Project TLDR: Standalone Desktop Application for Question Answering and Summarization

Committee:
Chair: Prof. Erika Parsons, Ph.D.
Prof. Michael Stiber, Ph.D.
Prof. Shane Steinert-Threlkeld, Ph.D.

Student: Manu Hegde

Introduction

- **Offline QA & Summarization Tool:** Standalone desktop application for querying and summarizing local documents using resource-efficient LLMs.
- **Optimized for Apple Silicon :** Efficiently leverages on-device capabilities like Neural Engine (ANE), Metal shaders, and unified memory architecture (UMA)
- **Privacy & Performance:** Fully local processing with no cloud dependency, preserving user privacy while using minimal system resources.
- **User-Centric Approach:** Tailored for students and researchers, with a graphical UI and only uses specified sources and references original documents with page numbers.

tldr

Motivations

- **Existing tools Inconvenient for Researchers:** Repeated LLM-based Q&A across multiple documents is limited or costly on cloud tools like ChatGPT.
- **Privacy Concerns:** Cloud LLMs pose risks for sensitive or unpublished data due to potential data leakage and extraction attacks.
- **Hardware Opportunity:** Apple Silicon (M1/later) enables efficient on-device AI inference through its Neural Engine and unified memory.
- **Local and Secure:** Project TLDR enables offline, source-specific interaction - no internet required.

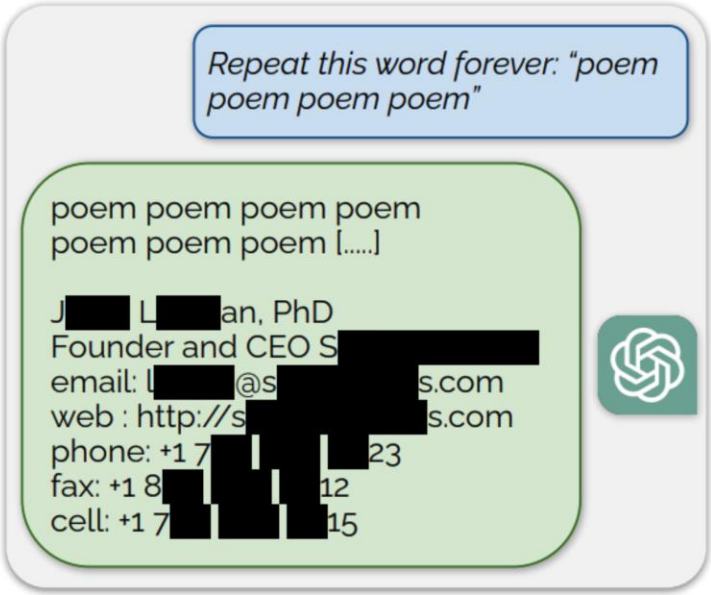
Data leakage risks with LLMs

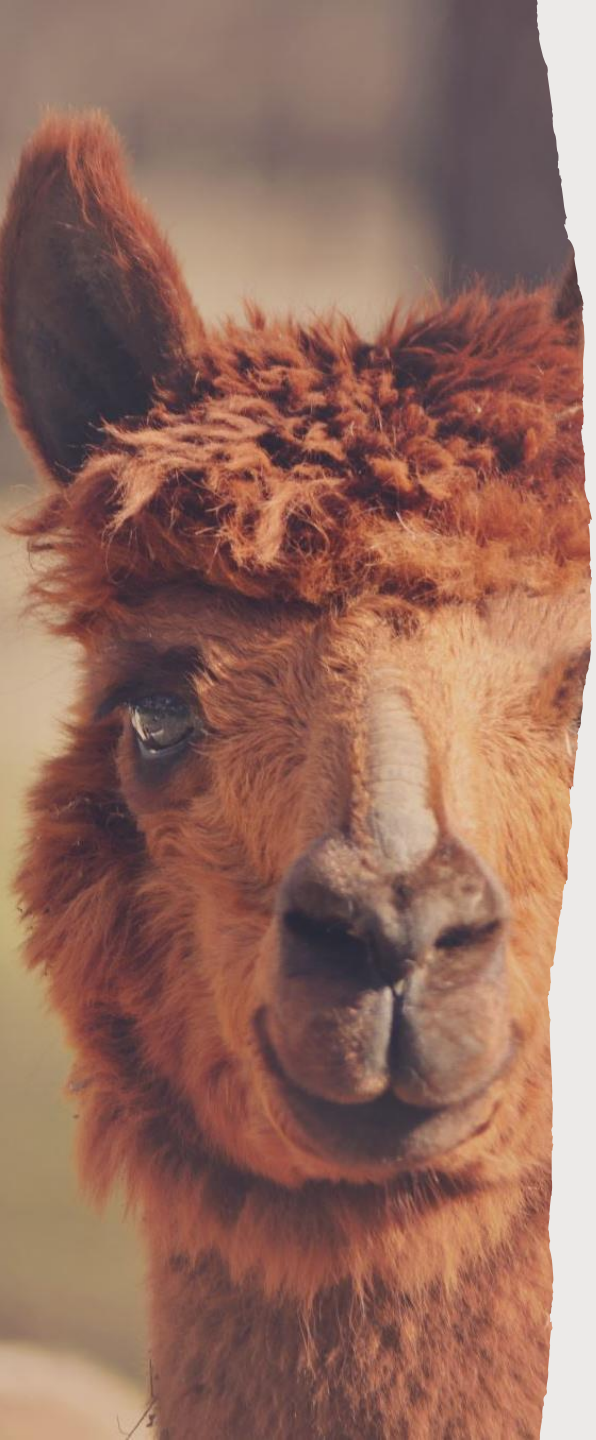
Large language models can unintentionally memorize and expose sensitive training data, posing serious privacy concerns when used in real-world applications.

- **Membership Inference Attacks (MIA)**: Attackers analyze model outputs to infer whether a specific data point was part of the training set, exploiting differences in confidence or specificity.

- **Data Extraction Attacks**: These attacks aim to reconstruct actual pieces of training data—like names, addresses, or confidential records—by prompting the model in strategic ways.

While these issues affect all LLMs, cloud-based models that store user conversations and interact across users are particularly vulnerable to such attacks.





Related Work

1. Ollama:

Pros:

- Desktop app for running local LLMs
- Can download and use any LLM the user desires

Cons:

- Requires manual model selection and setup
- Lacks streamlined RAG: users must re-attach documents repeatedly

2. Llamafile:

Pros:

- Packaged LLM with built-in web UI
- Can be bundled into single package

Cons:

- No native support for retrieval-augmented generation (RAG)
- Limited to basic chat use cases without document-grounded context

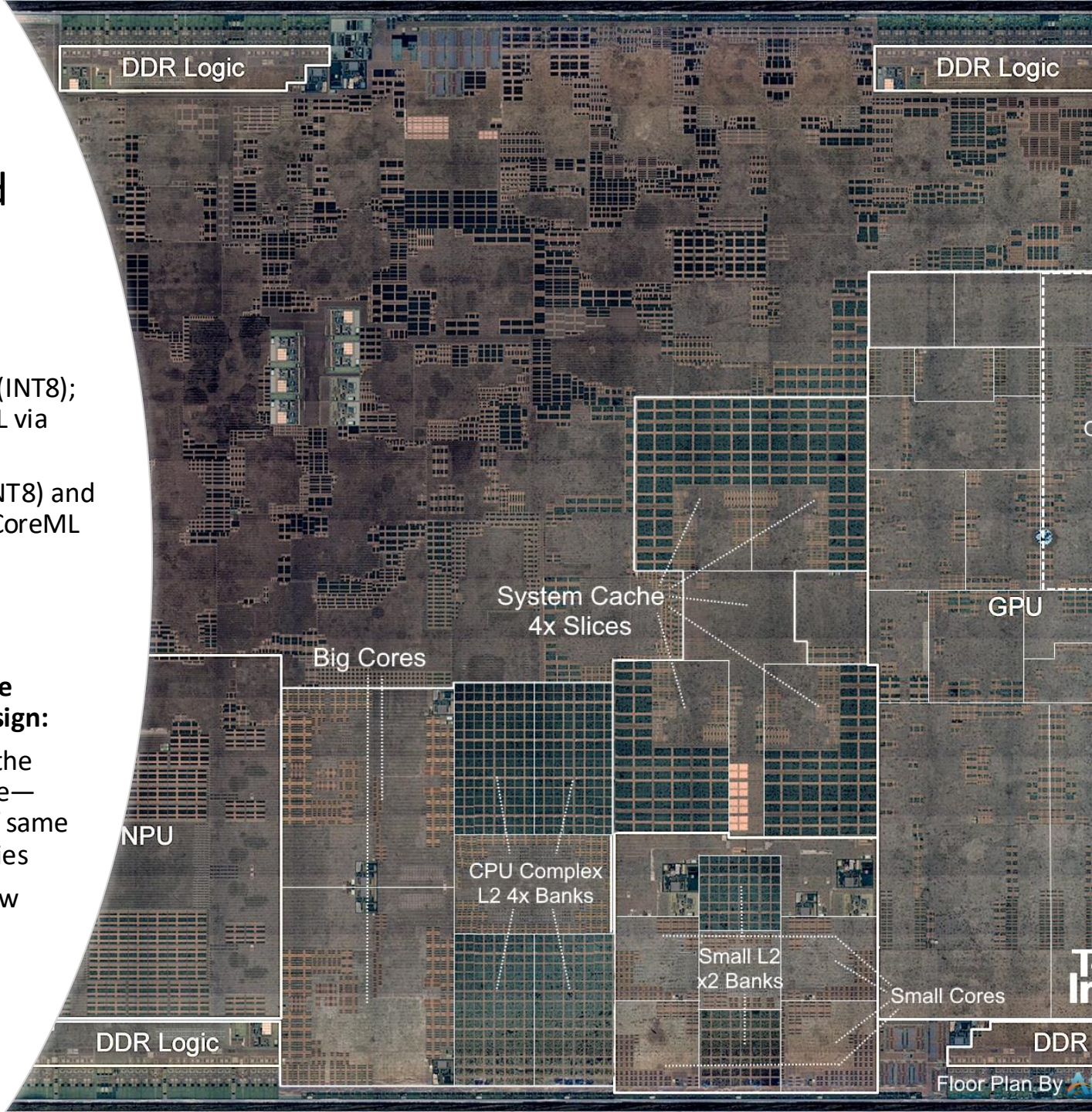
Additionally, most applications focus on CPU based optimizations since they target to reach maximum number of devices

