Project TLDR: Standalone desktop application for question answering and summarization using resource-efficient LLMs

Manu Hegde

A Capstone project report
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2025

Committee:

Erika Parsons

Michael Stiber

Shane Steinert-Threlkeld

Program Authorized to Offer Degree:
Computer Science and Software Engineering

University of Washington

Abstract

Project TLDR: Standalone desktop application for question answering and summarization using resource-efficient LLMs

Manu Hegde

Chair of the Supervisory Committee:
Erika Parsons
School of Science, Technology, Engineering & Mathematics

This project presents the design and development of a standalone desktop application for offline question answering and summarization over a user-provided document corpus, using resource-efficient large language models (LLMs). Targeted for Apple's M1/M2 hardware, the application leverages on-device computation via the Apple Neural Engine (ANE) and Metal shaders, exploring the use of the NPU beyond traditional CoreML applications. The application addresses key concerns around data privacy, resource efficiency, and accessibility. Unlike cloud-based services that require constant internet access and raise privacy risks, this application offers a secure, local alternative optimized for researchers and students. It features a graphical interface and supports retrieval-augmented generation (RAG) over the user's corpus, all while utilizing only a fraction of system resources to support seamless multitasking. Evaluation is conducted using both functional metrics (e.g., BERTScore against ChatGPT outputs) and non-functional metrics (e.g., memory and CPU usage). The result is a practical, efficient application that enables interaction with large academic corpora while preserving system responsiveness and data confidentiality.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables v

Chapter 1: Introduction 1
1.1 Background and Motivation 1
1.2 Our Contributions 2
1.3 Scope 4
1.4 Paper overview 4

Chapter 2: Theoretical Background 6
2.1 Large Language Models and Inference Components 6
2.2 LLM Weight Quantization 8
2.3 Apple M1 System-on-Chip (SoC) 10
2.4 Retrieval-Augmented Generation (RAG) Pipeline 12
2.4.1 Ingredients of a RAG Application 12
2.4.2 Embedding Phase 13
2.4.3 Retrieval Phase 15
2.4.4 Output Evaluation: 18

Chapter 3: Related Work 19
3.1 RAG Frameworks and Agentic Systems 19
3.2 llama.cpp 20
3.3 Ollama 21
3.4 Llamafile 21
3.5 Tinygrad project and Apple Neural Engine (ANE) 22
3.5.1 Hardware Overview 22

3.5.2 Software and Compilation Stack . . . 23
3.5.3 Instruction Format and Operation Structure . . . 24
3.5.4 Supported Operations and Activations . . . 24
3.5.5 tinygrad Implementation . . . 25
3.5.6 Security and Access . . . 25
3.5.7 ANE Takeaways . . . 25

Chapter 4: Methodology . . . 27
4.1 Application Modules and Design . . . 27
4.1.1 Overview . . . 27
4.1.2 User Interface . . . 29
4.1.3 RAG Backend . . . 32
4.1.4 Database (PostgreSQL) . . . 36
4.1.5 File System: Corpus Directory and Vectordump Files . . . 38
4.1.6 NPU Accelerated Cosine Similarity . . . 41
4.1.7 llama.cpp . . . 45
4.2 Application Design - Workflow Overview . . . 46

Chapter 5: Experimentation-TODO . . . 51
5.1 Training Results . . . 52
5.2 Autoencoder Results . . . 53
5.3 Generator Results . . . 57
5.4 Model Performance . . . 60
5.4.1 Error Measurements . . . 60
5.4.2 Execution time . . . 60

Chapter 6: Conclusion-TODO . . . 62
6.1 Conclusion . . . 62
6.2 Contributions . . . 62
6.2.1 Data preparation and training methods . . . 62
6.2.2 Model arquitecture . . . 63
6.3 Limitations . . . 63
6.4 Future work . . . 64

LIST OF FIGURES

Figure Number		Page
2.1	Transformer architecture [1]	7
2.2	Autoregressive decoding [2]	8
2.3	Apple M1 Architecture - (A12 Bionic) Chip floor plan [3]	10
2.4	High-level overview of a RAG system with its four main components [4]	13
2.5	The embedding phase: document ingestion, chunking, and vectorization [5]	14
2.6	The retrieval phase: querying the vector store and invoking the LLM [5]	16
2.7	Autoregressive decoding [2]	17
2.8	RAG Output evaluation metrics [6]	18
3.1	Apple Neural Engine Workflow	23
4.1	Modules of TLDR application	28
4.2	Graphical User Interface of TLDR Application	30
4.3	TLDR Application Icon as seen in MacOS Dock	30
4.4	UI Project File system	31
4.5	RAG Backend(lib_tldr) codebase	33
4.6	Documents table description	37
4.7	Embeddings table description	38
4.8	Vector dump file structure	40
4.9	lib-npu_accelerator codebase	42
4.10	NPU accelerator module workflow	43
4.11	RAG Output evaluation metrics [6]	46
4.12	RAG Output evaluation metrics [6]	47
4.13	RAG Output evaluation metrics [6]	48
4.14	RAG Output evaluation metrics [6]	49
5.1	Autoencoder and Generator loss evolution during training	52
5.2	Original vs Reconstructed frames	55

5.3 Original vs Reconstructed frames velocity histograms 56
5.4 Original vs Generated frames 58
5.5 Original vs Generated frames velocity histograms 59
5.6 Model MSE error metric 61

LIST OF TABLES

Table Number Page

2.1 Comparison of Common Quantization Formats in `ggml/llama.cpp` 9

5.1 Training and Testing errors (MSE) 53

5.2 CFD simulation vs DL Model execution time 61

ACKNOWLEDGMENTS

I would like to express my gratitude to Prof. Erika Parsons for all the valuable guidance and help during this work. Furthermore, I sincerely thank Prof. Steinert-Threlkeld and Prof. Stiber for accepting my request to be on the committee for this thesis and for providing precise feedback.

Chapter 1

INTRODUCTION

1.1 Background and Motivation

The field of Natural Language Processing (NLP) has undergone a significant transformation with the advent of Large Language Models (LLMs), which are capable of performing complex language understanding and generation tasks. Groundbreaking works such as the Transformer architecture [1], BERT [7], and GPT-family models [8, 9] have paved the way for highly capable models that support applications such as summarization [10], question answering [11], and document understanding [12]. These advances have been further systematized in the concept of foundation models [13], which emphasize the broad applicability and adaptability of pre-trained LLMs.

Despite their success, most widely used LLM applications operate via cloud-based services, which introduce significant limitations when it comes to privacy, data security, and control over computational resources. This is particularly concerning in academic contexts, where students and researchers often deal with sensitive or proprietary content. Recent studies have raised awareness of the risks associated with exposing private data to generative models, including membership inference [14] and data extraction attacks [15]. Moreover, surveys indicate increasing usage of LLMs in research and education, highlighting both the demand for such tools and the concerns around data governance [16, 17].

Simultaneously, the hardware landscape has evolved to enable local deployment of such models. Apple's M1 and M2 chipsets integrate high-performance CPUs, GPUs, and a dedicated Neural Processing Unit (NPU) through the Apple Neural Engine (ANE). These architectures offer a promising platform for efficient, on-device inference of LLMs, provided the models are adapted appropriately to operate under limited memory and compute budgets.

This convergence of high-capability models, growing privacy concerns, and increasingly powerful consumer hardware forms the backdrop for *Project TLDR*—a standalone desktop application for summarization and question answering over a user-specified corpus. The tool is designed to run entirely offline, preserving user privacy while leveraging optimized LLM inference. The project makes use of modern techniques such as quantization [18] and low-rank adaptation (LoRA) [19] to reduce computational overhead and improve deployment feasibility on M1/M2 hardware. Additionally, the use of Retrieval-Augmented Generation (RAG) [20] ensures that answers and summaries are grounded in user-provided text, enhancing both contextual relevance and factual consistency.

In essence, this project is motivated by the goal of empowering academic users with a practical, secure, and efficient means of engaging with large volumes of textual data. By tying together advances in NLP, secure computing practices, and consumer-grade hardware acceleration, Project TLDR aims to demonstrate that high-quality language understanding can be brought directly to the user's device—without compromise.

1.2 Our Contributions

In this project, we present *Project TLDR*, a privacy-preserving, offline, and resource-efficient desktop application that enables users to perform Question Answering (QA) and Summarization over personal document repositories. Designed primarily for MacOS systems powered by Apple's M1 and M2 architectures, the application aims to support academic and research workflows where confidentiality, simplicity, and efficiency are paramount.

Our key contributions are as follows:

- **Novel Utilization of Apple Neural Engine (ANE)**: A significant technical contribution of this project is our investigation into utilizing Apple's underused Neural Processing Unit (ANE), capable of up to 11 TOPS in INT8 precision [21]. While current LLM deployment frameworks such as LLaMA.cpp [22] or Ollama[23] do not harness this co-processor, we demonstrate and document methods to tap into the ANE

for local inference acceleration. We build on the NPU API reverse-engineering work by tinygrad [24] and leverage the learnings to open a new direction for efficient LLM deployment on Apple silicon devices. We hence leverage the NPU outside of traditional CoreML model deployment paradigm and demonstrate how it can be used for various use cases.

- **Retrieval-Augmented Generation (RAG) Architecture**: We implement a lightweight yet effective RAG pipeline [20] for performing QA and summarization tasks over local collections of documents. This enables the application to provide context-grounded, source-aware responses from user-specified corpora while leveraging limited compute resources.

- **Efficient On-Device Inference Using Quantized LLMs**: We leverage quantized transformer models [18], reducing memory and compute demands without compromising output quality. Instead of multi-gigabyte model downloads (as required by tools like Ollama [23] or LLaMA.cpp [22]), we use compact models (50–500MB) that support practical usage scenarios with minimal setup, enhancing portability and usability for non-technical users.

- **User-Friendly and Ready-to-Use Design**: Unlike tools such as Ollama [23] and LLaMAFile[25], which require technical familiarity and understanding the nuances of various models, our application provides a clean graphical interface with ready-to-use capabilities tailored to common academic needs—eliminating the steep learning curve and reducing operational friction.

- **Privacy-Preserving Document Analysis**: By running entirely on-device, our application mitigates the risks associated with uploading sensitive or proprietary documents to third-party services (e.g., ChatGPT, Claude, Gemini), which have raised concerns over data leakage [14, 15]. Users can securely summarize, query, and rephrase

information without network access or cloud APIs.

Through this suite of contributions, *Project TLDR* demonstrates that meaningful and secure LLM-powered applications can be brought directly to end-users without reliance on cloud services or specialized technical knowledge, thereby filling a critical gap in the current LLM applications ecosystem.

1.3 Scope

This project aims to develop a standalone desktop application capable of performing Question Answering (QA) and Summarization over a locally stored corpus of documents using Retrieval-Augmented Generation (RAG). Unlike most current solutions, such as ChatGPT or Gemini, which require users to upload documents each time they wish to query them, this application allows persistent storage and indexing of documents on the user's device. Once added to the local vector store, documents are automatically embedded and made queryable without requiring repeated user intervention.

