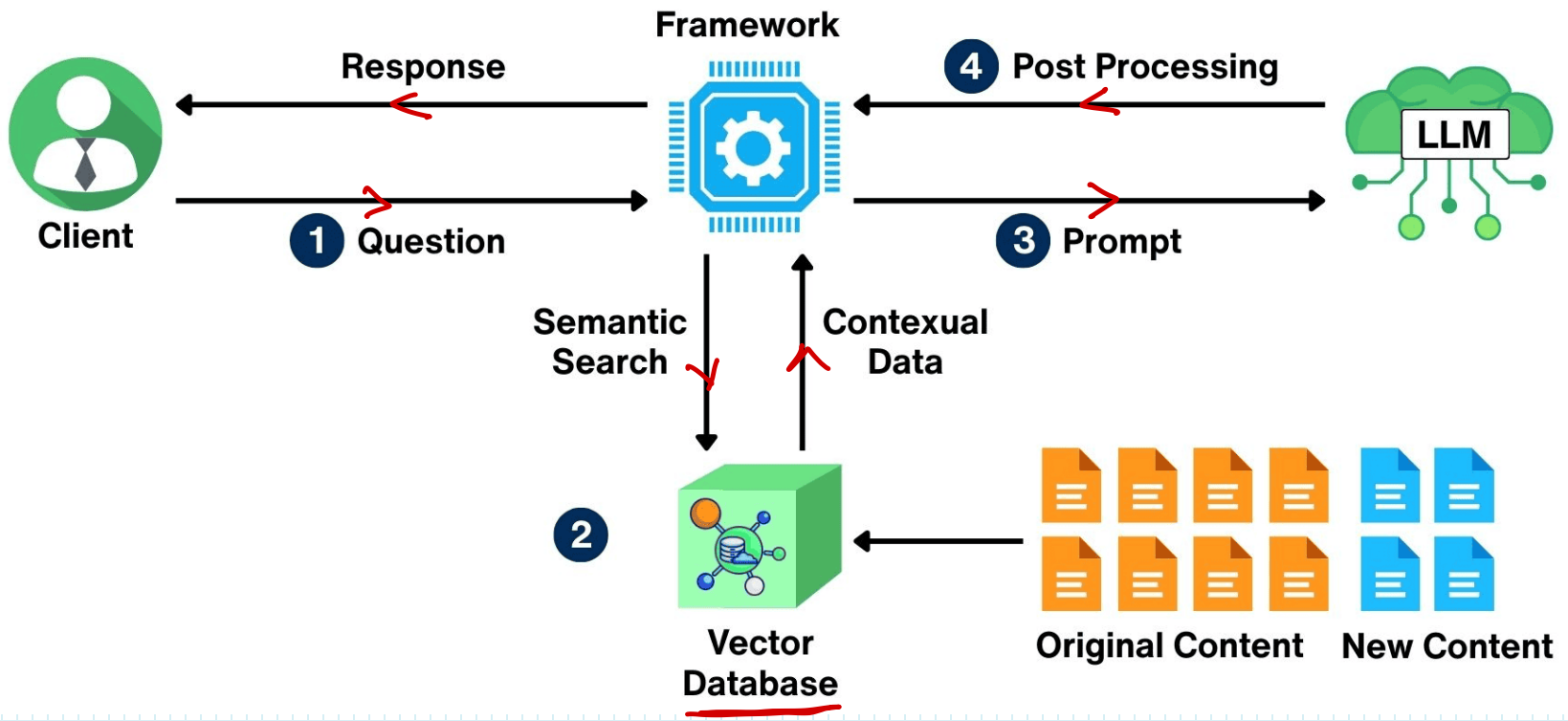
Generative AI

Trainer: Nilesh Ghule

Building Foundations for RAG

- RAG is "Retrieval-Augmented Generation".
  - AI framework that enhances LLMs by providing them relevant data sources to provide more accurate, and specific answers.
  - Prevents outdated info and hallucinations without model retraining/fine-tuning.
- RAG Foundations
  - Embeddings
  - Vector databases
- RAG Prerequisites
  - Using LLM and Agents
  - Prompt engineering



What Are Embeddings?

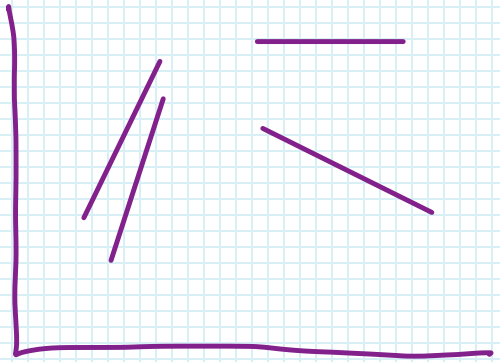
- Core Idea: A way to turn meaning into numbers.
- Definition: A vector (list of numbers) that represents the semantic meaning of text, images, audio, etc.
- Key Property: Similar meanings yield vectors that are close together in high-dimensional space.

```
"cat"  → [0.12, -0.91, 0.44, 0.03, ...]
"dog"  → [0.10, -0.88, 0.47, 0.05, ...]
"car"  → [-0.77, 0.22, -0.11, 0.90, ...]
```

- Embeddings translate semantics into a language computers understand: math
- Notice 'cat' and 'dog' have similar vectors; 'car' is far away. This simple 'trick' is the foundation of semantic search.