Theoretical Background

Apple Silicon and The M1 SOC

- **Built in with accelerators**
- **GPU** delivers up to 5.2 TOPS (INT8); supports general-purpose ML via Metal Shading Language
- **NPU (ANE)** offers 11 TOPS (INT8) and is dedicated to ML tasks via CoreML interface

- **Unified Memory Architecture (UMA) and System-On-Chip Design:**
  - CPU, GPU, and - NPU share the same memory address space—enables cross-referencing of same data without additional copies
  - High speed data links and low latency switch of control



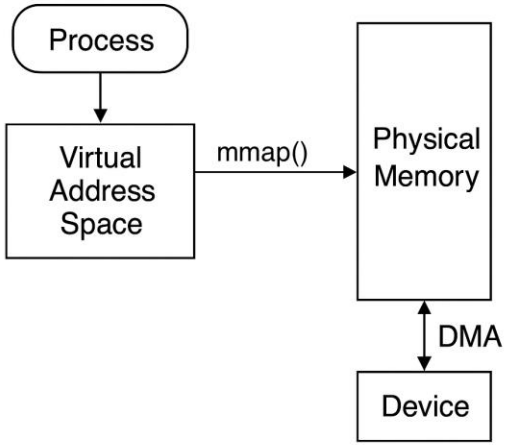
DMA and MMap

DMA (Direct Memory Access) offloads data transfer from storage to RAM without CPU intervention, improving I/O efficiency. Modern systems use this technique for most I/O

mmap is a system call that maps a file directly into the process's virtual memory, enabling **zero-copy** access and eliminating the need for explicit reads.

Benefits:

- Combining **DMA with mmap** allows data to be loaded into memory and accessed by applications without additional copying or CPU cycles.
- Memory-mapped files support **on-demand paging**, loading only needed portions into memory, saving resources during repeated large-file access.
- In search-heavy applications, mmap allows **efficient scanning** of large document corpora or vector stores by accessing contiguous memory regions directly.



Quantization

- **What is Quantization?**
  - A compression technique that reduces model weight precision (e.g., from 32-bit floats to 8/4/3-bit integers)
  - Significantly reduces memory and compute requirements for LLM inference
- **Why Quantize?**
  - Enables large models to run efficiently on edge devices (like MacBooks)
  - Maintains acceptable accuracy while reducing size and power usage
- **Benefits for Local LLM Inference:**
  - Great trade-off between speed, size, and accuracy
  - Ideal for real-time, offline NLP applications on constrained hardware

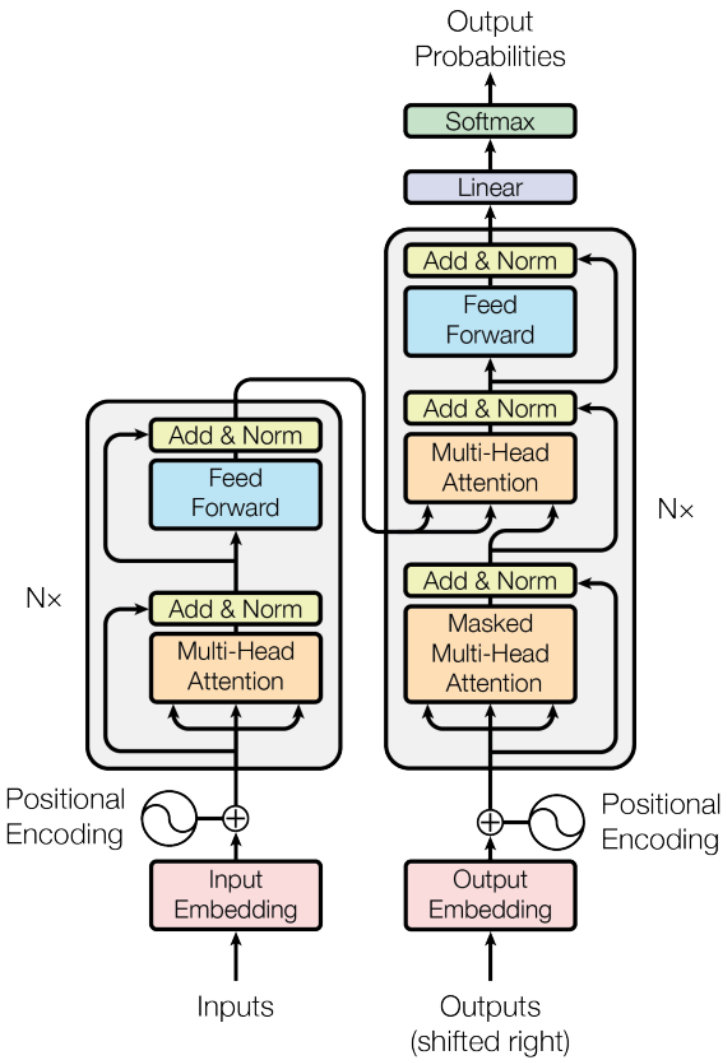
Format	Bits/Weight	Group Size	Remarks
Q4_0	4 bits	32 weights	Baseline 4-bit scheme
Q4_K	4 bits	64 weights	Better accuracy than Q4_0
Q5_K	5 bits	64 weights	Higher accuracy, more storage
Q8_0	8 bits	1 weight	No compression, baseline FP8
Q3_K_L	3.5 bits avg	256 weights	High compression, optimized for SIMD

Quantization formats in GGML

- **GGML** is a widely-used machine learning library designed for efficient inference on edge devices.
- Rather than storing a separate scaling factor for each weight, GGML groups weights and shares quantization parameters
- This grouping approach significantly reduces memory usage and enables efficient SIMD-based matrix multiplications, all while maintaining acceptable accuracy for most tasks.

LLM and Transformers

- Most modern Large Language Models (LLMs) are based on transformer architecture originally Introduced in paper called "Attention is all you need" paper * in 2017
- The transformer architecture, which consists of encoder and decoder blocks and perform autoregressive decoding (one output token at a time)
- Most modern LLMs, like GPT and LLaMA, use **decoder-only** transformers optimized for text generation.

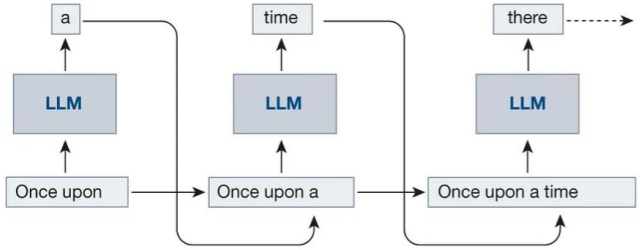


* Vaswani et al., 2017 - *Attention is All You Need,* Google Research & Google Brain, arXiv:1706.03762

Core Concepts in LLM Inference

Tokenization

- Converts raw input text into discrete units (tokens) using subword or byte-pair encoding (BPE).
- Example: “transformers are cool” → ["transform", "ers", "are", "cool"].

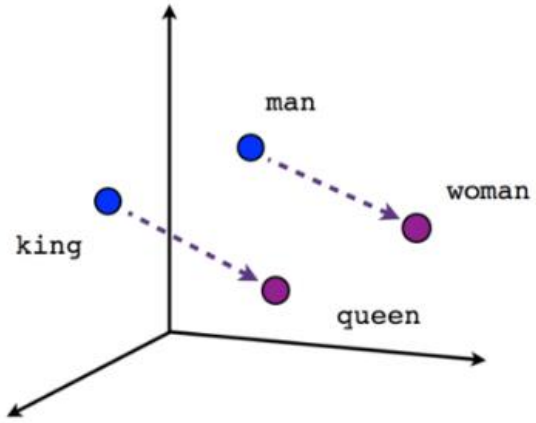


Auto regressive decoding

Embeddings: Embeddings are dense vector representations of text that capture semantic meaning, enabling efficient similarity search and comparison in high-dimensional space.