The application is designed to operate entirely offline, preserving data privacy while minimizing resource consumption. It is optimized for modern Apple Silicon devices (e.g., M1/M2 Macs), with the target of using less than 50% of system resources. By utilizing quantized models ranging between 50MB and 500MB in size and streamlining the entire RAG pipeline—from document ingestion to local inference—this tool ensures meaningful results without the need for large downloads or technical configuration. The scope includes a graphical interface, local embedding and vector database management, and natural language generation capabilities tailored for academic and research use cases.

1.4 Paper overview

This paper is organized as follows. Chapter 2 presents the theoretical foundations relevant to the project, including an overview of Large Language Models (LLMs), key components involved in inference such as the context window and KV cache, and a discussion on the Apple

M1 System-on-Chip (SoC) architecture, with emphasis on the GPU and Neural Processing Unit (NPU). It also covers zero-copy file access techniques, quantization methods, risks of data leakage in LLM usage, and how CoreML leverages the NPU.

Chapter **??** surveys related work, highlighting limitations of existing desktop LLM tools like Ollama and LLaMaFile, prior work on Retrieval-Augmented Generation (RAG), and efforts in reverse engineering NPU APIs by the tinygrad project. Chapter 4 details our methodology and low-level design, including vector dump and memory-mapped reads, interfacing with the NPU, and the internal structure of the RAG pipeline. Furthermore, it describes our software architecture and implementation details, covering our use of SwiftUI, static library linkage, the design of the LLM context pool, and thread management strategies.

Chapter 5 presents our experimentation process and evaluation results using BERTScore across different models and quantization levels. Finally, Chapter 6 concludes the paper, discusses limitations, and outlines future work including the development of an NPU backend for llama.cpp.

Chapter 2

THEORETICAL BACKGROUND

This chapter provides the theoretical foundation for the key concepts relevant to the design and implementation of this project. It focuses on topics related to Large Language Models (LLMs) and the architectural characteristics of Apple Silicon (M1/M2). The sections that follow explore essential components of LLM inference, the unique hardware features of Apple Silicon, and optimizations leveraged to enable efficient on-device performance.

2.1 Large Language Models and Inference Components

Large Language Models (LLMs) are transformer-based neural networks trained on massive text corpora to generate text in human language. They can mimic human-like conversations, storytelling, and can respond to abstract instructions. These models, such as GPT, LLaMA, and Falcon, rely on the transformer architecture introduced by Vaswani et al. [1], where self-attention mechanisms enable the model to capture dependencies across different parts of the input sequence (as seen in Figure 2.1). Inference in LLMs involves several critical components, each contributing to performance, quality, and resource efficiency.

At the core of LLM inference lies the **context window**, which denotes the maximum number of tokens the model can attend to at any given time. A token is a piece of input, ranging from a subword to even a phrase, depending on the tokenization scheme of the model. For models like LLaMA-2, the context window can be up to 4,096 or 8,192 tokens [26]. During inference, the model builds an internal representation of this input context, which is used to generate predictions for the next token.

The **Key-Value (KV) cache** is a performance optimization central to LLM performance. During autoregressive decoding for text generation, a Large Language Model (LLM)

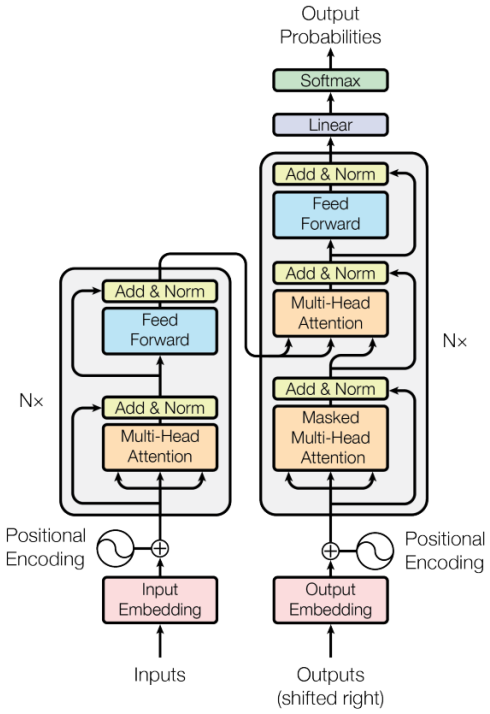


Figure 2.1: Transformer architecture [1]

processes the entire input sequence to generate a single output token. This newly generated token is then appended to the sequence, and the model repeats the process iteratively (as seen in Figure 4.14). However, this leads to redundant computation over previously processed tokens at each step. To mitigate this inefficiency, the *key* and *value* vectors computed during the attention mechanism of the transformer can be cached. This *KV caching* technique significantly reduces repeated computation by reusing the stored attention states from prior steps, thereby improving inference speed and memory efficiency. These cached representations allow the model to efficiently attend to all previously generated tokens without recalculating the entire attention graph [27]. This caching mechanism reduces computational overhead and is especially vital when running LLMs on resource-constrained devices.

Furthermore, the token generation is governed by a sampling algorithm, which yields the output token by sampling on the output probability distribution obtained from the model.

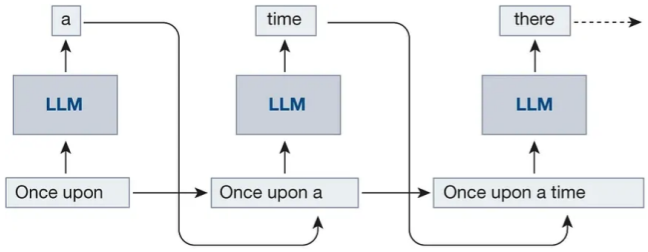


Figure 2.2: Autoregressive decoding [2]

Common strategies include greedy sampling, top-k sampling, and temperature scaling. A **token sampler** module is responsible for leveraging these strategies to generate the output token. This can be a resource-hungry step, since this involves processing the probability distribution over the entire vocabulary of the model.

Therefore, the core components required for consistent text generation in an LLM-based system include the model weights, the LLM context (comprising the KV cache and vocabulary), and the token sampler.

Finally, the inference engine managing the model (e.g., llama.cpp, Hugging Face Transformers, or Apple CoreML) orchestrates the context construction, efficient memory handling of the KV cache, and optimized computation for each transformer layer, accounting for quantization if any. These components together define the responsiveness, accuracy, and resource efficiency of the LLM in real-time applications.

2.2 LLM Weight Quantization

Quantization is a model weight compression technique that reduces the precision of the numbers used for representing model parameters. Typically, reducing the precision from 32-bit floating point to lower-bit integers such as 8-bit, 4-bit, or even 3-bit values. This significantly decreases memory usage and computational requirements, making it possible to

run large language models (LLMs) efficiently on edge devices or in real-time environments without sacrificing too much accuracy [18, 28].

Within the `ggml` framework and its application in `llama.cpp` [22], various quantization schemes have been introduced to significantly reduce the size and memory footprint of LLM weights. One such scheme is `Q3_K_L`, a 3-bit quantization format specifically designed to balance compression efficiency and model performance [29, 30]. In `Q3_K_L`, weights are grouped in sets of 256 and quantized with shared scaling factors, offset by a zero-point, enabling fine-grained approximation while preserving hardware efficiency. Notably, `Q3_K_L` makes use of 4-bit storage for each quantized value (3 bits for the quantized magnitude and 1 extra bit for improved alignment and bit-packing efficiency), along with 8-bit scales and 6-bit zero points per group. This layout ensures better alignment with SIMD instructions and allows for faster matrix-vector multiplications, which are critical during transformer inference [31]. 2.1 shows the comparison between common quantization schemes used in GGML. While this format slightly increases decoding complexity compared to simpler formats like `Q4_0`, it provides a favorable tradeoff between model accuracy and size, especially for models deployed in edge or offline settings [23, 25]. Therefore, *this project primarily employs model weights quantized using the `Q3_K_L` scheme for the Chat LLM.*

Table 2.1: Comparison of Common Quantization Formats in `ggml/llama.cpp`

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

2.3 Apple M1 System-on-Chip (SoC)

The Apple M1 chip, introduced in 2020, is a System-on-Chip (SoC) built on the ARM architecture, which integrates the CPU, GPU, and Neural Processing Unit (NPU) on a single die [32]. This architectural design provides several advantages that are particularly relevant for local inference with large language models (LLMs), particularly for tasks like summarization and question answering.

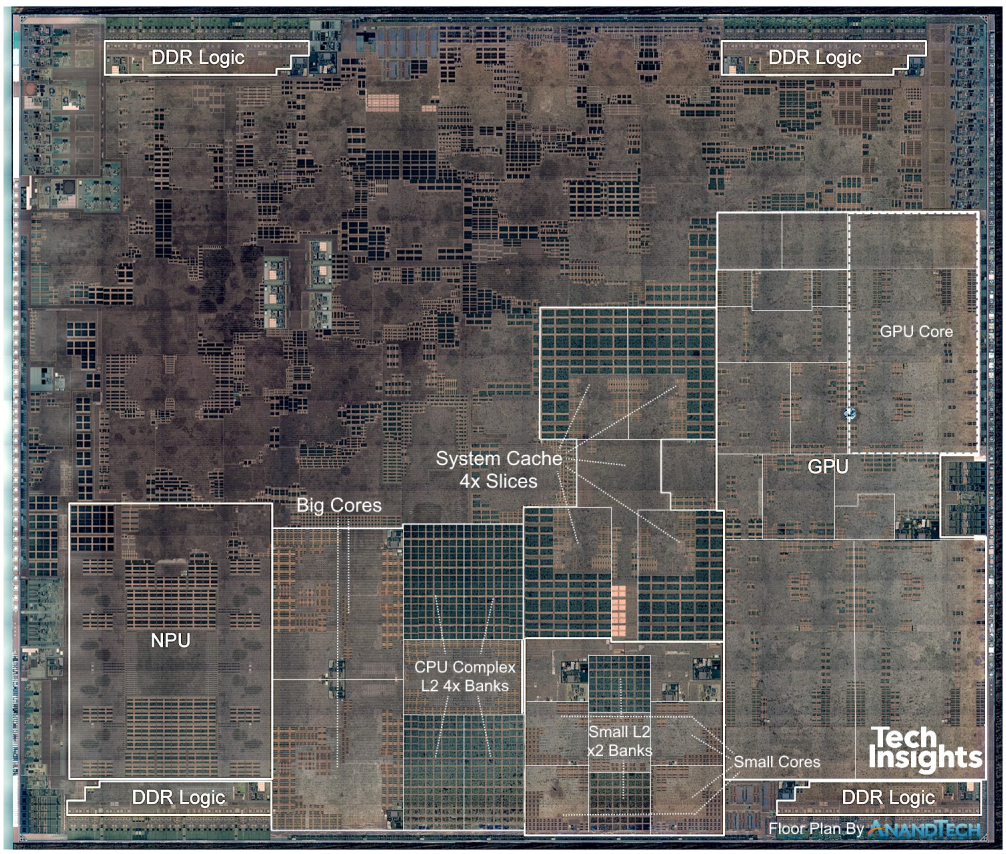


Figure 2.3: Apple M1 Architecture - (A12 Bionic) Chip floor plan [3]

Unified Memory Architecture

One of the defining features of the M1 SoC is its Unified Memory Architecture (UMA), which allows the CPU, GPU, and NPU to share the same memory pool [3]. This eliminates the need for memory copying across processing units and enables zero-copy execution. As a result, applications like retrieval-augmented generation (RAG) benefit from significantly reduced memory overhead and latency. Shared virtual addressing also simplifies software development by allowing seamless access to data structures across different compute units.

