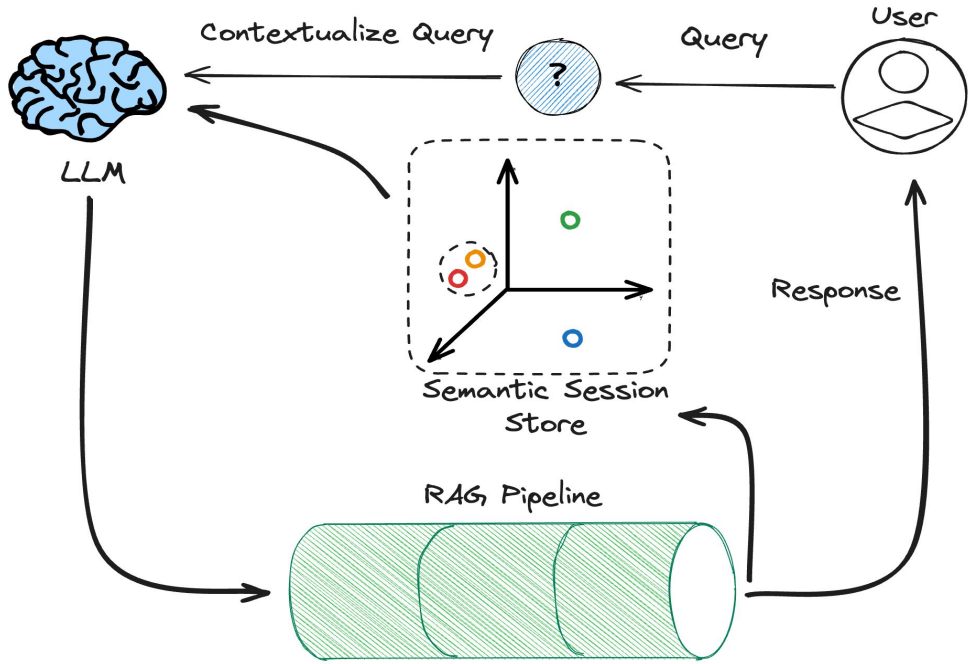


Semantic Routing

- Choose best processing for a request

Short-term Memory (aka session management)





List based sessions append the full or recent chat history to each query.

- Increased token count leads to higher costs and latency
- Information is "lost in the middle" with large contexts ("*Lost in the Middle: How Language Models Use Long Contexts*")

Semantic sessions append only the relevant chat history to each query.

- Vector search provides a flexible context window
- Removing irrelevant context can improve LLM responses

Short and long-term Memory in RedisVL

```python
from redisvl.extensions.session_manager import SemanticSessionManager

semantic_session = SemanticSessionManager(name='tutor')
```

Set up session manager

```python
prompt = "what is the size of England compared to Portugal?"
response = "England is larger in land area than Portal by about 15000 square miles."
semantic_session.store(prompt, response)
```

Add messages and their source - user, llm or tool

```python
chat_session.add_messages([
    {"role":"user", "content":"What is the capital of France?"},
    {"role":"llm", "content":"The capital is Paris."},
    {"role":"user", "content":"And what is the capital of Spain?"},
    {"role":"llm", "content":"The capital is Madrid."},
    {"role":"user", "content":"What is the population of Great Britain?"},
    {"role":"llm", "content":"As of 2023 the population of Great Britain is approximately 67 million people."},]
    session_tag='student one')
```

Session tags enable user privacy in one session manager

```python
prompt = "what have I learned about the size of England?"
semantic_session.set_distance_threshold(0.35)
context = semantic_session.get_relevant(prompt, session_tag='student one')
```

```
{'role': 'user', 'content': 'what is the size of England compared to Portugal?'}
{'role': 'llm', 'content': 'England is larger in land area than Portal by about 15000 square miles.'}
```

See all conversation history,recent history,
or semantically relevant history

Lab 3: RAG w/ Caching and Memory

03_redisvl_production_rag.ipynb

1. Instantiate and configure a SemanticCache
2. Test the RAG with the cache
3. Implement an indexed chat history
4. Bring the code from previous labs together to create a production-ready RAG

Lab Debrief

- Lab 3

In this lab you

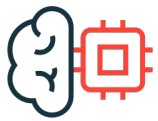
- Instantiated and configured a SemanticCache
- Tested the RAG with the cache
- Implemented an indexed chat history
- Brought the code from previous labs together to create a production-ready RAG

Dealing with the gaps

Common design patterns

Retrieval Augmented Generation

- Current documents provide context to LLM

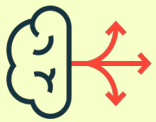


Semantic Caching

- Return cached responses; save time & money.