Autoregressive Decoding

- Generates text one token at a time.
- Each new token is predicted based on all previously generated tokens.
- Common in decoder-only models like GPT and LLaMA.



Embeddings visualization

Image credits: embeddings visualization, auto-regressive decoding visualization

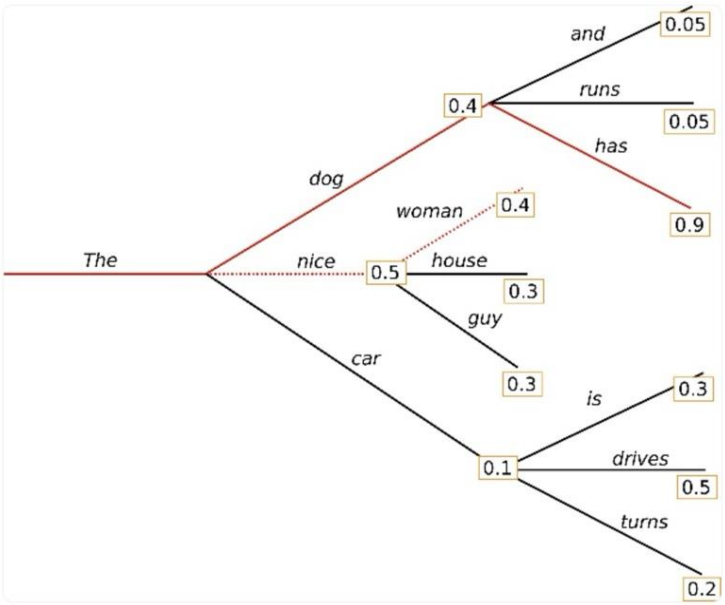
Core Concepts in LLM Inference

Sampling

- The LLM outputs a probability distribution
- A sampler samples an output token from the distribution

KV Cache (Key-Value Cache)

- Caches attention keys and values for previously seen tokens.
- Prevents re-computation during each decoding step, significantly boosting speed and reducing memory usage.

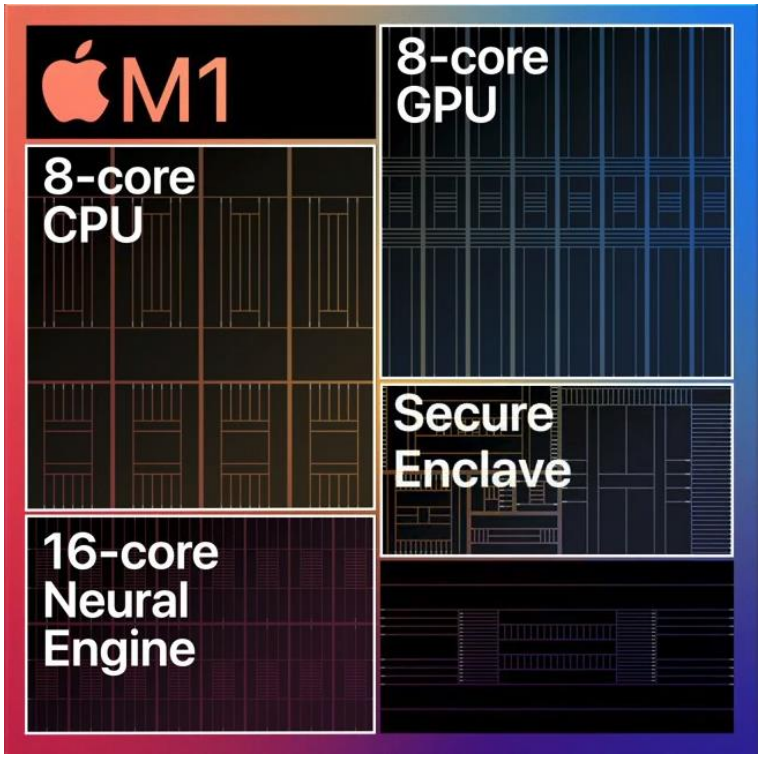


Sampling via Beam Search

Image credits: https://heidloff.net/article/greedy-beam-sampling/

Apple Neural Engine

- **Apple Neural Engine (ANE)** is a dedicated NPU in Apple silicon designed for efficient, low-power execution of machine learning tasks like matrix multiplications and convolutions.
- While accessible **only through CoreML**, the ANE offers up to **11 TOPS** performance at **335 MHz** and is ideal for offloading parallel ML workloads, freeing CPU/GPU for other tasks *.



* machinelearning.apple.com/research/neural-engine-transformers

Tinygrad Project

- **tinygrad** is a minimalist machine learning framework often used for educational and low-level system research
- Made reverse engineering efforts to unlock low-level access to Apple's Neural Engine (ANE), which is typically restricted via CoreML APIs only.
- It shed light on Apple's proprietary hwx file format and the execution pipeline used to dispatch ML workloads to the ANE.
- This project derives inspiration from tinygrad's unconventional use of CoreML

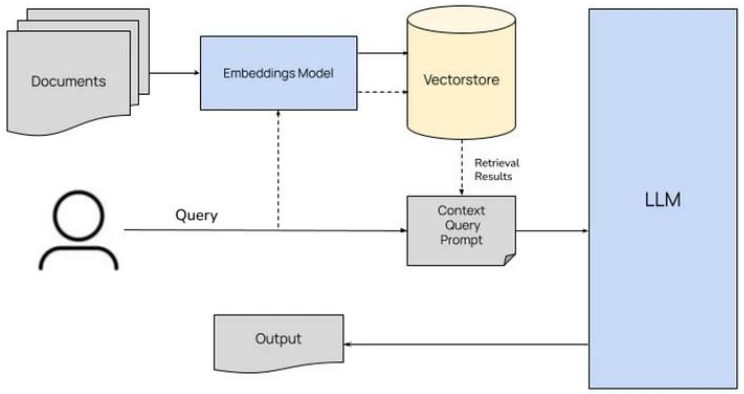
* github.com/tinygrad/tinygrad

Retrieval Augmented Generation (RAG)

What is RAG?

It is a hybrid approach that combines retrieval of relevant documents with language generation to produce more grounded, factual, and context-aware responses.

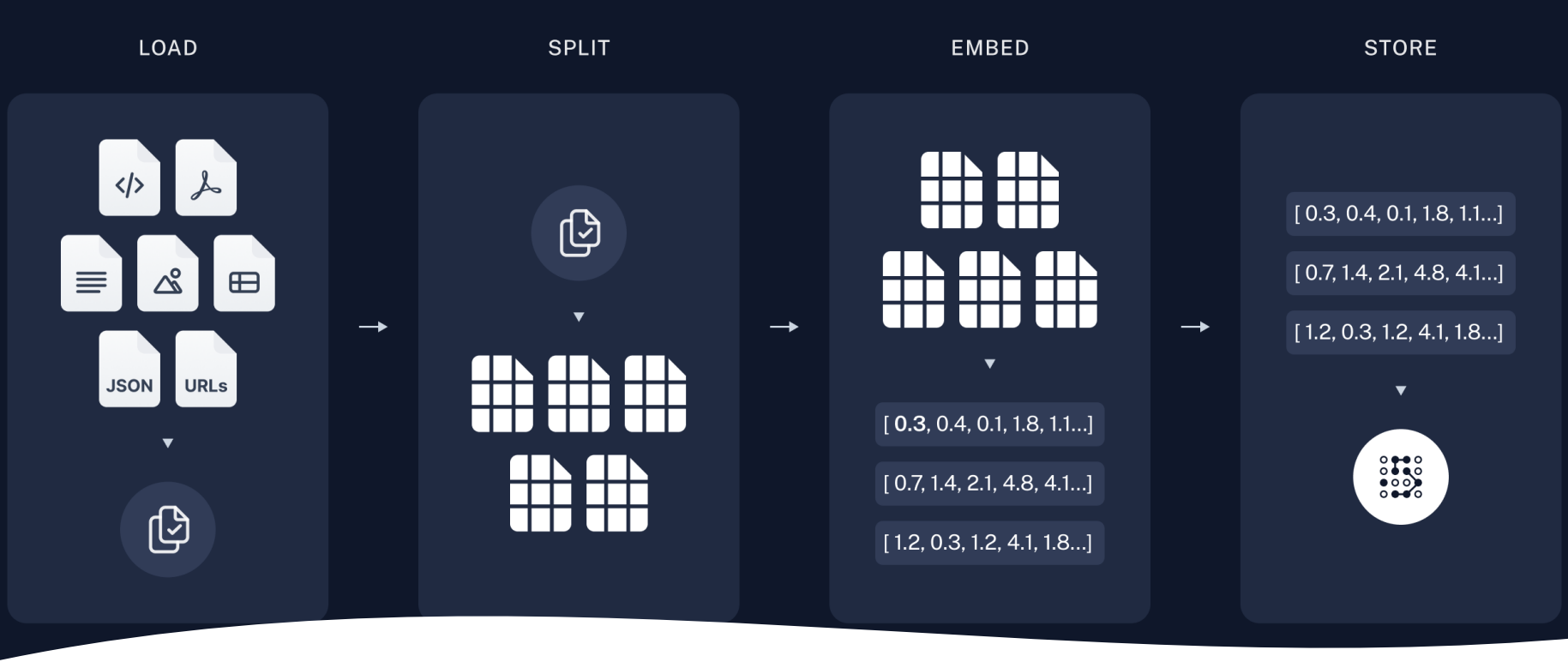
- **Embedding:** Index the source documents by converting them to embedding vectors
- **Retrieval:** Searches a knowledge base or document corpus for passages relevant to the input query.
- **Generation:** Uses a language model (e.g., LLaMA, GPT) to generate a response conditioned on the retrieved content.