Why Embeddings? Computers Understand Geometry

- Computers understand: Numbers, Math, Distance.
- Embeddings allow us to:
  - Convert meaning → into math (vectors)
  - Compare meaning → by calculating distance between vectors



- Without embeddings, you're stuck with keyword matching. With embeddings, you enable semantic search.
- Critical Mindset:
  - Embeddings do NOT store facts, reason, or answer questions.
  - They ONLY represent meaning and allow similarity comparison.
- Note:
  - Embeddings create a 'semantic map.' LLMs later reason over points on that map.
  - Embeddings are not intelligent. They are just capturing meaning as spatial relationships.

Embeddings in Action: Similarity is Geometry

- We measure similarity using Cosine Similarity.
- It measures the angle between vectors, not their magnitude. We care about direction, not length.

$$\frac{x \cdot y}{\|x\| \cdot \|y\|}$$ ← cosine similarity betn vectors x & y

```
import numpy as np

def cosine_similarity(a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

cat = np.array([0.2, 0.9, 0.1])
dog = np.array([0.25, 0.85, 0.15])
car = np.array([-0.8, 0.1, 0.9])

print("cat vs dog:", cosine_similarity(cat, dog)) # ~1 (close)
print("cat vs car:", cosine_similarity(cat, car)) # ~0 or negative (far)
```

- Cosine similarity is the magic. A value of 1 means identical direction. ~0 means orthogonal (unrelated). Negative means opposite direction.

From Math to Real Text Embeddings

- Use a model (like sentence-transformers) to generate embeddings from real text
- The model outputs high-dimensional vectors (e.g. 384, 768 dimensions).
- Each dimension encodes an abstract semantic feature (topic, sentiment, style) learned by the model.

```
from sentence_transformers import SentenceTransformer
import numpy as np

model = SentenceTransformer('all-MiniLM-L6-v2') # 384 dimensions
sentences = ["I love football", "Soccer is my favorite sport", "I enjoy cooking pasta"]
embeddings = model.encode(sentences)

# Compare
sim_1_2 = cosine_similarity(embeddings[0], embeddings[1]) # High (~0.8)
sim_1_3 = cosine_similarity(embeddings[0], embeddings[2]) # Low (~0.1)
```

- Note: Even without shared keywords ('football' vs 'soccer'), the vectors are close. The model captures the semantic overlap (sports). This is the magic - it works across paraphrases and even languages.

Embedding Model Selection

- A good model places similar text close, separates different meanings, and works across paraphrases.
- Trade-offs: Dimensionality, Model Size, Speed, Hardware Needs, Training Data
- It does not mean:
  - More parameters = always better
  - Larger dimension = always better
- Note: Embedding quality is about retrieval behavior, not intelligence.
- Critical Rule:
  - One collection = one embedding model.
  - Never mix models in the same collection.
  - Note: Mixing models is like using two different maps with different scales—distances become meaningless.
- There is no universal 'best' model.

all-MiniLM-L6-v2 vs nomic-embed-text-v1.5

Feature	all-MiniLM-L6-v2	nomic-embed-text-v1.5
Dimensions	384	~768+
Speed	Very Fast (CPU-friendly)	Slower (GPU preferred)
Semantic Depth	Medium (General-purpose)	High (Technical, nuanced)
Best For	Prototyping, <100k chunks, low latency	Technical docs, large corpus, subtle distinctions
Mental Model	Wide-angle lens (fast, general)	Telephoto lens (precise, slower)

- MiniLM is your reliable baseline. If retrieval fails with it, the problem is likely your chunking or logic, not the model.
- Nomic gives you more resolution when you need to distinguish fine-grained meanings.