On-chip Accelerators: GPU and NPU

The M1 chip includes a built-in GPU with 7–8 cores, capable of delivering up to 5.2 TOPS of performance in INT8 precision [33]. It supports general-purpose computation via Metal Shading Language (MSL), analogous to NVIDIA's CUDA, with added features like managed thread indexing and access to 7+ GB of RAM.

The Apple Neural Engine (ANE), or NPU, is an application-specific integrated circuit (ASIC) designed for efficient neural network operations. It offers 11 TOPS of INT8 compute and is exclusively used for machine learning tasks. However, it is only accessible via Apple's CoreML framework, which supports Swift and Python interfaces. The NPU dynamically collaborates with the CPU, handling SIMD operations like matrix multiplication while delegating sequential logic to the CPU [34].

Unlike the GPU, the NPU is rarely used by general-purpose applications. This makes it a promising candidate for dedicated LLM inference workloads, as its usage is unlikely to interfere with the system's overall responsiveness.

Implications for Local LLM Inference

The M1's integrated architecture is highly beneficial for deploying LLMs locally. Unified memory minimizes data movement, while the GPU and NPU offer parallel computation for matrix-heavy operations common in transformers. Although the NPU cannot be directly

accessed via C++ (barring experimental reverse engineering efforts [34]), its performance can be leveraged through CoreML model conversion pipelines [35].

In the context of this project, these capabilities enable a lightweight, resource-efficient desktop application that performs real-time summarization and question answering over a user-provided corpus without relying on cloud resources.

2.4 Retrieval-Augmented Generation (RAG) Pipeline

Retrieval-Augmented Generation (RAG) is an architectural framework designed to enhance the capabilities of language models by integrating external knowledge retrieval into the generation process [20]. Unlike standalone LLMs that rely solely on pre-trained knowledge, RAG introduces dynamic document retrieval as part of the inference pipeline, thereby improving factual accuracy, contextual relevance, and grounding.

The RAG pipeline can be decomposed into modular phases, each responsible for a specific task in the information flow. The following subsections describe these phases and their associated components.

2.4.1 Ingredients of a RAG Application

A Retrieval-Augmented Generation (RAG) system typically comprises four key components: **Document loader**, **Text embedder**, **Context retriever**, and **Response generator**. Both the Text embedder and the Response generator consist of a Language Model. As illustrated in Figure 2.4, documents are chunked, embedded, and stored in a vector database. This database is later queried to retrieve relevant context for a given user query, which is then passed along to the language model. This enables the model to generate responses grounded in a specific knowledge source, thereby enhancing the factual accuracy and credibility of the output.

The RAG pipeline can be conceptually divided into two major phases: **Embedding** and **Retrieval**.

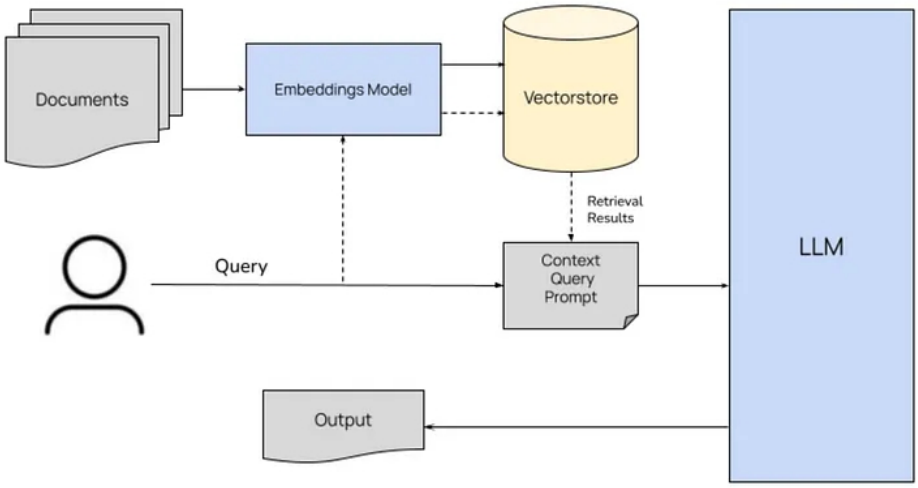


Figure 2.4: High-level overview of a RAG system with its four main components [4]

2.4.2 Embedding Phase

The embedding phase forms the foundation of a Retrieval-Augmented Generation (RAG) system by building the knowledge corpus that the system will later reference to answer user queries (Figure 2.5). While the concept of text embeddings has existed since the introduction of Word2Vec in 2013 [36], their application for contextual storage and retrieval became prominent with the advent of *in-context learning*, as introduced in the GPT-3 paper [8]. This technique leverages the autoregressive nature of decoder-only transformers to generate relevant outputs based on few or even a single example [37].

The embedding phase comprises several key steps:

a. **Document Processing:** Documents in formats such as TXT, PDF, or DOCX are loaded. This may be done in text-only or multi-modal modes, the latter preserving diagrams and figures. This task is handled by the Document Loader module.

   Currently, this project focuses exclusively on PDF files, as most books and academic papers are commonly distributed in this format.

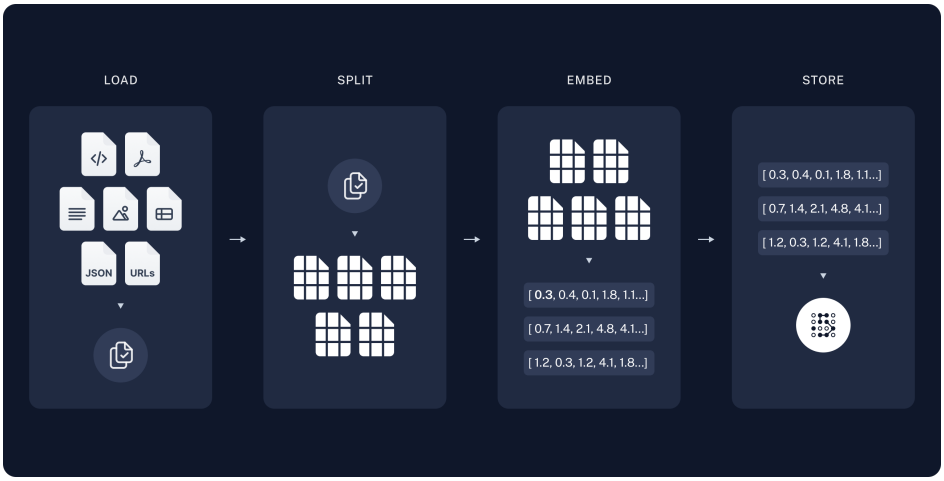


Figure 2.5: The embedding phase: document ingestion, chunking, and vectorization [5]

b. **Chunking:** The loaded documents are split into smaller, semantically coherent segments using text splitters [38]. Chunking is necessitated by the limited context window of LLMs, which ranges from 2K tokens in GPT-3 to 128K tokens in LLaMA 3 [39]. Moreover, using large contexts also demands significant GPU memory, especially when optimizations like KV caching and speculative decoding are employed.

   The size of text chunk has a significant impact on the RAG performance due to the following reasons:

   - *Chunks too large* may result in the "Lost in the Middle" effect [40], where only the beginning and end of the chunk influence the output.
   - *Chunks too small* lead to redundancy and latency, as overlapping content is required to preserve context and sense of continuity.

   *This project leverages **MiniLM**[41] which has a context size of 512 tokens.* MiniLM is known for its effectiveness in knowledge distillation[42], to generate sentence-level embeddings. These embeddings capture the overall meaning of a text segment and serve as semantic representations, enabling efficient retrieval of relevant context.

c. **Chunking Techniques:**

- **Length-based chunking:** Divides text based on fixed token or character length. Tokens are determined using subword tokenization techniques like byte-pair encoding [43].
- **Semantic chunking:** Splits based on document structure, e.g., paragraphs.
- **Context-aware splitting:** Ensures that subtopics are not broken across chunks.

This project creates fixed size text chunks of 500 characters with overlap of 20 characters on each end. These parameters were chosen to match the context size of the LLM and also based on common practices and existing research results [44].

d. **Text Embedding:** In this step, each chunk is converted into a dense vector using a text embedding model. The embedding model used here does not need to match the LLM used in generation, as the embeddings are utilized solely for vector similarity search. Once relevant chunks are retrieved, their original textual content is passed to the LLM, not the embeddings themselves. Compact and efficient models like BERT or DistilBERT are often employed for this task due to their relatively small embedding sizes (e.g., 512 or 768 dimensions), which makes them computationally lightweight compared to models like LLaMA [26] or GPT-3 [8], which produce larger embeddings in the range of 1024 to 12,288 dimensions. Although these smaller embeddings may be less precise, they are generally sufficient for high-level semantic retrieval.

2.4.3 Retrieval Phase

The retrieval phase is initiated whenever a user submits a query or task. This phase makes use of the vector database constructed during the embedding phase to locate and extract relevant content for the given input prompt, which is then passed to the language model for response generation (Figure 2.6).

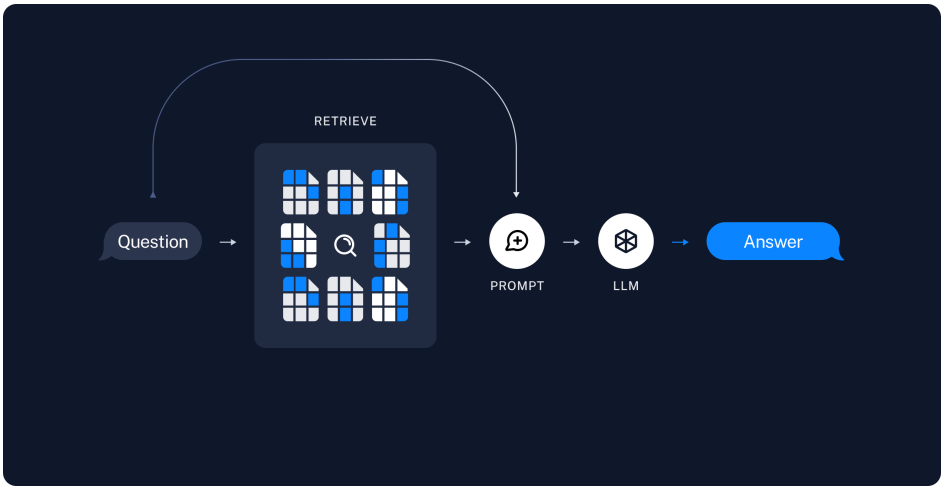


Figure 2.6: The retrieval phase: querying the vector store and invoking the LLM [5]

a. **Context Retrieval:** The context retriever module follows a two-step process:

   (a) Embedding the user's query using the same embedding model employed during the embedding phase.

   (b) Performing a similarity search in the vector database to identify top-matching chunks based on the embedded query.