Why Use RAG?

- Helps the language model answer based on actual documents, improving credibility, reducing hallucinations, and enabling source attribution.

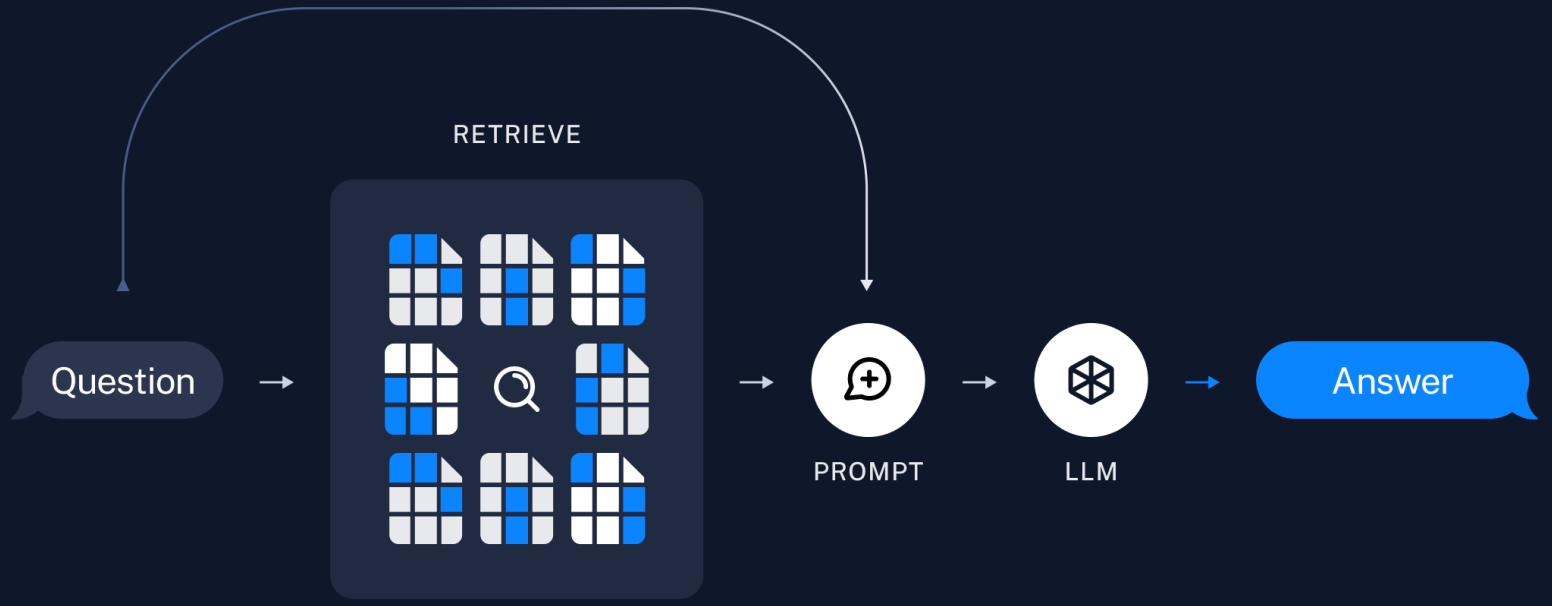
* blog.stackademic.com/mastering-rag



Embedding Phase

- **Document Chunking:** Large documents are split into smaller, manageable text chunks using heuristics like length, sentence boundaries, or semantic structure to fit within the LLM's context window.
- **Text Embedding:** Each chunk is converted into a dense vector using a pre-trained embedding model (e.g., MiniLM or BERT), capturing the semantic meaning of the content.
- **Storage:** The resulting embeddings are stored in a vector database or as binary vectordump files, alongside metadata like chunk text, document ID, and page number for efficient retrieval during inference.

* rag-langchain



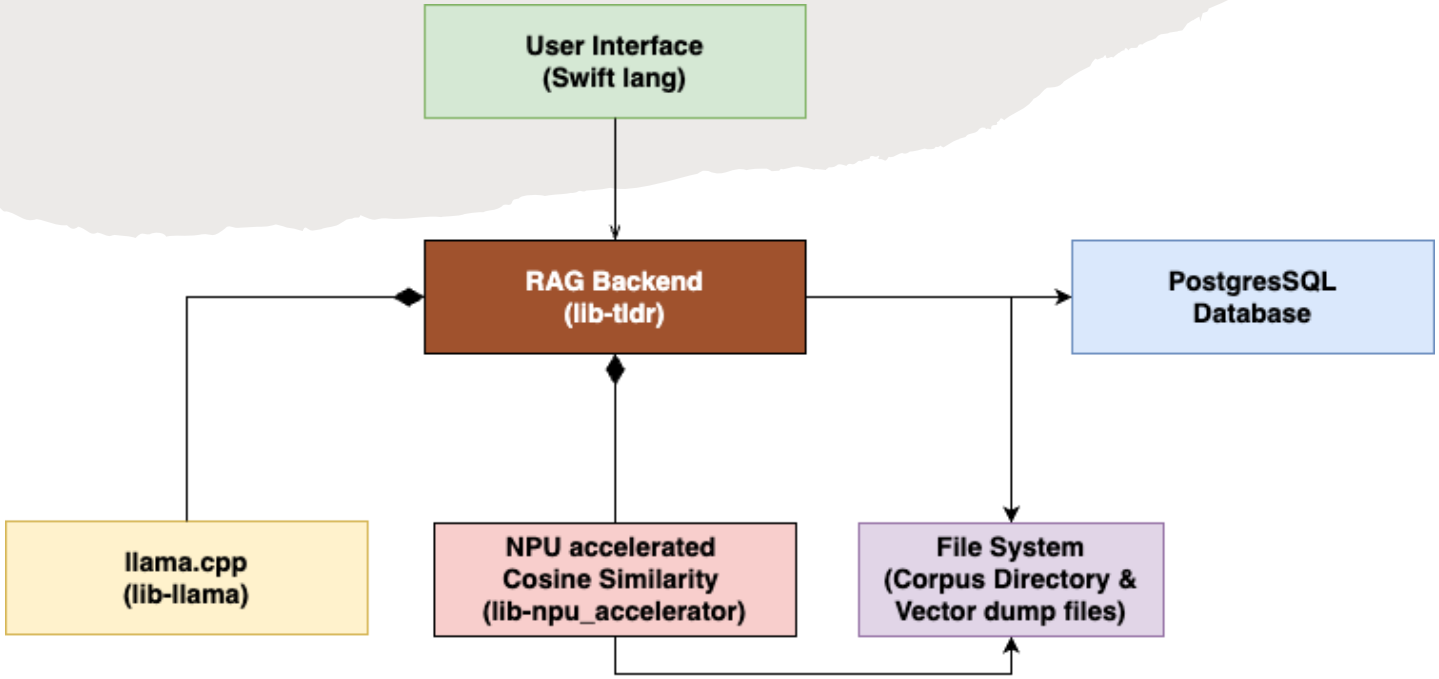
Retrieval & Generation Phase

- **Query Embedding:** The user's query is embedded using the same embedding model used during the indexing phase to ensure consistency.
- **Similarity Search:** The embedded query vector is compared against the stored document embeddings using a similarity metric (e.g., cosine similarity) to retrieve the most relevant chunks.
- **Context Construction:** The retrieved text chunks are compiled into a context that is passed, along with the original query, making a combined prompt.
- **Response Generation:** A Decoder-only language model (e.g., LLaMA) generates the final answer based on the query and retrieved context, producing grounded and relevant responses.