Long-term Memory

- Include relevant history as context for LLM



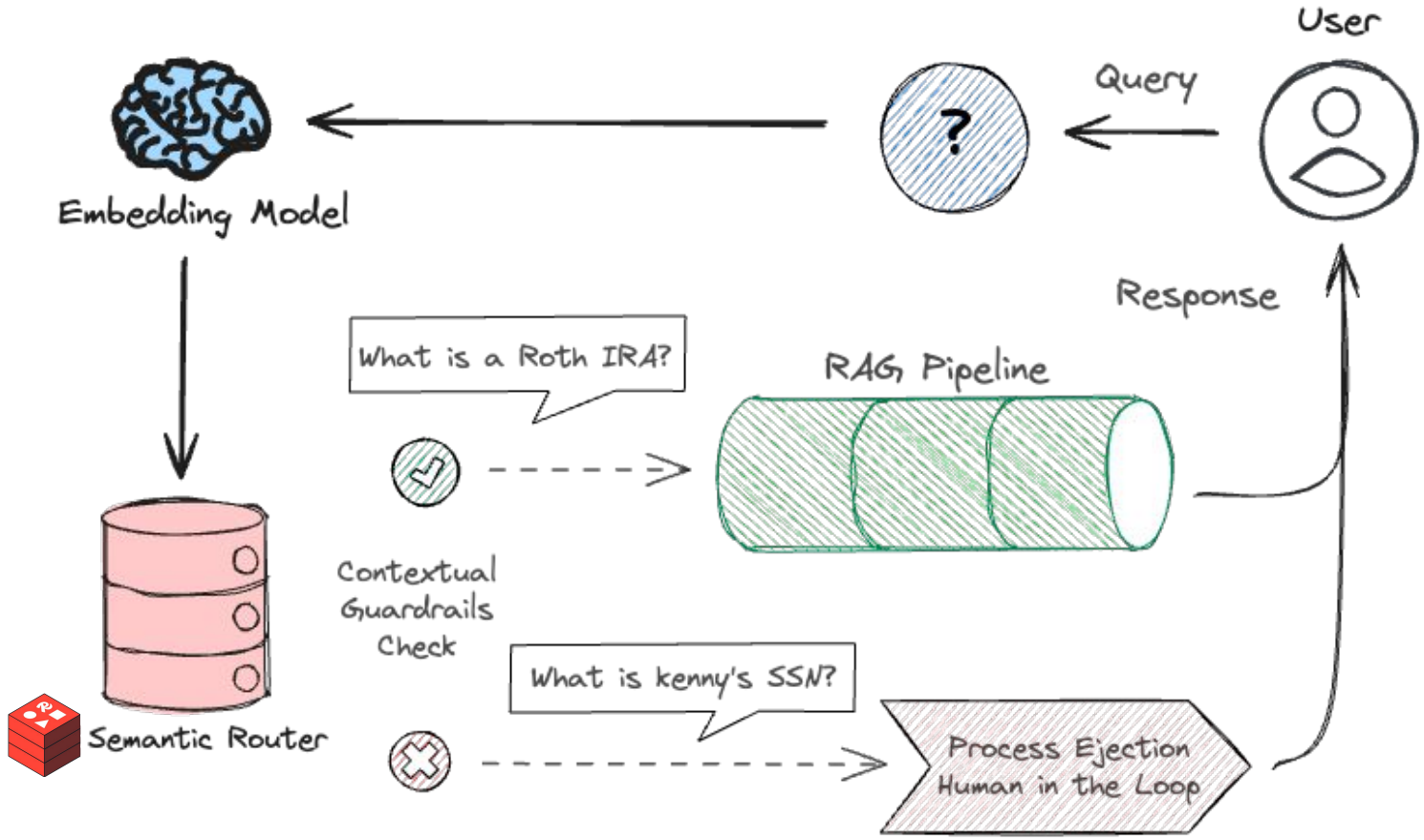
Semantic Routing

- Choose best processing for a request

Semantic routing

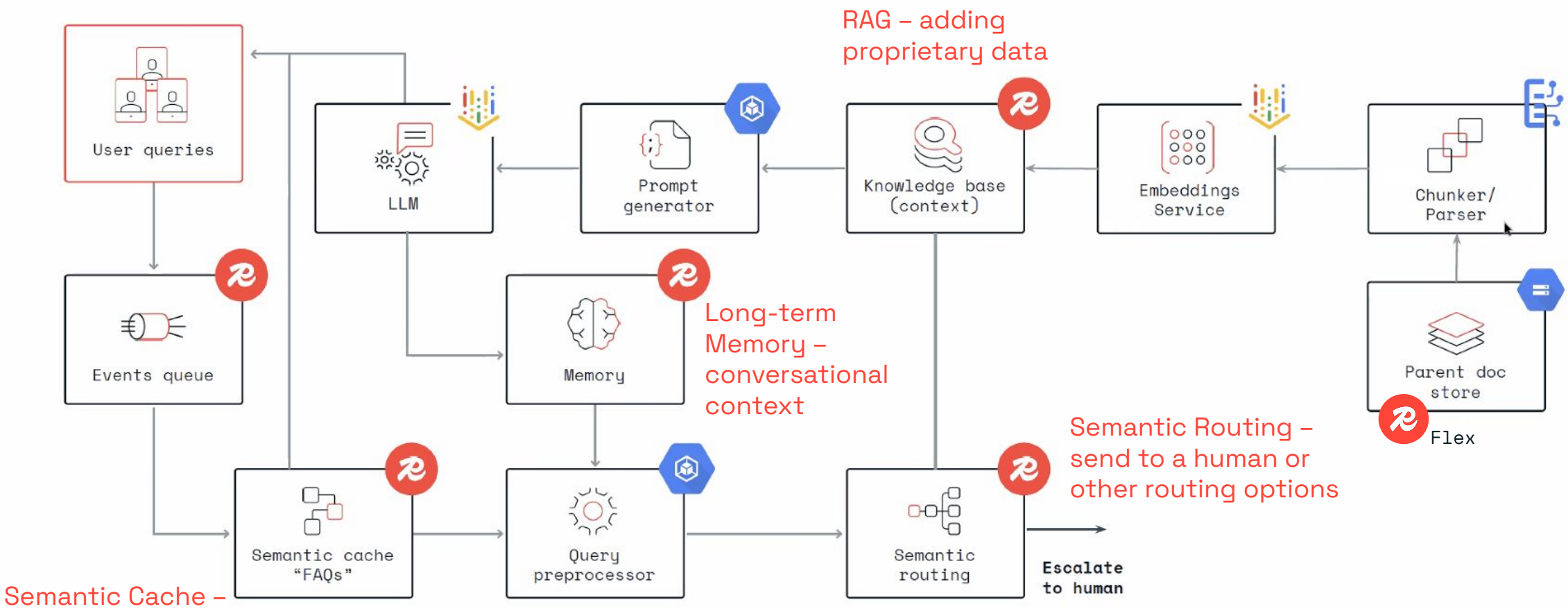
Assign incoming queries to proper handlers based on semantics:

- **LLM selection:** choose the right LLM for the given task
- **Guardrails**: Prevent access to undesirable topic areas
- **Data segregation**: route queries to appropriate data sources



Pushing RAG forward.