   The quality of similarity search is crucial, as it directly impacts the relevance and accuracy of the LLM-generated response. In the embedding space, proximity denotes semantic similarity—closer vectors imply greater contextual relevance. However, efficient neighbor discovery in high-dimensional vector space is computationally expensive, and full database scans are often impractical [45].

b. **Search Algorithms:** To balance accuracy, latency, and resource usage, the following approximate nearest neighbor (ANN) algorithms are commonly employed:

   (a) **Cosine Similarity Search:** Measures the cosine of the angle between vectors to determine their similarity. It is widely used in embedding-based retrieval tasks

due to its scale invariance and intuitive geometric interpretation. This is **highly parallelizable** and simple to implement. [46]. *This is project primarily relies on cosine similarity for embedding retrieval.*

(b) **k-Nearest Neighbors (kNN):** A brute-force method that identifies the k closest vectors through exhaustive comparisons [47].

This project uses kNN as a fallback mechanism to search in the database using pg-vector [48].

(c) **HNSW (Hierarchical Navigable Small World) Graphs:** Constructs a layered graph to enable fast traversal through neighbors across different granularity levels [47].

(d) **ScaNN (Scalable Nearest Neighbors):** A Google-designed ANN method that integrates pruning, quantization, and partitioning for scalable, memory-efficient retrieval [49].

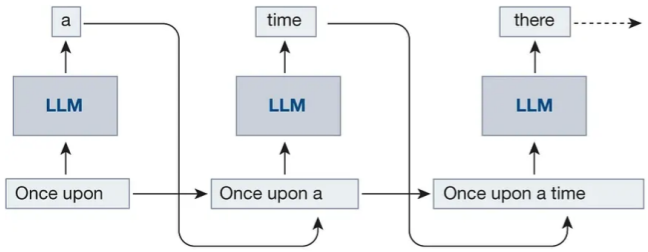


Figure 2.7: Autoregressive decoding [2]

c. **Output Generation:** In this phase, the retrieved context and the user input—both in plain text form—are fed into the Large Language Model, as illustrated in Figure 1.3. The LLM processes the portion of input that fits within its context window and generates an output token. This token is then concatenated with the input and passed

back to the LLM. The process repeats iteratively until the LLM produces an ¡EOS¿ (end-of-sequence) token.

2.4.4 Output Evaluation:

The evaluation of a Retrieval-Augmented Generation (RAG) system focuses on the quality of the output in terms of its relevance to the user's query and the information stored in the vector database. Since different components contribute uniquely to the overall performance, multiple metrics are used rather than a single aggregated score to allow for precise attribution and targeted optimization.

Classical metrics such as Precision, Recall, and F1 score, alongside NLP-specific metrics like ROUGE, are employed to assess performance. The evaluation considers several factors, as illustrated in Figure 1.5:

- **Faithfulness, Output Relevance & Semantic Similarity:** Measures the quality of the output relative to the input queries and retrieved context.
- **Context Recall & Context Precision:** Assess the effectiveness of context retrieval and its utilization by the LLM.

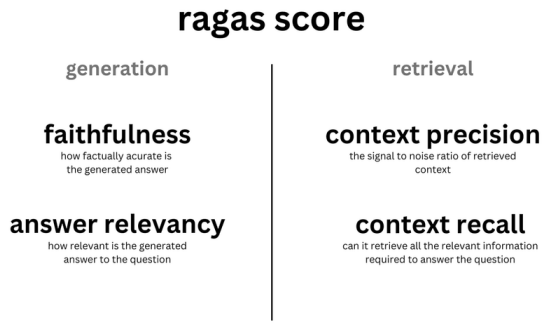


Figure 2.8: RAG Output evaluation metrics [6]

Chapter 3

RELATED WORK

This chapter presents an overview of existing technologies and research efforts relevant to the development of efficient, on-device language model-based applications. It focuses on three major areas: Retrieval-Augmented Generation (RAG), efficient LLM runtimes like `llama.cpp`, and lightweight distribution and serving solutions such as Ollama and Llamafile. It also looks at the work done by the tinygrad project in attempting to reverse engineer the NPU API.

3.1 RAG Frameworks and Agentic Systems

Retrieval-Augmented Generation (RAG) has become a widely adopted paradigm to enhance the factual grounding of large language models (LLMs) by integrating external knowledge through retrieval mechanisms. To support this architecture, several frameworks have been developed to facilitate the construction of RAG pipelines. **LangChain** [50] provides a composable and extensible framework for chaining LLMs with retrieval components, supporting a broad ecosystem of vector stores, retrievers, and tools. **LlamaIndex** (formerly GPT Index) [51] offers structured pipelines for indexing and querying private or domain-specific data sources. **Haystack** [52] by deepset provides a modular toolkit for constructing production-ready pipelines with retriever-reader architectures.

In parallel, the emergence of **agentic systems**—LLM-powered agents capable of tool use, memory, and reasoning—has opened new avenues for task-oriented automation. Frameworks such as LangChain [50] and AutoGen [53] support multi-agent orchestration and planning. These systems increasingly incorporate **LLM routing** strategies [54] that dynamically select or combine multiple models or tools based on the query or context, enhancing efficiency and

specialization. Additionally, **RAGAS** [55] introduces an evaluation framework to measure the quality and factual alignment of RAG outputs, contributing to the robustness of such systems in real-world applications. Together, these frameworks and methodologies form the foundational ecosystem for deploying scalable, intelligent, and grounded LLM applications.

Although this project does not directly leverage any existing RAG frameworks yet, it certainly has drawn inspiration and insights wherever applicable.

3.2 llama.cpp

`llama.cpp`[56] is a C++ implementation of large language models (Meta LLaMA to begin with), optimized for local inference on commodity hardware without GPU requirements. Built upon the GGML tensor library, it provides quantized inference for large models using CPU-friendly formats like 4-bit or 5-bit quantization, making it suitable for running models such as LLaMA, Mistral, and other open-weight transformers on devices ranging from laptop to Raspberry Pi.

Key features of `llama.cpp` include:

- Highly efficient CPU inference with quantized models.
- Cross-platform support (macOS, Linux, Windows).
- Integration with popular tooling such as LangChain and Open Interpreter.
- Support for multi-threaded inference and memory-mapped model weights for efficient memory usage.

Llama.cpp serves as a foundational component for many desktop LLM applications that prioritize local execution and privacy-preserving computation. *In this project (Project TLDR), llama.cpp is used for LLM inference tasks, including both embedding generation and text generation.*

3.3 Ollama

Ollama is a developer-friendly platform for running LLMs locally with simplified model management and serving. It wraps models like LLaMA 2, Mistral, and Code LLaMA into a streamlined runtime with a CLI and RESTful API, abstracting away hardware-specific setup and providing a plug-and-play experience for developers [57].

Ollama supports:

- Running quantized models locally with GPU acceleration where available.
- Seamless model downloading and serving.
- Custom model creation using a simple Modelfile syntax.

It is widely used for prototyping private, offline chatbots and assistants. However, the users need to be technically savvy in cases of issues downloading or running the models. Furthermore, models often may not be quantized and could lead to downloading of model weights in order of many GBs.

3.4 Llamafile

Llamafile, developed by Mozilla-Ocho, enables packaging a complete LLM runtime into a single, self-contained executable file [58]. It leverages the `llama.cpp` backend and Cosmopolitan Libc to build universal binaries that run across major operating systems (Windows, macOS, Linux) without requiring dependencies.

Notable features:

- Distributable as a single file under 1GB (depending on model).
- Useful for shipping LLM-based tools with zero-install requirements.
- Integrates with web frontends for local chatbot deployment.

Llamafile is widely used for prototyping private, offline chatbots and assistants. However, it often requires users to be technically proficient, especially when encountering issues related to downloading or executing models. Moreover, many models are not pre-quantized, potentially resulting in downloads of several gigabytes of model weights.

3.5 Tinygrad project and Apple Neural Engine (ANE)

The Apple Neural Engine (ANE) is a custom neural processing unit designed by Apple to accelerate machine learning workloads on its silicon platforms. Introduced with the A11 Bionic chip, the ANE has evolved into a high-performance, low-power DMA-based inference engine embedded in Apple's M-series chips. This section synthesizes insights obtained by the reverse-engineering efforts of the `tinygrad` [24] project to examine ANE's architecture, capabilities, and compilation flow.

3.5.1 Hardware Overview

The ANE operates primarily as a DMA engine optimized for convolutional operations and supports a wide range of neural network layers and fused operations. Its key hardware features include:

- **16-core architecture:** A 16-wide Kernel DMA engine for parallel computation.
- **5D Tensor Support:** Tensors are structured with width (column), height (row), planes (channels), depth, and group (batch).
- **Supported Data Types:** `UInt8`, `Int8`, and `Float16` (with `Float32` inputs automatically downcast).
- **Manually Managed 4MB L2 Cache:** Applied only to input/output data; weights are embedded in the compiled program.
- **Execution Unit:** Executes up to 0x300 micro-operations per instruction.

- **Performance:** Approximate 11 TOPS throughput, assuming 32×32 MAC at 335 MHz.

All memory strides are constrained to multiples of 0x40 bytes, reflecting hardware alignment requirements.

3.5.2 Software and Compilation Stack

The ANE software stack is heavily abstracted behind Apple's proprietary frameworks but has been reverse-engineered to reveal a structured flow (as shown in Fig 3.1):

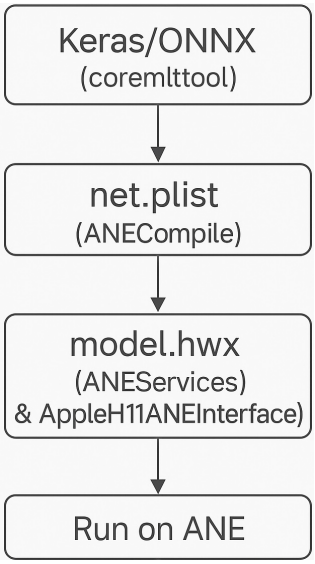


Figure 3.1: Apple Neural Engine Workflow

1. **Model Definition:** Models are authored in Keras or ONNX.

2. **Conversion:** Models are converted to CoreML format using open-source tools such as `coremltools`.

3. **Intermediate Representation:** CoreML is internally converted into `net.plist` by Apple's Espresso framework.

4. **Compilation:** The `ANECompiler` service transforms `net.plist` into a hardware-specific binary (`.hwx`), a Mach-O formatted executable.

5. **Execution:** The `AppleNeuralEngine` and `ANEServices` handle execution via the kernel extension `AppleH11ANEInterface`.

3.5.3 Instruction Format and Operation Structure

Each ANE instruction is 0x300 bytes and comprises multiple segments:

- **Header:** Includes DMA addresses and next-op offset.
- **KernelDMASrc:** Specifies weights, bias, and channel usage.
- **Common:** Describes input/output shapes, types, kernel size, and padding.
- **TileDMASrc/TDMADst:** Layout and stride configurations for input/output tensors.
- **L2 and NE:** L2 cache flags and activation parameters.

3.5.4 Supported Operations and Activations

ANE supports a variety of operations:

- **Core Ops:** CONV, POOL, EW, CONCAT, RESHAPE, MATRIX_MULT, TRANSPOSE
- **Advanced:** SCALE_BIAS, SOFTMAX, INSTANCE_NORM, BROADCAST, L2_NORM
- **Fused Ops:** NEFUSED_CONV, PEFUSED_POOL, etc.
- **Activations:** RELU, SIGMOID, TANH, CLAMPED_RELU, PRELU, LOG2/EXP2, CUSTOM_LUT

Over 30 activation functions are supported in hardware.