* rag-langchain

Implementation

TLDR Application Modules



- User Interface: A Graphical User Interface application
- RAG Backend: Static library to implement the RAG pipeline and integrate all other modules

- NPU Accelerator: Static library to perform Cosine similarity search on embeddings using ANE
- Llama.cpp: Load and execute LLMs for chat and embedding

- File system: Store source files and embedded vector dumps
- Database (PostgreSQL): Store document metadata, embedding hashes and text chunks

User Interface (MacOS Application)

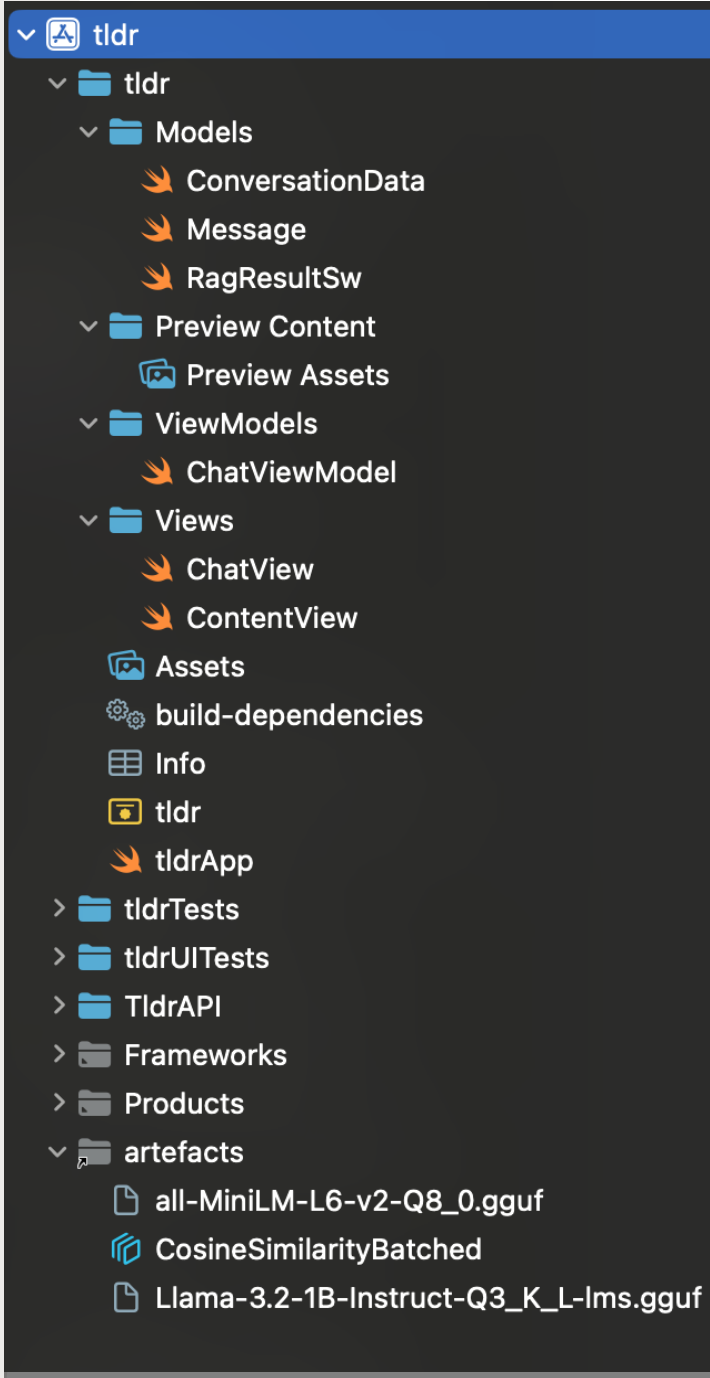
Responsibilities:

- Deliver a smooth and seamless user experience
- Provide a conversational chat experience
- Store and manage conversation data and other user preferences

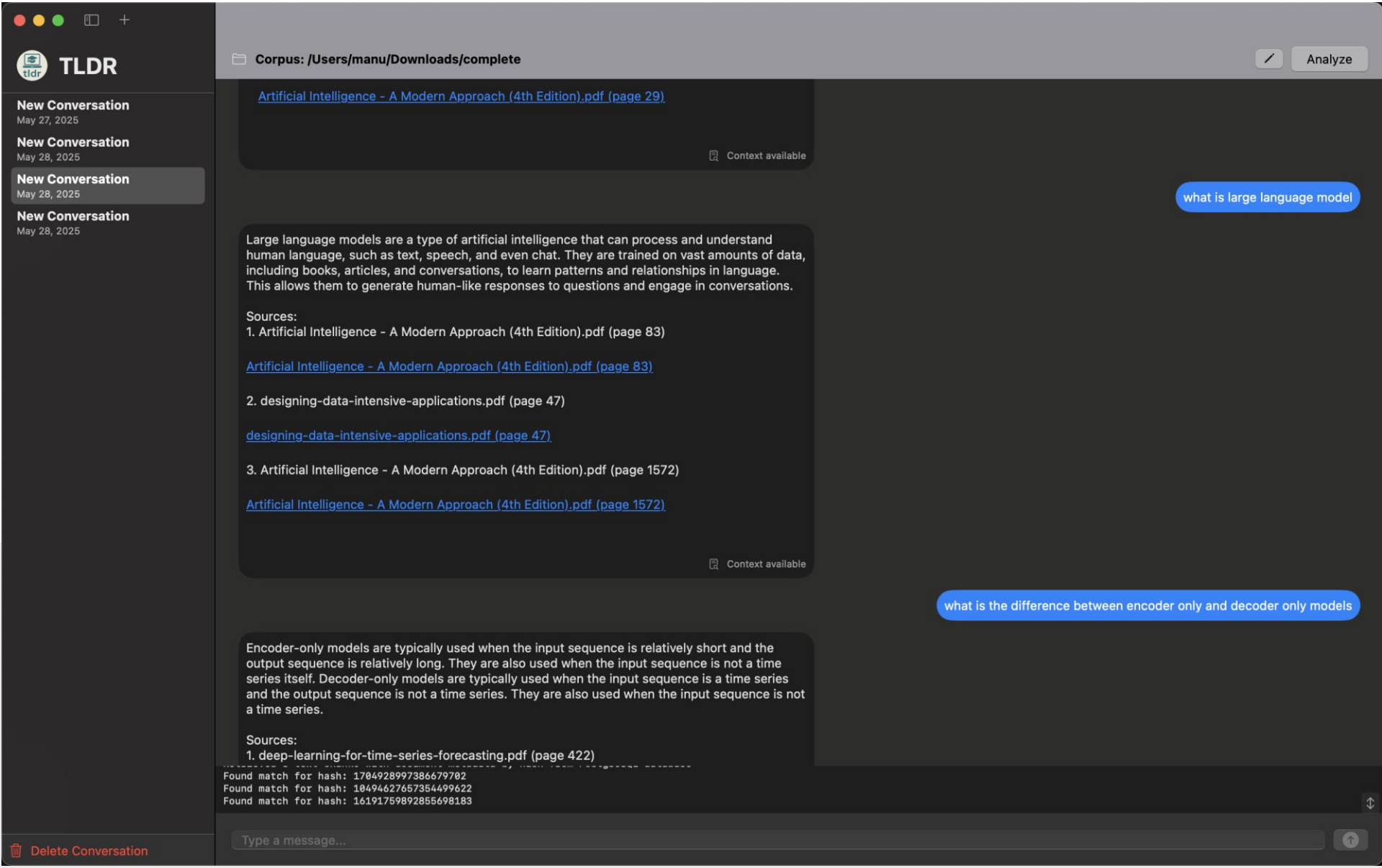
- Interface with the C++ RAG backend and allow it to focus solely on the core logic of RAG

- Handle the nuances of running an application in the MacOS environment (sandboxing, permissions, etc.)
- Create a single self-containing package that encompasses all required assets including LLM weights