Using Embedding models

```
# Cosine Similarity = (A · B) / (||A|| * ||B||)
def cosine_similarity(a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
```

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2') # 384 dimensions
sentences = ["I love football", "Soccer is my favorite sport", "You are irritating."]
embeddings = model.encode(sentences)
```

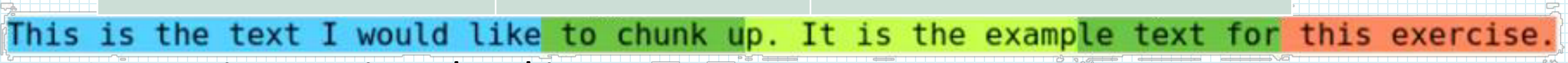
```
from langchain_openai import OpenAIEmbeddings

embeddings_model = OpenAIEmbeddings(
    base_url="http://localhost:1234/v1",
    api_key="dummy-key",
    model="text-embedding-nomic-embed-text-v1.5",
    check_embedding_ctx_length=False
)
sentences = ["I love football", "Soccer is my favorite sport", "You are irritating."]
embeddings = np.array(embeddings_model.embed_documents(sentences))
```

Chunking

- Splitting source text into smaller, meaningful units before embedding.
- Why it's necessary?
  - Embedding models have context limits (e.g., 512 tokens).
  - An embedding represents the average meaning of its input text.
  - A 20-page PDF as one embedding becomes a 'blurry' vector. Retrieval fails.
- Chunking - Real-life analogy
  - **Cutting a textbook into flashcards**
  - Each flashcard must:
    - Make sense alone
    - Be small enough to find
    - Be big enough to explain

Chunking Strategy

Chunk Size	Pros	Cons
Small (100-300 tokens)	High precision	Loses broader context
Medium (300-600)	Best balance	May need **overlap**
Large (800+)	More context	Poor recall, blurry meaning



- Semantic vs Naive Chunking
  - Naive (Bad): Split every N characters. Break sentences randomly
  - Semantic (Better): Split by: paragraphs, headings, sentences.
  - Best practice: Respect natural language boundaries first, size second
- Chunk Metadata (Very Important)
  - Always store metadata with each chunk
  - metadata = { "source": "doc_12.pdf", "page": 5, "chunk_id": 3 }
- Chunks are automatic knowledge units.

Chunking Examples with Langchain v1

```python
# 1. Basic Fixed-Size Chunking
from langchain_text_splitters import CharacterTextSplitter
text_splitter = CharacterTextSplitter(chunk_size=500, chunk_overlap=50)
docs = text_splitter.create_documents([raw_text])
# Simple, but can split sentences unexpectedly
```

```python
# 2. Recursive Character Chunking
from langchain_text_splitters import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(chunk_size=800, chunk_overlap=100,
separators=["\n\n", "\n", " ", ""])
docs = text_splitter.create_documents([raw_text])
# Good for mixed formatting (paragraphs, sentences, bullet points)
# It try paragraph → sentence → word boundaries
# And preserves semantic meaning better
```

```python
# 3. Token-Based Chunking
from langchain_text_splitters import TokenTextSplitter
text_splitter = TokenTextSplitter(chunk_size=300, chunk_overlap=50)
docs = text_splitter.create_documents([raw_text])
# Cares about token limits. For token priced models.
```

Chunking Examples with Langchain v1

```
# 4. Markdown-Aware Chunking
from langchain_text_splitters import MarkdownHeaderTextSplitter
headers_to_split_on = [ ("#", "Header 1"), ("##", "Header 2"), ("###", "Header 3") ]
text_splitter = MarkdownHeaderTextSplitter(headers_to_split_on=headers_to_split_on)
docs = text_splitter.split_text(markdown_text)
# Chunks align with semantic sections in Markdown document.
```

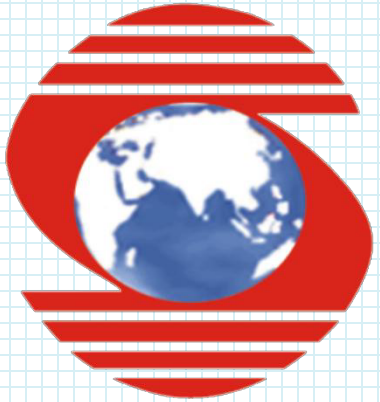
```
# 5. Code-Aware Chunking
from langchain_text_splitters import RecursiveCharacterTextSplitter
code_splitter = RecursiveCharacterTextSplitter.from_language(language="python",
chunk_size=1000, chunk_overlap=100)
docs = code_splitter.create_documents([code_text])
# Source code (Python, JS, Java, etc.) - Avoids breaking functions/classes
```

```
# 6. Sentence-Based Chunking (NLP-Style)
from langchain_text_splitters import SentenceTransformersTokenTextSplitter
text_splitter = SentenceTransformersTokenTextSplitter(chunk_size=256, chunk_overlap=20)
docs = text_splitter.create_documents([raw_text])
# Best for: Q&A datasets, Short factual text, High precision retrieval
```

Chunking - Practical problems & guidelines

- Chunking Failure Reasons (Common)
  - Chunk too large: Query hits irrelevant parts
  - Chunk too small: No context, Fragments retrieved
  - No overlap: Important info lost between chunks
  - Mixed topics per chunk: Embedding meaning becomes ambiguous

- If unsure about strategy, try these defaults
  - 300–500 tokens
  - 50–100 token overlap
  - Sentence-aware splitting
  - One topic per chunk



Thank You!

Nilesh Ghule

<nilesh@sunbeaminfo.com>