3.5.5 tinygrad Implementation

The `tinygrad` project interfaces directly with ANE through a three-stage pipeline:

- **1_build:** Generates CoreML models using `coremltools`.
- **2_compile:** Uses Objective-C and Apple's private ANECompiler framework to compile models into HWX binaries.
- **3_run:** Loads HWX binaries and executes them on ANE using custom Objective-C wrappers around `AppleH11ANEInterface`.

The implementation also includes tools like `hwx_parse.py` for disassembling HWX files and visualizing internal ops.

3.5.6 Security and Access

Execution on ANE requires system entitlements that are typically unavailable to third-party applications:

- `com.apple.ane.iokit-user-access`
- Workarounds: amfid patching, kernel extension modification, or use of provisioning profiles.

3.5.7 ANE Takeaways

The ANE represents a proprietary, highly optimized inference accelerator that is difficult to access and understand due to Apple's closed ecosystem. Reverse engineering, as demonstrated by `tinygrad`, reveals a modular, DMA-centric architecture capable of executing complex neural network operations at high throughput and low latency. As Apple continues

to iterate on the ANE, deeper access and tooling may unlock broader ML deployment options on Apple hardware.

Chapter 4

METHODOLOGY

This chapter outlines the technical methodology adopted in the development of the system, covering both low-level design and software implementation. The approach prioritizes performance, privacy, and modularity, leveraging hardware accelerators where possible and maintaining efficient control over data flow and execution.

4.1 Application Modules and Design

This section provides an overview of the internal architecture and design principles behind the TLDR desktop application. The application is built with the goal of providing a fast, private, and efficient interface for question-answering and summarization tasks over a user-defined corpus of documents. To achieve this, the design incorporates several performance-aware and hardware-conscious modules, especially tailored for Apple's M1/M2 architecture.

The TLDR system is structured into modular components, each responsible for a specific functionality in the information processing pipeline. These include embedding generation, context retrieval, vector storage, prompt construction, and output generation via a language model. The workflow between these modules is coordinated to support seamless execution, low latency, and high responsiveness, all while maintaining data privacy by running entirely on-device.

4.1.1 Overview

The TLDR application follows a modular architecture where different components are responsible for distinct tasks in the RAG (Retrieval-Augmented Generation) pipeline. Figure 4.1 illustrates the overall design and the control flow between the modules.

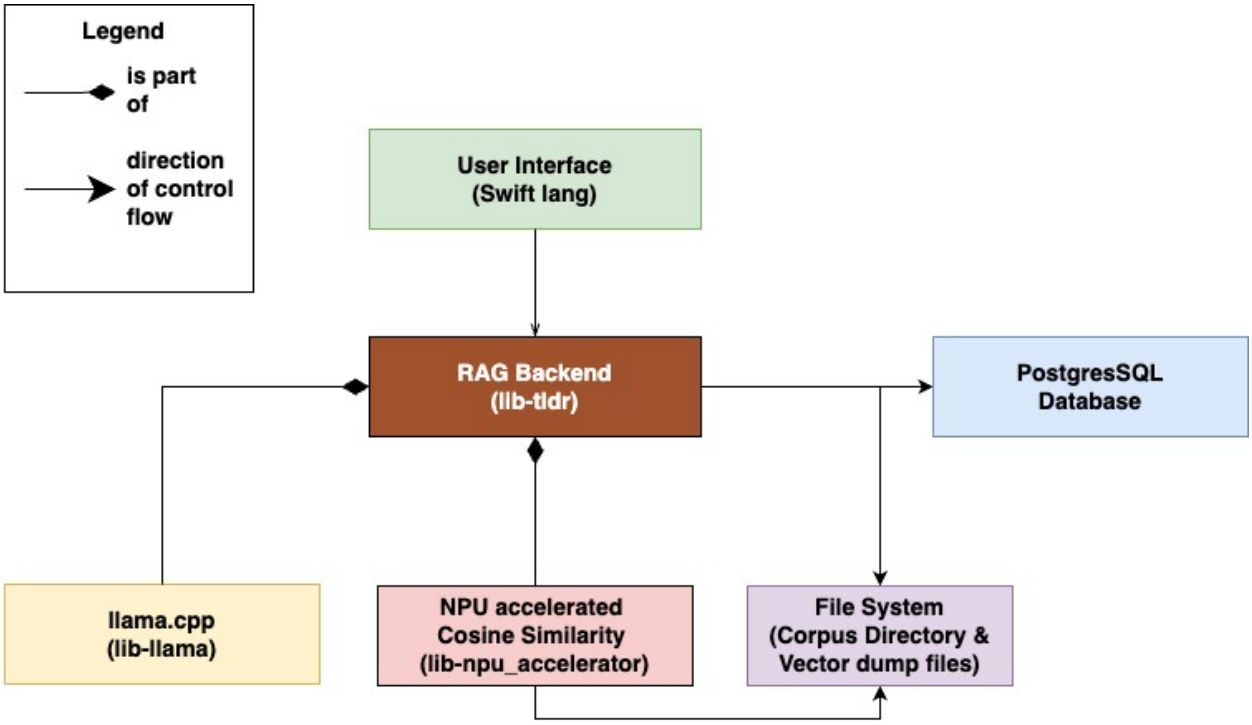


Figure 4.1: Modules of TLDR application

- **User Interface:** This module provides the graphical frontend for the user. Developed using Swift language for MacOS, it allows users for the users to seamlessly leverage the capabilities of the application. It is responsible for workflows dealing with user experience while delegating all core logic to the backend modules.

- **RAG Backend:** This is the core orchestrator of the application. It manages the full pipeline, including handling user queries, initiating vector search, performing retrieval, and forwarding context to the language model. It communicates with all supporting modules such as the database, NPU based vector search module, llama.cpp and the file system.

- **Database (PostgreSQL):** Stores metadata and document indexing information. It ensures efficient retrieval and persistence of preprocessed documents and vector refer-

ences. It is plays a crucial role of mapping retrieved vector hashes to their text chunks and document metadata during context retrieval.

- **File System (Corpus Directory and Vector Dump Files):** Contains the document corpus (directory containing source documents) and their corresponding vector dump files. These vector dump are leveraged by the vector search engine using memory-mapped I/O for efficient vector search with low memory overhead.

- **NPU Accelerated Cosine Similarity:** Implements hardware-accelerated cosine similarity by leveraging Apple's Neural Processing Unit (NPU). The backend invokes this module for fast and parallelizable vector cosine similarity computation.

- **llama.cpp:** This module is responsible for language generation i.e LLM inference. It acts as a plugged-in module for the RAG backend and contributes by generating embeddings and chat response during the corresponding stages of the RAG pipeline.

4.1.2 User Interface

The User Interface is in the form of a native MacOS desktop application developed using Swift lang in the Xcode development environment (as depicted in Fig 4.2, Fig 4.3). The user interface is designed to be intuitive, lightweight, and self-contained. The application packages all necessary dependencies, including static libraries and LLM weights. enabling fully offline functionality without requiring additional installation or configuration.

The user interface module has the following core purposes:

- Provides a clean, chat-style interface where user prompts and LLM responses are displayed as distinct messages to mimic a conversational flow.

- Manager all responsibilities regarding user interaction, including persisting user conversation history and any additional user preferences and thereby enable RAG backend to only focus on the RAG functionalities.

- *Make a single, self contained portable package that is easy and intuitive to distribute.*

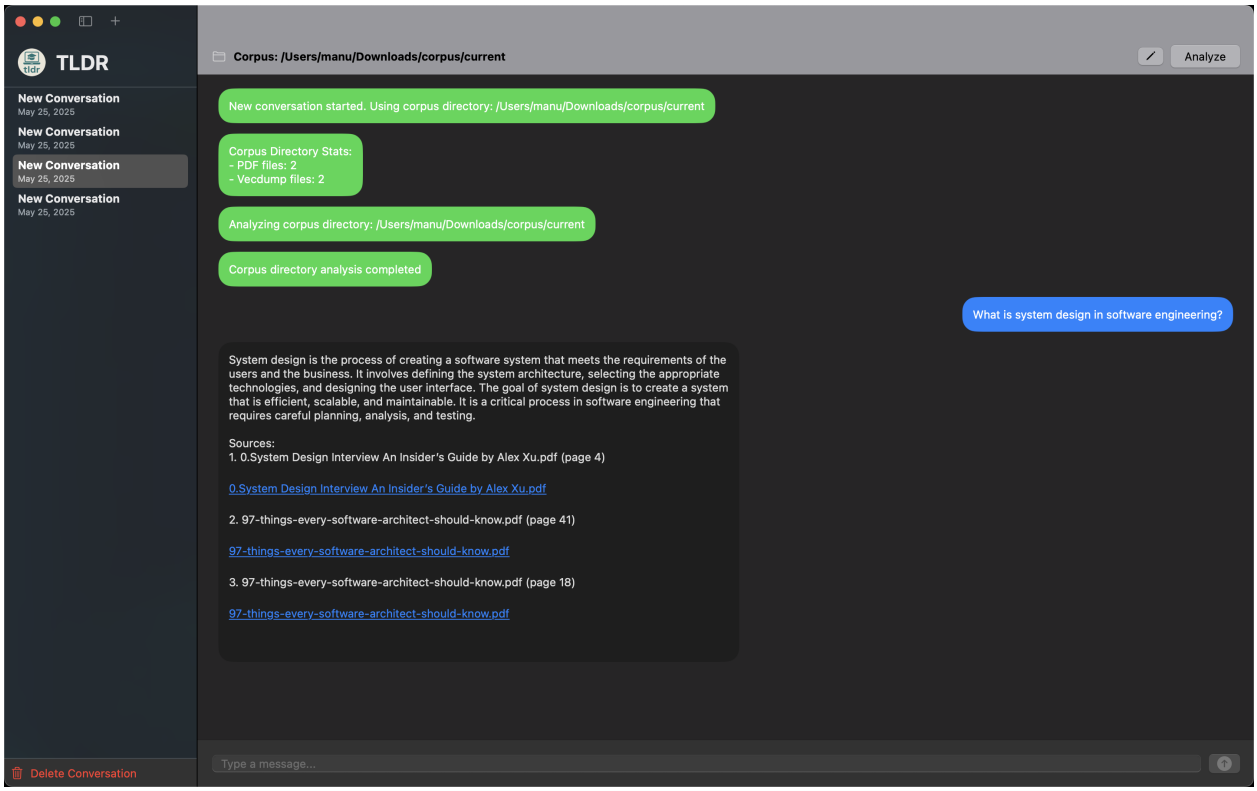


Figure 4.2: Graphical User Interface of TLDR Application

Figure 4.3: TLDR Application Icon as seen in MacOS Dock

Codebase Organization

The codebase of the GUI application is illustrated in Figure 4.4. It is organized into several components that serve both core functionality and supporting roles within the application.