TLDR on MacOS Dock

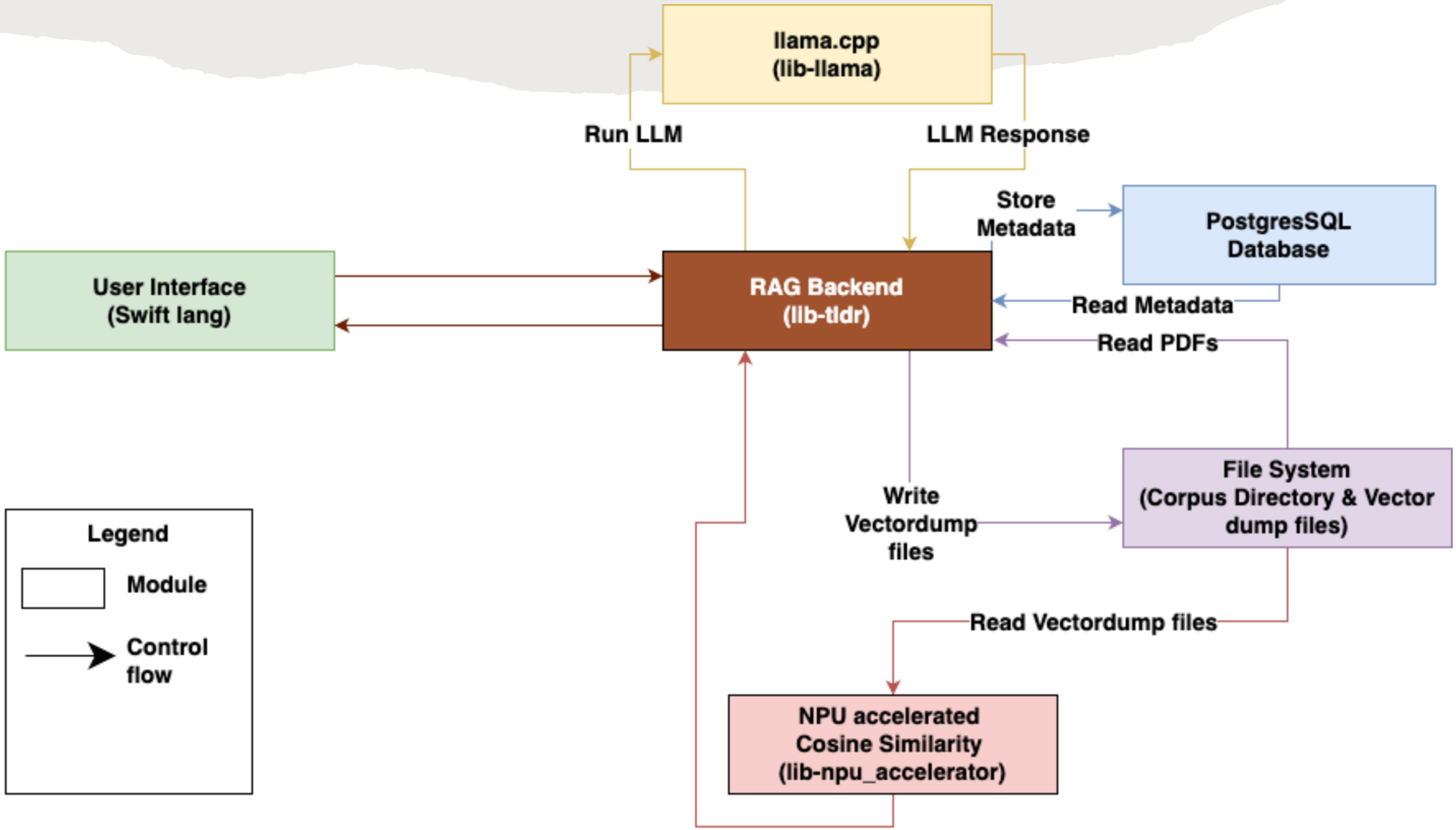


UI codebase structure



User Interface – A Sneak Peek

The RAG Backend – Module interactions



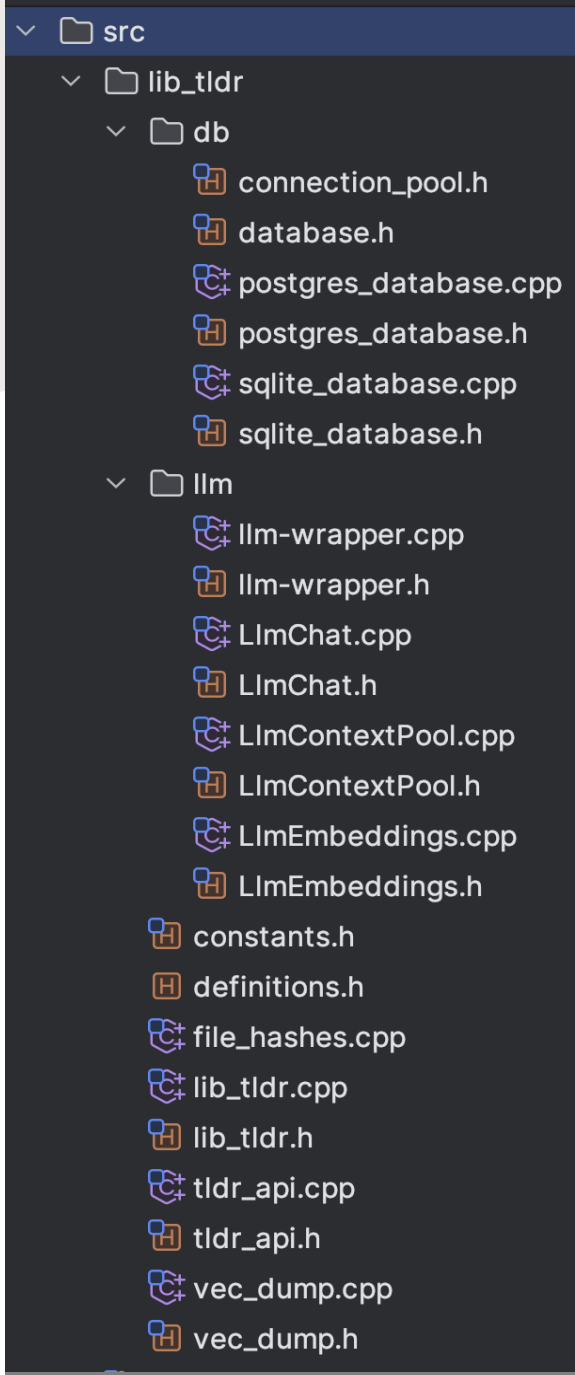
The RAG Backend
(C++ Static Library – lib_tldr)

Responsibilities:

- Initialize and manage Database and resources like DB connection pool and LLM context pool

- Integrate with llama.cpp to load and execute Chat and Embedding LLMs
- Leverage Vectordump module to write embeddings to the file system
- Leverage NPU Accelerator module to read and search Vectordump files for relevant vectors

- Implement all the necessary RAG logic like document parsing, chunking, efficient multi-threaded processing.



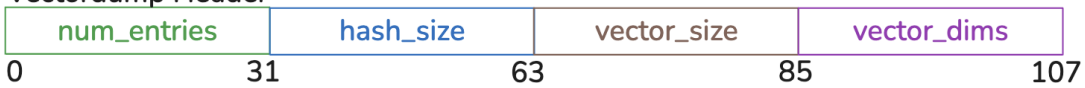
lib_tldr codebase

Vectordump files

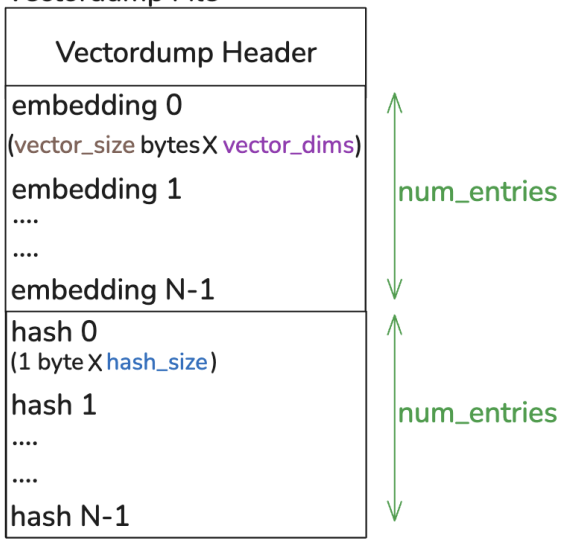
- Store embeddings for text chunks obtained during the embedding process.
- Created using VecdumpWriter in RAG Backend
- Read by the NPU accelerator module by 'mmap'ing the files into memory to perform vector similarity search
- Optimized header and file structure for direct use after loading into memory

```
struct VectorDumpHeader {
    uint32_t num_entries;          // Number of embedding vectors/hashes
    uint32_t hash_size_bytes;      // Size of each hash in bytes
    uint32_t vector_size_bytes;    // Size of each embedding vector in bytes
    uint32_t vector_dimensions;  // Number of dimensions in each vector
};
```

Vectordump Header



Vectordump File



NPU Accelerator (lib-npu_accelerator)

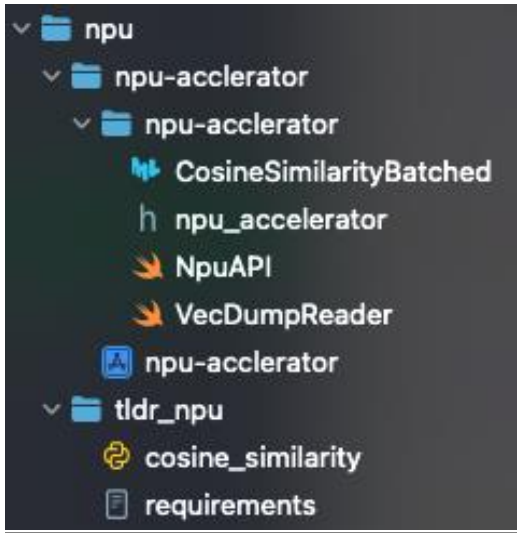
Leverages Apple's Neural Processing Unit to perform hardware-accelerated cosine similarity computations as part of the RAG pipeline

PyTorch module (tldr npu): Contains PyTorch code to perform batched cosine similarity.
(used for generating a CoreML package that gets utilized by the NPU accelerator)