Generative apps optimized for speed, accuracy, & cost

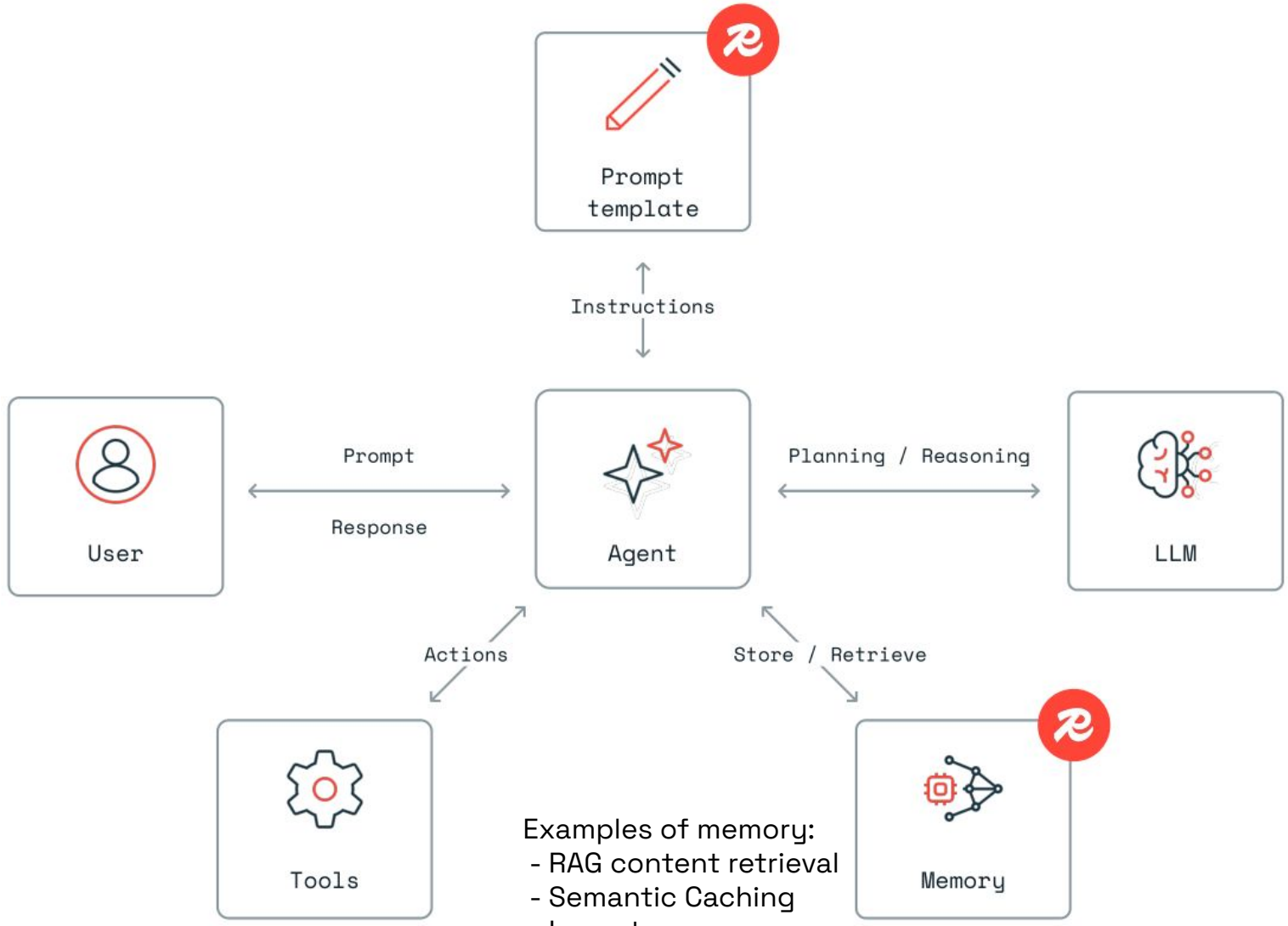


Redis and Agentic AI

Don’t just stand there. Do something!