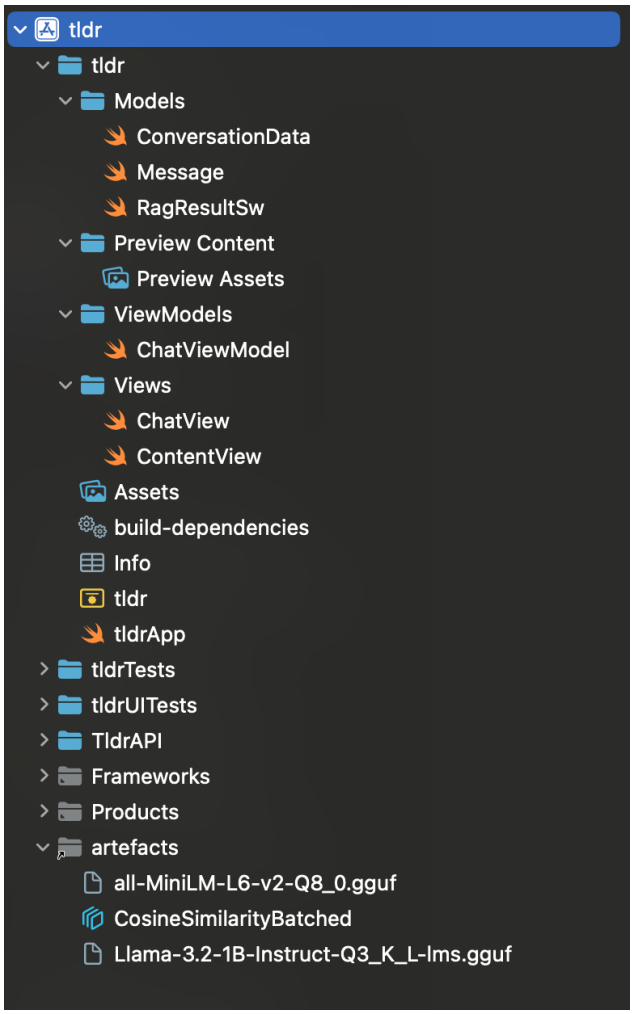


Figure 4.4: UI Project File system

Structure:`tldr`: The Swift language codebase that creates the user interface. `TldrAPI`: C++ module with bindings for Swift UI. It serves as a bridge between the Swift UI and the C++ static library of the RAG Backend.

`artefacts`: Quantized LLM weights and coreml packages, i.e:

b.
- `Llama-3.2-1B-Instruct-Q3_K_L` model for chat
- `all-MiniLM-L6-v2-Q8_0` model for generating embeddings
- `CosineSimilarityBatched` coreml package for cosine similarity on NPU

A. UI Architecture (MVVM):

a. `tldrApp`: The main SwiftUI entry point where the application lifecycle begins.

b. Models: Includes `ConversationData`, `Message`, and `RagResultSw` to represent the chat state and RAG outputs. These modules leverage *UserDefaults* a built-in key-value persistance mechanism provided by Apple's Foundation framework to efficiently store and retrieve their information.

c. Views: Comprises SwiftUI components like `ChatView` and `ContentView` to render the main interface.

d. ViewModels: Contains `ChatViewModel`, which handles user interaction and backend coordination.

e. Preview Content: Includes `Preview Assets` for SwiftUI previews to support development and layout testing.

f. Assets: Stores static resources such as icons and other UI elements.

This organization promotes maintainability and allows for a clear separation of concerns between UI presentation, interaction logic, and backend communication.

4.1.3 RAG Backend

The RAG backend is implemented as a C++ static library named `lib_tldr`. It serves as the backbone for all core functionalities of the project. This module encapsulates the complete RAG pipeline logic and integrates essential components such as the large language model for text and embedding generation, the vector dump generator and reader, database and

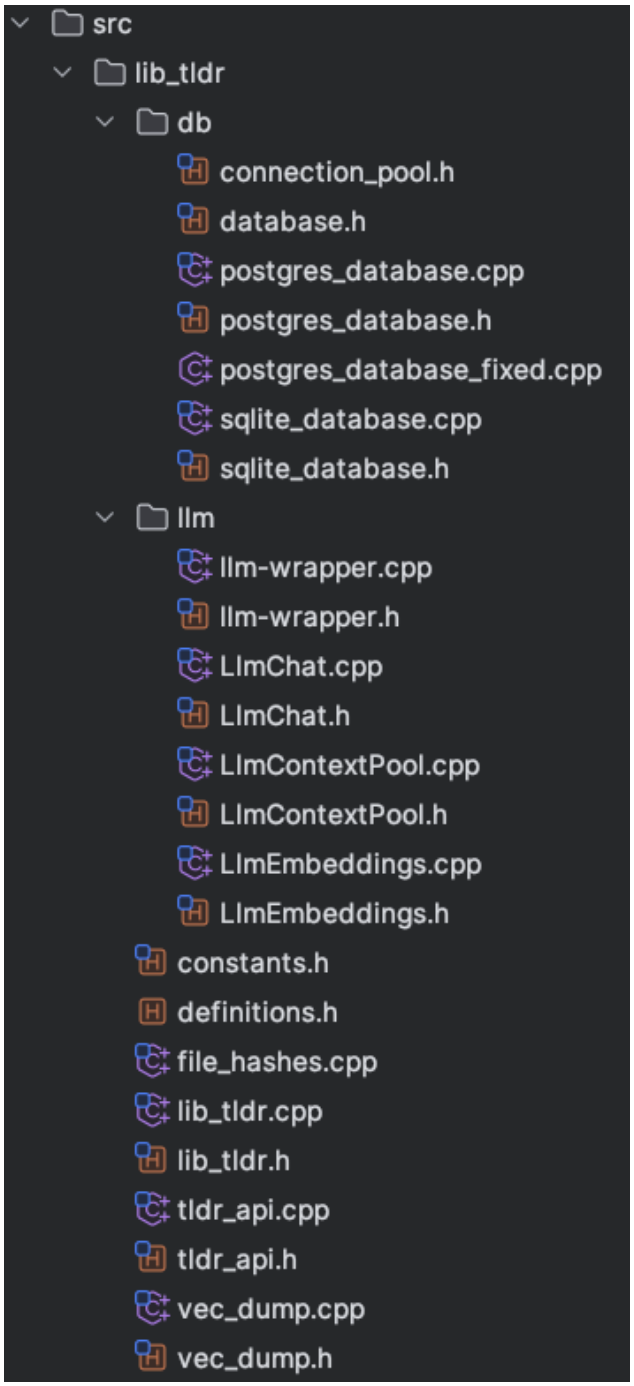


Figure 4.5: RAG Backend(lib_tldr) codebase

file system handlers for storage, and the cosine similarity-based vector retriever (as seen in Fig 4.5).

The architecture emphasizes modularity and resource efficiency, enabling plug-and-play replacement or extension of components. It adheres to the SOLID principles of object-oriented programming, promoting adaptability, maintainability, and scalability.

Following are the logical sub-modules of RAG Backend:

A. Language Model Interface:

The system utilizes `llama.cpp` as the backend for both text generation and embedding extraction. The following components facilitate this integration:

(a) `LlmChat.cpp &.h`: Manage the logic to take user input and retrieved context and generate a response from the chat LLM.

(b) `LlmEmbeddings.cpp &.h`: These modules extract dense vector representations (embeddings) from document chunks, which are subsequently used for semantic similarity search and retrieval.

(c) `LlmContextPool.cpp &.h`: These components manage the lifecycle of LLM context objects, which encapsulate the model state necessary for efficient inference. Since LLMs typically require a context to maintain internal buffers, tokenizer state, and memory allocations, it is computationally expensive to initialize a new context for every query or embedding operation. By pooling and reusing contexts across multiple operations, the system significantly reduces initialization overhead and ensures smoother, low-latency performance during both chat and embedding workflows.

(d) `Llm-Wrappers.cpp &.h`: Takes care of initializing and cleaning up resources related to the LLMs. It initializes common ggml backend for Apple metal and setting up LLM context pools and initialize Chat and Embedding LLMs by loading their weights into memory.

B. Database Interaction: All persistent storage is handled via the PostgreSQL backend, though an optional SQLite backend is also implemented (but not currently utilized). The database interaction is managed within the `db` submodule, which includes the following:

   a. `database.h` Defines an abstract class that enforces a uniform interface for any underlying database implementation. This design allows the RAG codebase to remain unchanged when switching between different database technologies.

   b. `postgres_database.cpp &.h`: These files handle database initialization, schema definition, and CRUD operations related to documents and embeddings in PostgreSQL Database.

   c. `connection_pool.h`: Provides a lightweight connection pooling mechanism to manage multiple concurrent database sessions efficiently. This is especially beneficial during large-scale embedding operations where multiple inserts are performed rapidly. It is efficient to store readily available connections and re-use them instead of creating a new connection for every db interaction.

   The PostgreSQL schema stores both high-level document metadata (e.g., title, author, page count) and low-level embedding-related information (e.g., text chunk, hash, embedding vector, page number, and timestamps).

C. Embeddings Vector Storage and Retrieval

   `vec_dump.cpp &.h` are responsible for managing the serialized storage of raw vector data ("vecdumps"). These are binary representations of embedding vectors that can be rapidly accessed and processed.

   Additionally, the system incorporates a hardware-accelerated module referred to as the `npu-accelerator`, which leverages macOS's Neural Engine to perform cosine similarity

search over large sets of embeddings. This offloads compute-intensive operations from the CPU, enabling real-time retrieval performance on resource-constrained devices.

D. Core Workflow: The `lib_tldr.cpp &.h` contains the core logic that glues the entire system together, such as:

- Initializing the LLMs and the Database tables (when necessary)
- Creating DB connection pool and LLM context pool
- Checking for changes in the corpus directory and embedding new documents
- Performing Retrieval Augmented Generation
- Cleaning up and releasing acquired resources

E. RAG Backend API: While the backend contains numerous functions and data structures for internal logic, a clean and minimalistic API (Application Programming Interface) is exposed. This allows its client modules (such as the UI module) to leverage its capabilities without being closely coupled with the internal mechanisms of the library.

The `tldr_api.cpp &.h` files expose a clean, C-style API interface for the user-facing application layer on top of the functions present in `lib_tldr.cpp &.h`.

4.1.4 Database (PostgreSQL)

The application uses PostgreSQL as its persistent storage backend to manage and retrieve embedding data required during the Retrieval-Augmented Generation (RAG) process. The database stores preprocessed document chunks, their vector embeddings, associated metadata, and file-level information.

PostgreSQL was chosen over lighter-weight alternatives like SQLite due to its superior concurrency handling. Specifically, SQLite's single-writer limitation presented a bottleneck in the multi-threaded embedding pipeline, where concurrent writes to the embedding store

are common. PostgreSQL's support for multiple concurrent writers allows the embedding process to scale efficiently without serialization delays.