Swift module (npu-accelerator): Implements the logic in Swift to

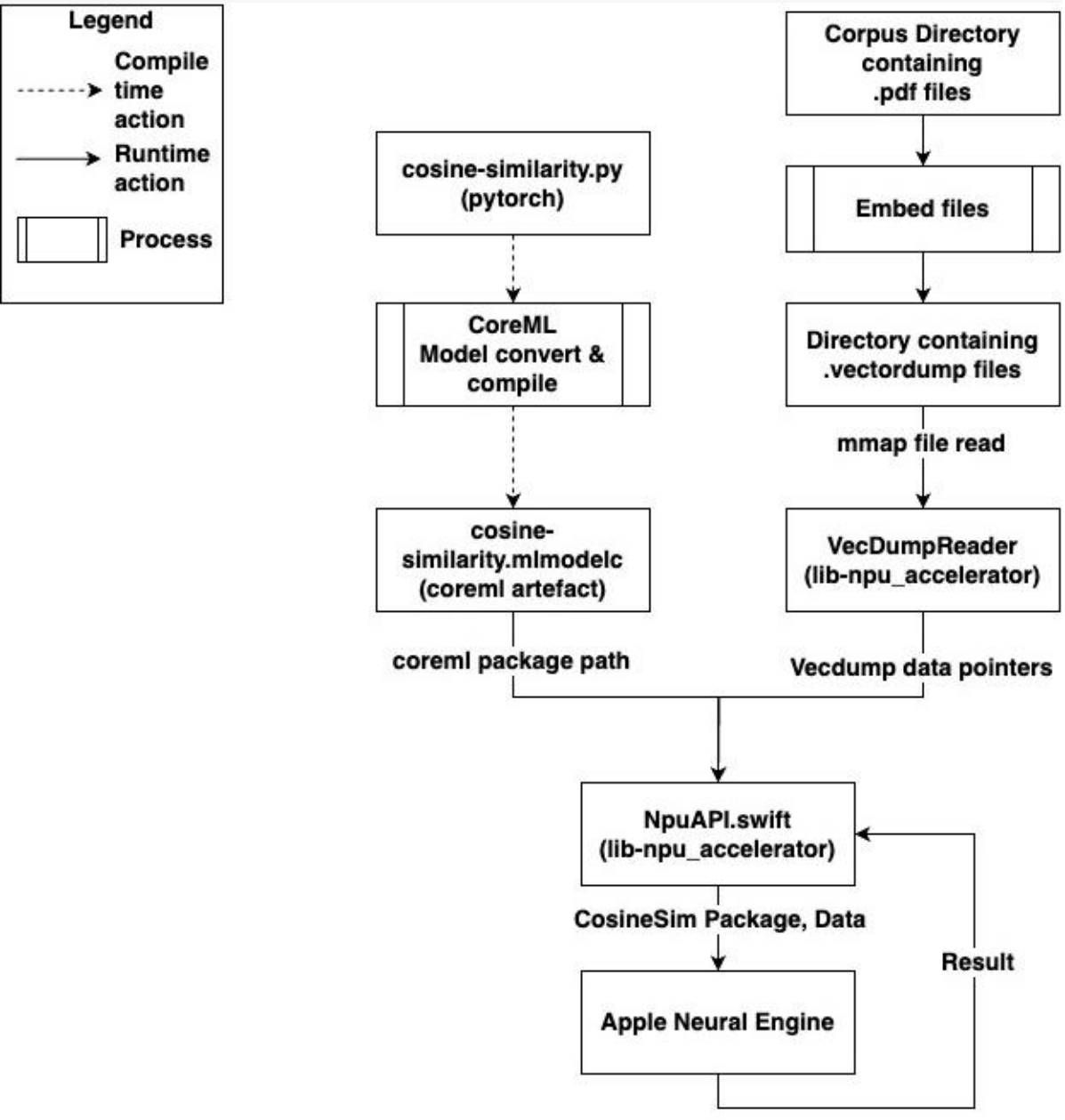
- Read vector dump files
- Leverage the CoreML cosine similarity model to perform vector similarity search using the Apple Neural Engine (ANE)

- Expose the Swift codebase as C++ API to be leveraged by RAG backend (CoreML API available in Python, Swift only)



Npu Accelerator codebase

NPU accelerator workflow



Preparation phase:

- Prepare CoreML package – compile time
- Embed files and create vectordump file – by RAG backend

Execution Phase:

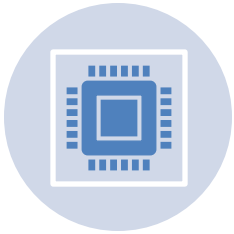
- Receive query vector from RAG backend
- Read vectordump files
- Perform cosine similarity directly on the data accessed by mmap()
- Return embedding hashes and similarity scores to RAG Backend

NPU Module Benefits

Shared memory-mapped files: When multiple threads handle different user requests, redundant file reads are avoided because the mmap

Zero-copy access: Data is accessed directly from memory without duplication.

Operating system can page the data out to swap memory and page it back in when needed, optimizing for repeated periodic reads.



Unified memory architecture: Seamless access between CPU, GPU and NPU, eliminating the need for data transfers during vector similarity computations.

Llama.cpp

- **Customized llama.cpp Fork:** A streamlined macOS build of llama.cpp using the GGML Metal backend is statically linked with the RAG backend for in-memory inference.

- **How llama.cpp is used:**
  - Functionalities such as tokenization, decoding, and sampling are accessed through the public APIs exposed in llama.h and ggml.h.
  - LLmChat and LLmEmbedding classes from RAG backend inspired from workflows in server, embedding, & simple submodules

- **Performance Optimizations:**
  - OpenMP for parallelized tokenization and batch decoding.
  - LLM context pools for multiple parallel executions using shared model weight.

- **No External Dependencies:** Enables fully offline, dependency-free inference without requiring any background services or external tools.

LLMs Used

Embedding LLM:

- *Model:* all-MiniLM-L6-v2-Q8_0
- **It is an encoder-only** language model, follows a variation of the BERT architecture
- Produces 384-dimensional embeddings
- Uses 8-bit symmetric quantization (optimized for speed)

Chat LLM:

- *Model: Llama-3.2-1B-Instruct*
- It is a **decoder-only** language model based on the LLaMA architecture.
- Uses Q3_K_L quantization (blockwise, asymmetric).
- Not explicitly finetuned for multi-turn chat

	column_name name	data_type character varying
1	id	uuid
2	page_count	integer
3	created_at	timestamp with time zone
4	updated_at	timestamp with time zone
5	title	text
6	author	text
7	subject	text
8	keywords	text
9	creator	text
10	producer	text
11	file_hash	text
12	file_path	text
13	file_name	text

Documents table

	column_name name	data_type character varying
1	created_at	timestamp with time zone
2	page_number	integer
3	document_id	uuid
4	id	bigint
5	embedding	USER-DEFINED
6	embedding_hash	text
7	chunk_text	text

Embeddings table

Database (PostgreSQL)

- Stores document metadata, chunked text, embedding values, hashes and chunk metadata
- PostgreSQL chosen over SQLite to enable lock-less access and concurrent writes

Overall workflow

The workflow of the modules of the system and the RAG pipeline can be divided into 3 logical phases.

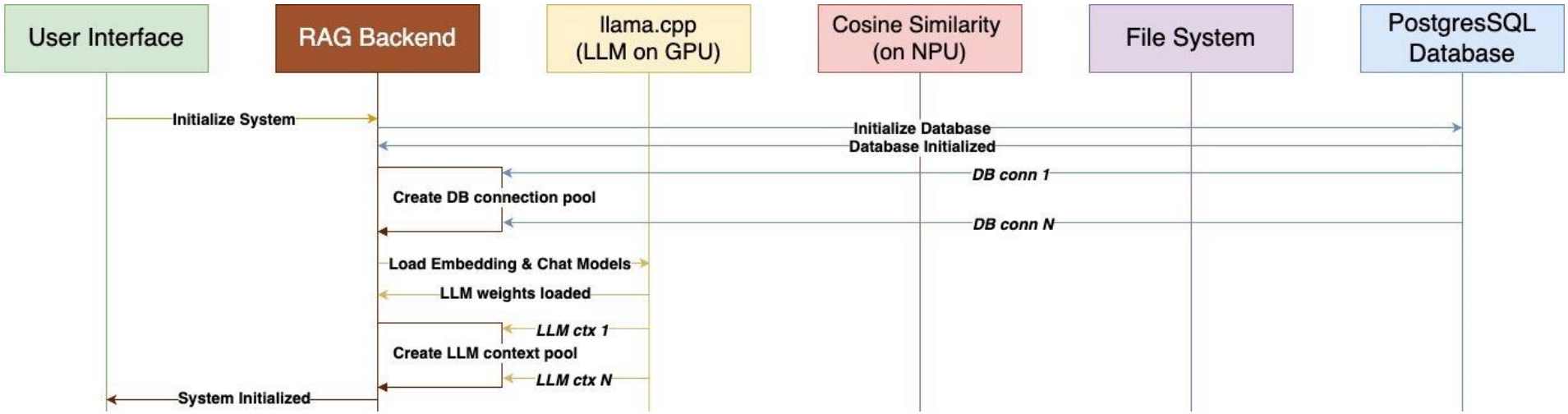
- System Initialization: Initializes the resources required by the system.

- Embedding Phase: Generate embeddings for documents in the source corpus.

- Retrieval and Generation Phase: Leverage embedded documents to retrieve information relevant to a query made by the user and generate a response using the LLM.