AI Agents

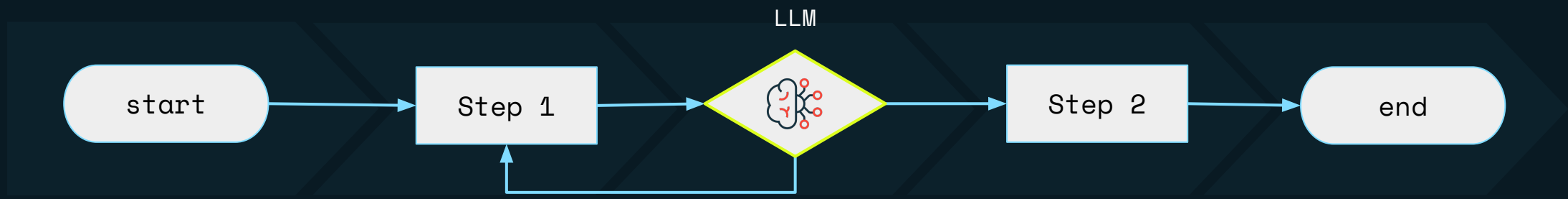
Bring fast memory to agents



Examples of memory:
- RAG content retrieval
- Semantic Caching
- Long-term memory
- Agent state (short-term memory)