Table: documents

This table stores metadata for each unique input file in the corpus. It ensures file-level uniqueness through the `file_hash` field and includes fields such as file name, author, subject, page count, and timestamps. It acts as a parent entity in a one-to-many relationship with the `embeddings` table (refer to table definition in Fig 4.6).

	column_name name	data_type character varying	is_nullable character varying (3)
1	id	uuid	NO
2	page_count	integer	YES
3	created_at	timestamp with time zone	YES
4	updated_at	timestamp with time zone	YES
5	title	text	YES
6	author	text	YES
7	subject	text	YES
8	keywords	text	YES
9	creator	text	YES
10	producer	text	YES
11	file_hash	text	NO
12	file_path	text	NO
13	file_name	text	NO

Figure 4.6: Documents table description

Table: embeddings

This table stores the chunk-level data required for semantic search. Each row contains a reference to a document, the original text chunk, its embedding vector, and an embedding

hash. During query time, the vector search module returns the top-K most similar vectors based on cosine similarity. The corresponding text chunks are then retrieved from this table using the hash and document ID to form the LLM context (refer to table definition in Fig 4.7).

	column_name name	data_type character varying	is_nullable character varying (3)
1	created_at	timestamp with time zone	YES
2	page_number	integer	YES
3	document_id	uuid	YES
4	id	bigint	NO
5	embedding	USER-DEFINED	NO
6	embedding_hash	text	YES
7	chunk_text	text	NO

Figure 4.7: Embeddings table description

4.1.5 File System: Corpus Directory and Vectordump Files

Vector dump files are binary data structures designed for efficient storage of document embeddings. For each input document, a corresponding vector dump file is created. Each such file contains embedding vectors generated from text chunks along with their corresponding MD5 hashes. This format enables fast similarity search and content verification in document retrieval systems.

Corpus Directory

The corpus directory serves as the source of truth for the RAG pipeline. It contains the raw documents—primarily PDF files—that are parsed, chunked, and processed by the language model to construct the knowledge base used for information retrieval.

This project recursively scans the specified corpus directory, identifies all PDF files, and computes their corresponding embeddings and stores them in **v̇ecdump** files.

Vectordump Files

Vector dump files are obtained as a result of the embedding process. After a document is split into chunks, each chunk is processed by the LLM and yields embeddings. Embeddings are higher dimensional representations of the input text, as interpreted by the LLM. These embeddings are then stored in a dedicated subdirectory named *_vecdump*, located within the corpus directory itself. The *_vecdump* folder houses binary `.vecdump` files that are later used during the similarity search phase to efficiently retrieve semantically relevant chunks.

The vector dump file follows a sequential binary layout consisting of three main components: a metadata header, followed by embedding data, and finally hash data (as illustrated in Fig 4.8. This structure allows for efficient random access to embeddings while maintaining data integrity through hash verification. Hash is also further used for fetching the corresponding text chunk after similar vectors are obtained for a query.

Vectordump Header

In order to process the vector dump file, the header is first read and the necessary information is obtained regarding the data layout in the file. The information is arranged in the layout as illustrated in Figure 4.8. The elements are as follows:

- `num_entries` - Total number of embedding/hash pairs stored in the file
- `hash_size_bytes` - Size of each MD5 hash in bytes (always 16 for standard MD5)
- `vector_size_bytes` - Total byte size of each embedding vector
- `vector_dimensions` - Number of floating-point dimensions per embedding vector

The header is read by simply pointing a pointer of type $structVectorDumpHeader$ to the memory location to which the file is loaded. This helps obtain the necessary information required to access the data sections of the file.

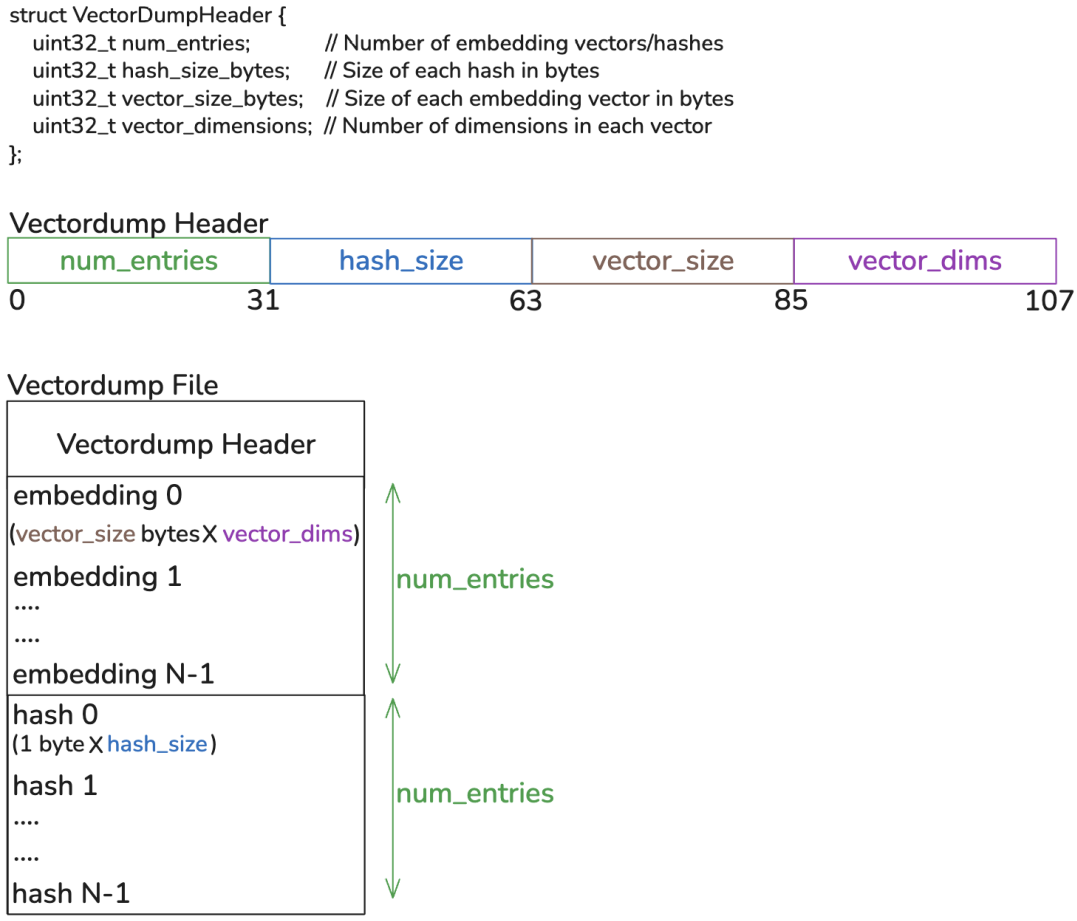


Figure 4.8: Vector dump file structure

Data Sections

Once the header section is loaded, the data obtained is then used for calculating the memory locations of the embedding and hash arrays. Pointers are used to access these locations to simulate the structure of an array on top of raw binary data read into the memory. This approach is simple and efficient and prevents needless memory allocations and data copies.

Embeddings: Contains N consecutive embedding vectors, where $N =$ `num_entries`. Each vector occupies `vector_size_bytes` and represents a `vector_dimensions`-dimensional

embedding, stored as 32-bit floating point values. In the current case, we use a 384-dimensional vector embedding.

Hashes: Contains N consecutive MD5 hash values, each exactly 16 bytes. However, the smallest unit of storage is of $uint64_t$ type, i.e., units of 2 bytes. Hence, a 16 byte MD5 hash would have a hash size of 8. The hash at index i corresponds to the MD5 digest of the original text chunk used to generate `embedding[i]`.

Data Relationship

The file maintains strict positional correspondence: for any index $i \in [0, N-1]$, `embedding[i]` and `hash[i]` represent the same document chunk. This one-to-one mapping enables efficient lookup operations and integrity verification during retrieval.

The data is used as follows:

1. Load the first half of the file in memory using **mmap** and perform cosine similarity search.

2. Obtain index of the top K relevant vectors from Cosine similarity module.

3. Fetch the hash values at the obtained indices.

4. Query the database for text chunks associated with the hash values.

This design allows for prioritized access to necessary data and its direct usage for cosine similarity search with no further processing or data manipulations, allowing for an efficient search through the entire corpus.

4.1.6 NPU Accelerated Cosine Similarity

The NPU accelerator module (lib-npu_accelerator) is a specialized component of the TLDR MacOS desktop application that leverages Apple's Neural Processing Unit to perform hardware-accelerated cosine similarity computations as part of the RAG pipeline. The module con-

struction and usage is a multi-step process involving multiple components. The codebase structure is depicted in Fig 4.9 and the workflow in Fig 4.10.

Codebase structure

The codebase for the npu module as seen in Fig 4.9 has the following components:

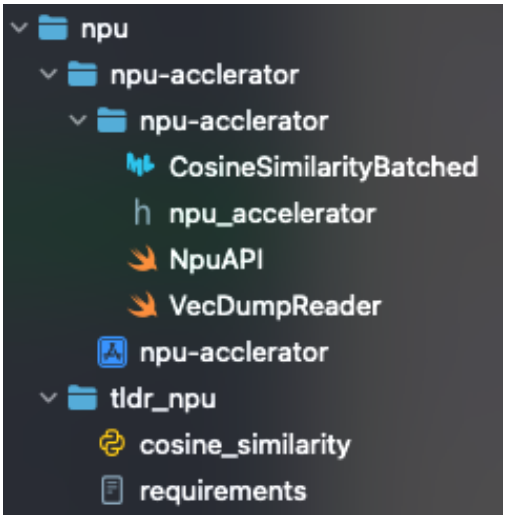


Figure 4.9: lib-npu_accelerator codebase

- **PyTorch module (`tldr_npu`):** Contains PyTorch code to perform batched cosine similarity. This module is used to generate a CoreML model package, which is later utilized by the NPU accelerator for high-performance similarity computation.

- **Swift module (`npu-accelerator`):** Implements the logic in Swift to read vector dump files and leverage the CoreML cosine similarity model to perform efficient vector similarity search using the Apple Neural Engine (ANE).

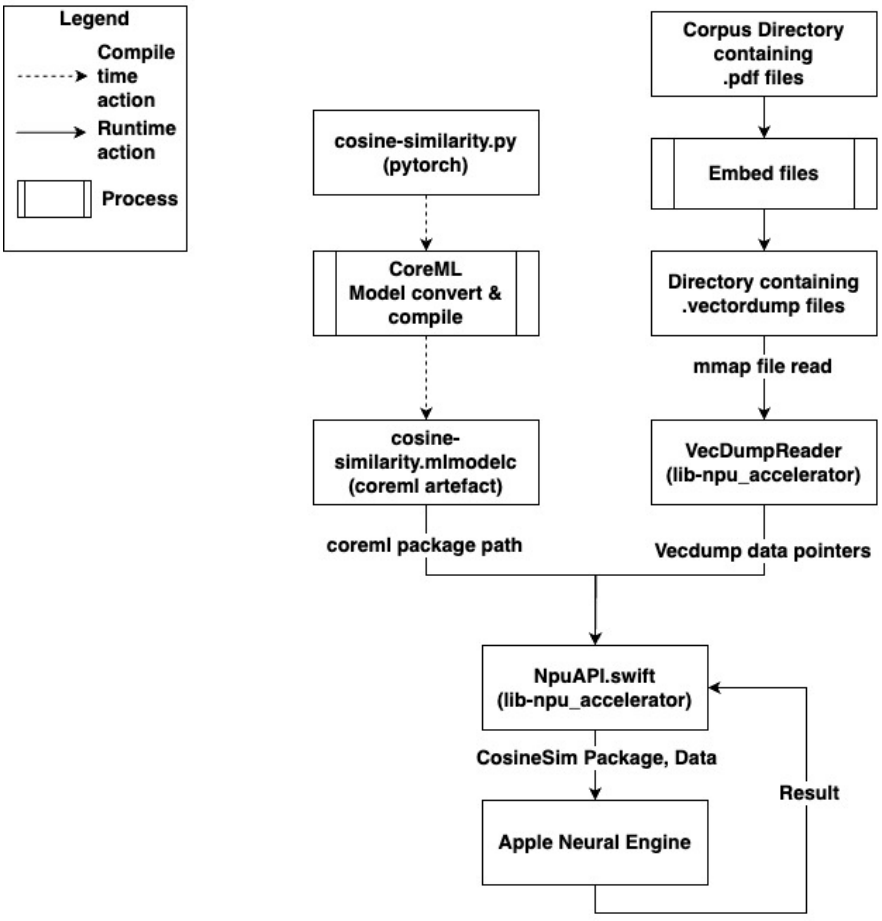


Figure 4.10: NPU accelerator module workflow

Processing Workflow

The NPU accelerator workflow could be visualized as dual-pipeline workflow as illustrated in Fig 4.10.