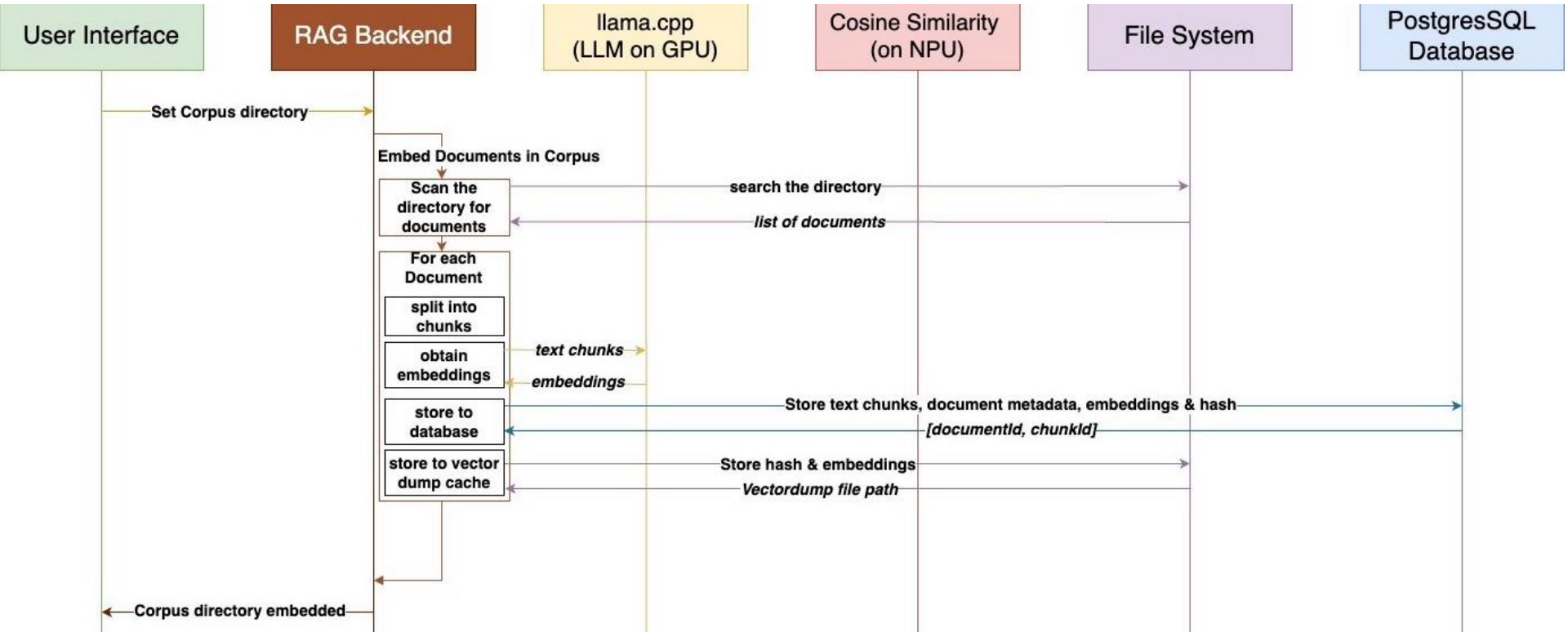
Application Workflow - System Initialization

Initialize the the PostgreSQL Database and its connection pool → Load LLM weights for chat and embedding models → Create Context pools for each model



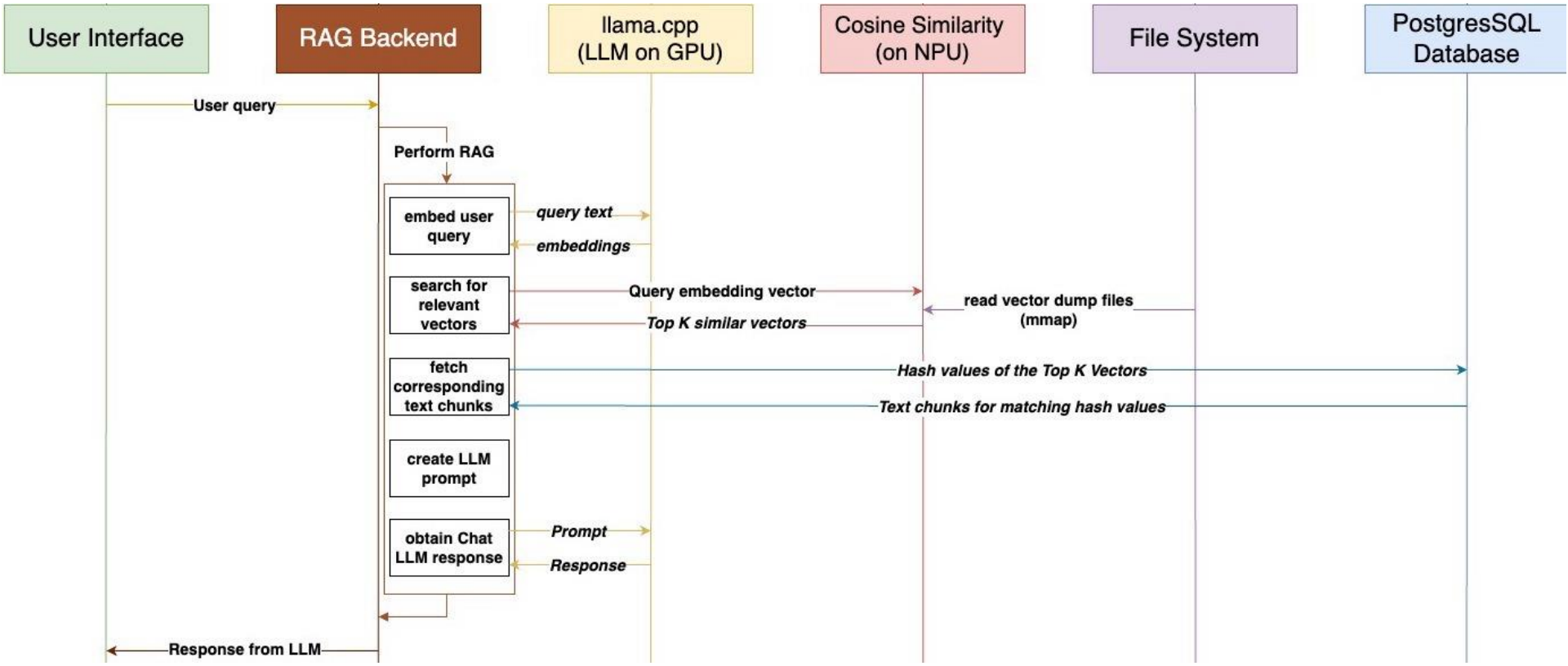
Application Workflow - Index Corpus Directory

Recursively scan Corpus directory for documents(PDFs) → For each document: Load, chunk, and convert to embeddings (using the embedding model) → Store embeddings, text chunks and associated metadata in DB → Write embedding vectors to .vecdump files



Application Workflow - Perform RAG

Embed the user query using the same embedding LLM → Trigger the NPU accelerated Cosine Similarity search on corpus directory → Obtain text chunks corresponding to embedding hashes yielded from search results → Combine chunks into single context/prompt text for the LLM → Invoke the Chat LLM and obtain response



Results

Findings

- The embedding process consumes approximately 90% of the total compute time, making it the most resource-intensive component.
- Optimizing solely for maximum resource utilization is not always effective
- Thermal performance significantly affects system behavior—excessive heat can degrade performance, yet it is often overlooked in theoretical planning.
- While cloud-based models like ChatGPT offer superior capabilities, our solution achieves comparable results, making it a practical approximation within the scope of this project.
- We have a desktop application that can perform retrieval over a directory of documents each of which could be 100s or 1000s of pages and receive responses with references to exact document and its page number (with clickable links)

Demonstration

Please wait for the live demo to be launched

OR

Visit TLDR Demo video - https://www.youtube.com/watch?v=WmP0AhPfIxM

Results

Our Contributions

- Apple Neural Engine (ANE) Utilization: Explores leveraging of the underused NPU for LLM inference, going beyond standard CoreML use cases.
- Retrieval-Augmented Generation (RAG) without Internet : Implements a lightweight RAG pipeline that only uses on device resources.
- Portability and Ease of use: Application size of less than 1GB that includes everything.
- Quantized LLMs: Uses compact models (50–500MB) for efficient, on-device inference allowing for seamless multitasking.

Limitations

- Using only the directly relevant text chunk can paint incomplete picture
- LLM Inference still only leverages the GPU (support NPU as part of future work)

- Indexing(embedding) takes 90+% of overall runtime
- Many smaller quantized LLM weights available currently (3B or less) are only instruction tuned i.e trained for text completion and not for chat. This can lead to unfavorable responses during chat.

- Excessive usage limits to quick draining of power especially on portable devices
- The breadth of retrieval space is limited not by resources but by the context size of the Chat LLM, since both input and expected output must fit in the LLM context.

Future Work

- Add NPU backend for GGML and llama.cpp
- Quantize and convert fine-tuned chat-optimized LLMs to the GGUF format
- Perform context-aware chunking and retrieve chunks adjacent to the most relevant chunk as well

- Make an optimized decoding and tokenization workflow dedicated for embedding (encoder-only, no kv-cache, etc)
- Enhance data safety mechanisms for vectordump files

- Extend the application support beyond Apple Silicon
- Add NPU and GPU acceleration support for Sqlite and Postgres vector search extensions