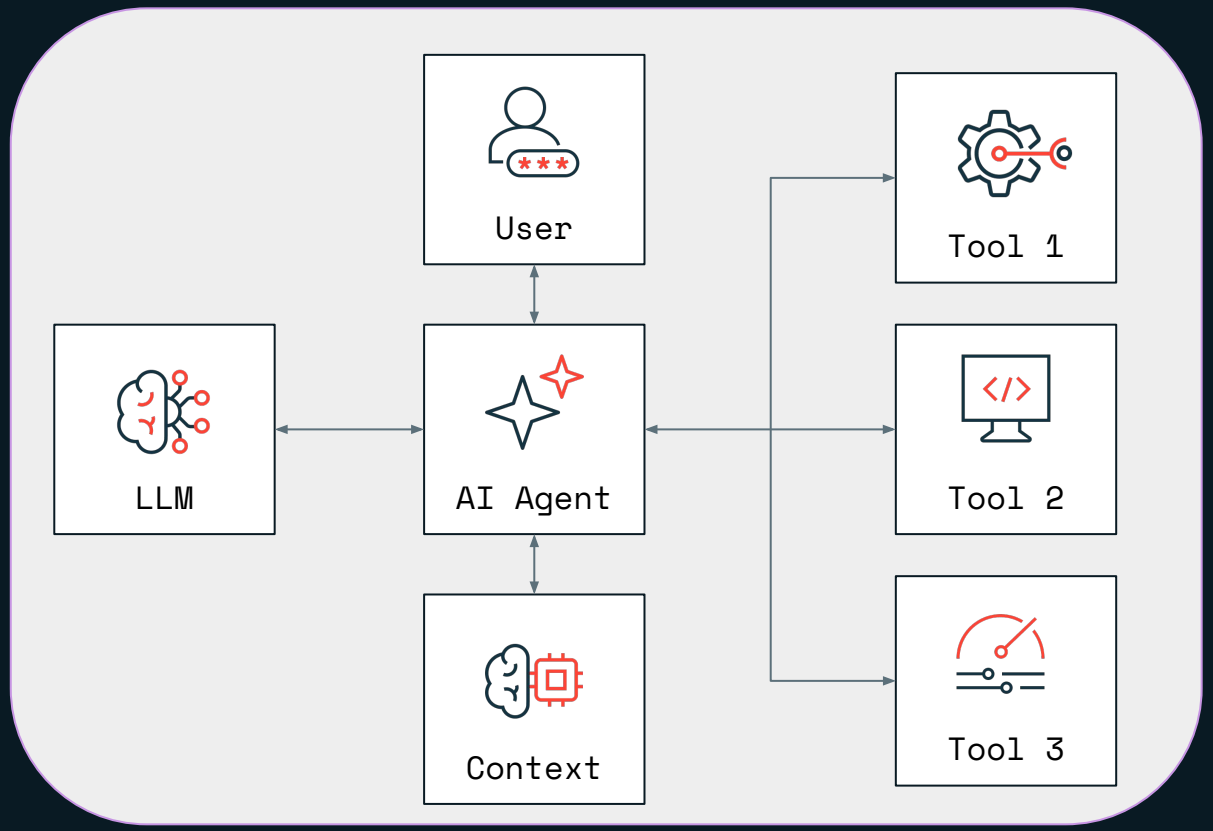
Agentic Workflows

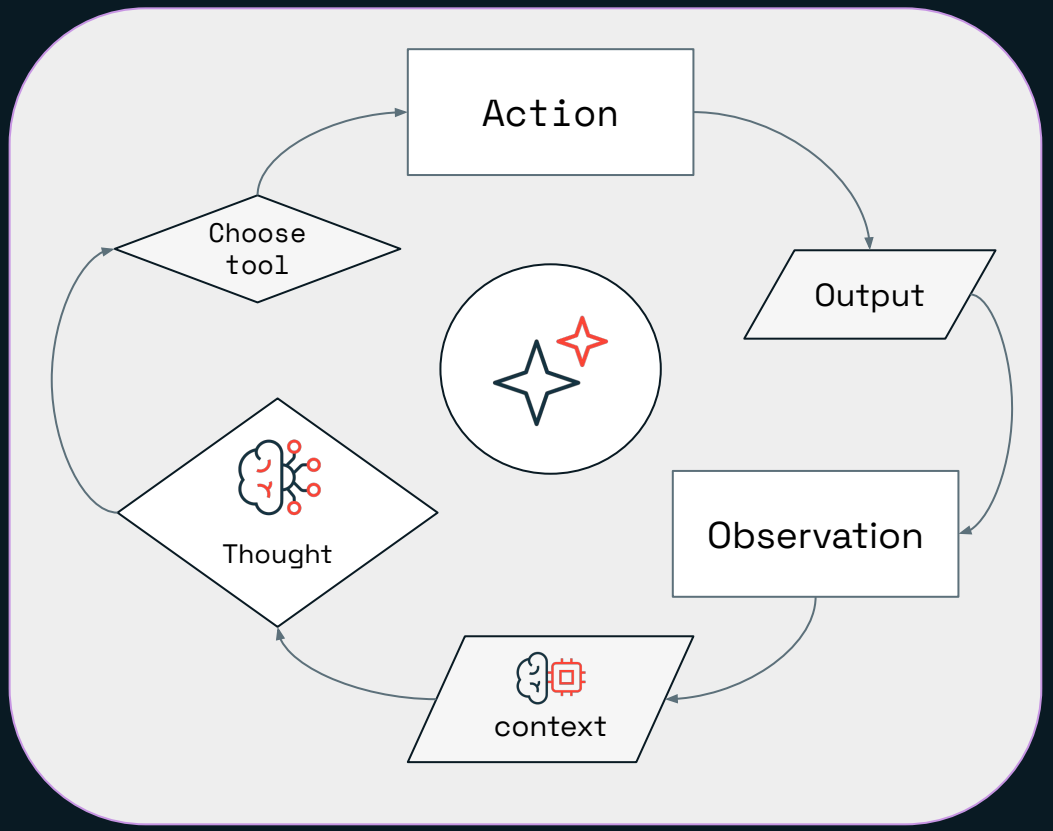
When LLMs are used to control the flow we have a modern “Agentic” application



ReAct Approach

For-loop-like combining reasoning+actions+feedback





Agentic memory optimizes multi-step chains.

Fast

Retrieve relevant info from memory to accelerate agents.

Integrated

Session management that is fully managed for AI agents interactions.

Scalable

Use for any number of agents and agent chains in production apps.

LangChain stores memory for AI agents using Redis Cloud.

Industry: Tech
Country: USA

About LangChain

LangChain is the leading framework for GenAI designed to simplify the creation of applications using large language models. As a LLM integration framework, LangChain's use-cases include document analysis and summarization, chatbots, and code analysis.

Click or Scan for full story

Challenge

LangChain needed a fast and efficient memory solution for agent configurations that don't slow down LLM applications, and worked over a variety of use cases.

Solution

Redis Cloud provides storage for a wide variety of configurations and data relevant to LLM conversations and agent configuration, and works at scale.

Results

OpenGPTs has over 6k stars on Github and is used in a wide variety of industries for AI applications.

Redis has proved to be dependable and versatile solution for storing all the information to make these AI apps run.

LangChain

"We're using Redis Cloud for everything persistent in OpenGPTs including as a vector store for retrieval and as a database to store messages and agent configurations. The fact that you can do all of those in one database from Redis is really appealing."

Harrison Chase
CEO of LangChain

Q&A

Supplemental resources

Redis AI Resource Repo

https://github.com/redis-developer/redis-ai-resources

Look here to find example code for what our Applied AI team is working on.

RedisVL

https://github.com/redis/redis-vl-python

The Redis Vector Library codifies best practices and makes it easier to get going in python.

LangChain & LlamaIndex

https://github.com/redis/redis-vl-python

The Redis Vector Library codifies best practices and makes it easier to get going in python.

Give us your feedback

This helps us improve the experience for future workshops

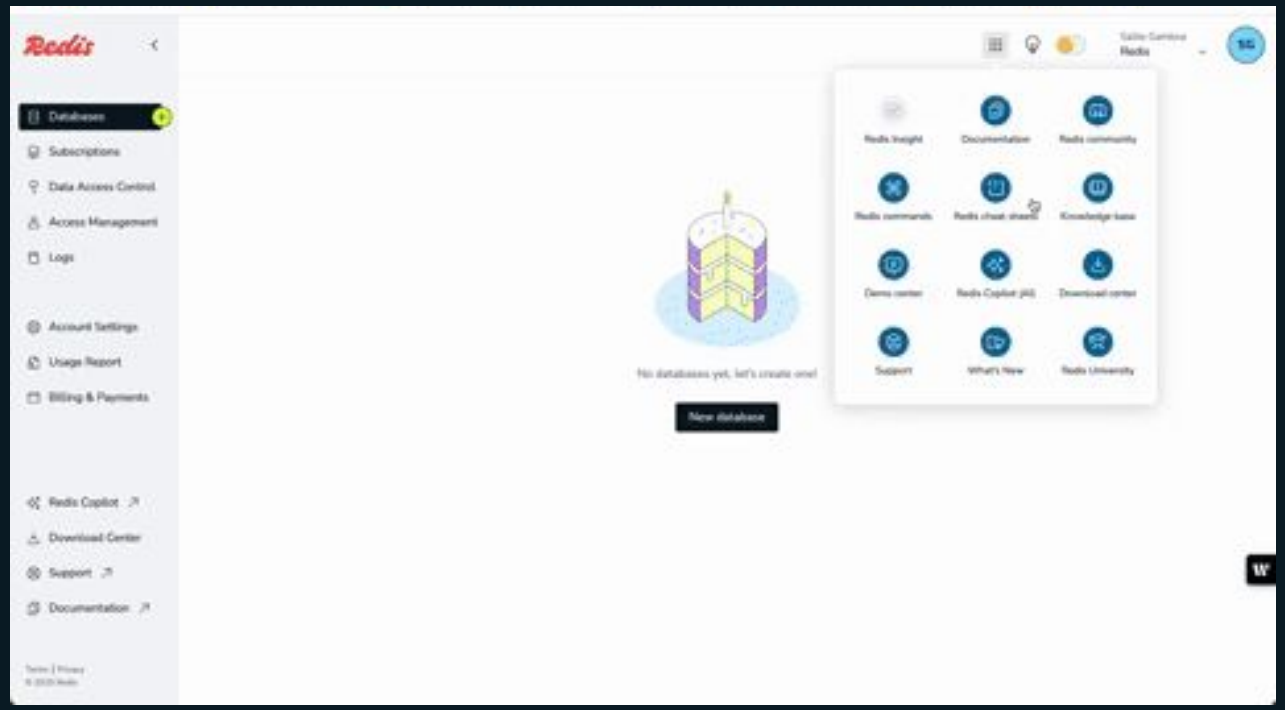
Direct Link

tinyurl.com/rrsfai

QR Code

Redis University

redis.io/university



✅ Easy | Single sign in with Redis cloud account

✅ Free | To all Redis users

✅ Available 24x7 | Self-paced, on demand

✅ More hands on | Exercises

For a deeper dive

>> Take the full Redis for AI learning path <<

Certificate of completion

1. Check your email
2. Create Redis University account
3. Certificate in your profile
4. Share to Linkedin

Thank you.