Phase I - Prepare the components: The left pipeline begins with a PyTorch-based cosine-similarity.py implementation that serves as the foundation for similarity computations. This PyTorch model undergoes a CoreML model conversion and compilation process, transforming the original implementation into an optimized CoreML artifact specifically designed for Apple's Neural Engine execution. The compilation step produces the *cosine-*

similarity.mlmodelc package, which contains the optimized model ready for hardware acceleration.

The right pipeline operates in parallel, handling document processing and vector storage. Although this portion of the pipeline is executed by the RAG Backend and is not directly part of the NPU module, it still is a logical component of the NPU module workflow as depicted in Fig 4.10.

Phase II - Perform vector search: This phase is triggered at runtime when a user submits a query and the RAG pipeline is activated. The workflow converges at `NpuAPI.swift`, a Swift-based interface that integrates the CoreML similarity model and the document embeddings obtained via the `VecDumpReader`.

The `VecDumpReader` accesses vector dump files using memory-mapped I/O (`mmap`), exposing the data as raw pointers without intermediate copies. These pointers are passed directly to the `CosineSimilarityBatched` module, which performs batched cosine similarity computations using Apple's Neural Engine (ANE).

`NpuAPI` orchestrates this process by coordinating the CoreML model execution and the memory-resident embeddings, enabling efficient hardware-accelerated similarity search. The output consists of similarity scores and embedding hash values, which are used by the RAG system to retrieve the most relevant document chunks for generating a contextually informed response. Furthermore, `NpuAPI` also serves as a bridge to the C++ layer, exposing these capabilities to the RAG backend for use during the retrieval phase of the RAG pipeline.

Although cosine similarity involves a full scan of the embedded corpus for each query, the system architecture mitigates performance concerns through the following mechanisms:

- **Shared memory-mapped files:** When multiple threads handle different user requests, redundant file reads are avoided because the `mmap` mechanism loads the file into memory only once.

- **Zero-copy access:** Since the data is accessed directly from memory without duplication, there is no overhead for repeated reads. Additionally, the operating system

can page the data out to swap memory and page it back in when needed, optimizing memory usage.

- **Unified memory architecture:** The Apple M1/M2 SoC's unified memory architecture enables seamless access between CPU, GPU and NPU, eliminating the need for further data transfers during vector similarity computations.

4.1.7 llama.cpp

The project integrates a customized fork of `llama.cpp`—a lightweight C++ inference engine for LLaMA and related transformer models—to serve as the core engine for both text generation and embedding extraction. The fork is hosted at `https://github.com/manuhg/llama.cpp`, and includes minor modifications that streamline the build process for macOS targets. Specifically, the build scripts were stripped down to produce a static library using the `ggml` Metal backend, optimized for Apple Silicon devices. The result is a single, portable C++ static library named `libllama.a` which is then statically linked with the RAG backend (`lib-tldr.a`).

The core functionalities such as tokenization, decoding, and sampling are accessed through the public APIs exposed in `llama.h` and `ggml.h`. Two primary components—`LLmChat` and `LLmEmbedding`—are built around workflows inspired by simple.cpp [59] `server.cpp` [60] and `embedding.cpp` [61] from the upstream `llama.cpp` [62] project.

Further, to improve the performance, `OpenMP` support was added for parallelizing the tokenization and batch decoding steps. This optimization ensures efficient utilization of CPU cores, resulting in faster preprocessing and inference, particularly during multi-threaded interactions.

The `llama.cpp` is hence directly integrated into the RAG backend at compile time. This enables the application to perform in-memory LLM inference by directly invoking components of `llama.cpp`, without relying on any external dependencies or background processes for this core functionality.

4.2 Application Design - Workflow Overview

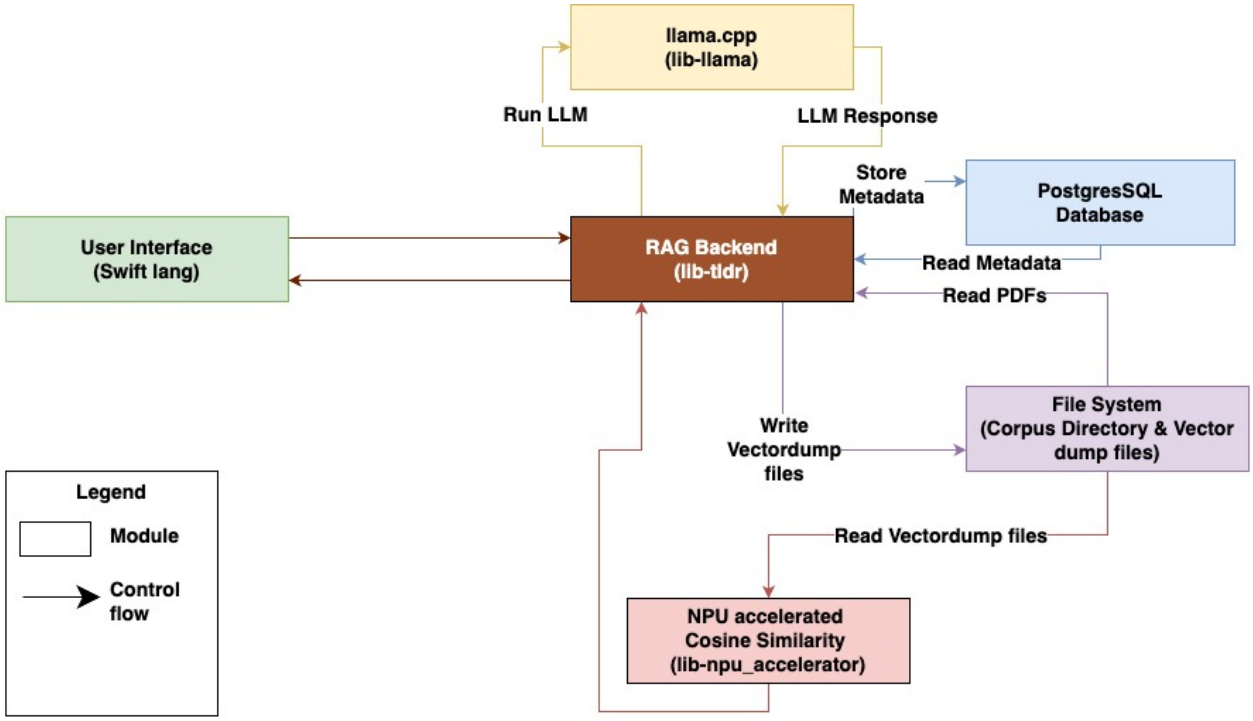


Figure 4.11: RAG Output evaluation metrics [6]

Figure **??** illustrates the end-to-end workflow of the TLDR application. It is divided into three primary phases: system initialization, document corpus embedding, and query-based retrieval-augmented generation (RAG).

System Initialization

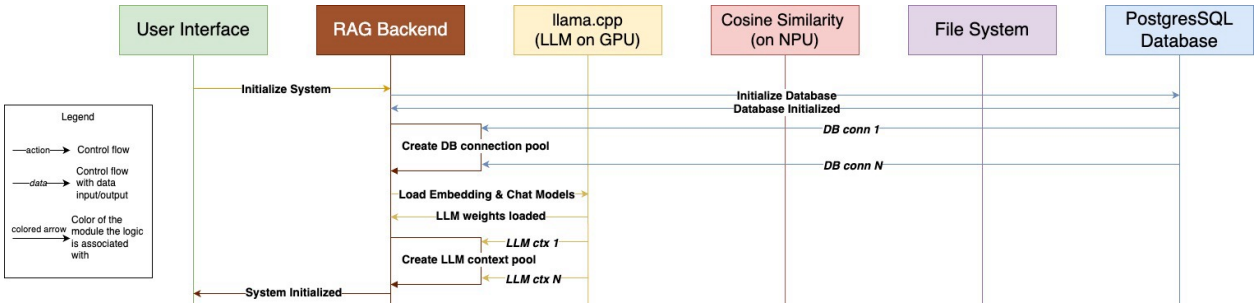


Figure 4.12: RAG Output evaluation metrics [6]

The initialization phase sets up the backend infrastructure and prepares the application for use. This includes:

- The user launches the application, triggering the backend.
- The **RAG Backend** initializes a connection pool to the **PostgreSQL Database**.
- The LLM weights and context pools are loaded via `llama.cpp`, enabling multi-threaded inference.
- Cosine similarity routines are prepared via the **NPU accelerator** module.
- System status is communicated back to the **User Interface**, indicating readiness.

Embedding the Corpus

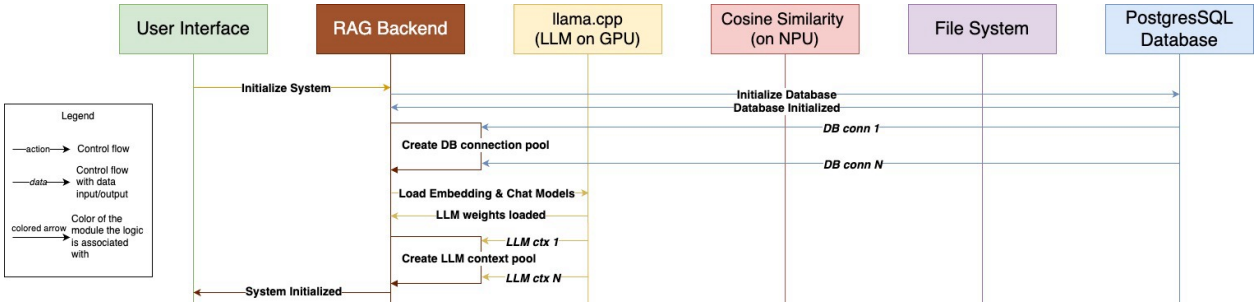


Figure 4.13: RAG Output evaluation metrics [6]

When a user specifies a directory to embed:

- The **RAG Backend** scans the specified directory for documents using the File System.
- Each document is loaded, chunked, and converted to embeddings using a pre-defined embedder.
- Embeddings, text chunks, and associated metadata are inserted into the **PostgreSQL Database**.
- In parallel, the backend also writes a vector dump file to the **File System**, which stores the hash of each vector for quick access.

This dual-storage mechanism (DB + mmap vector cache) allows fast retrieval during inference while maintaining queryable metadata.

Performing RAG

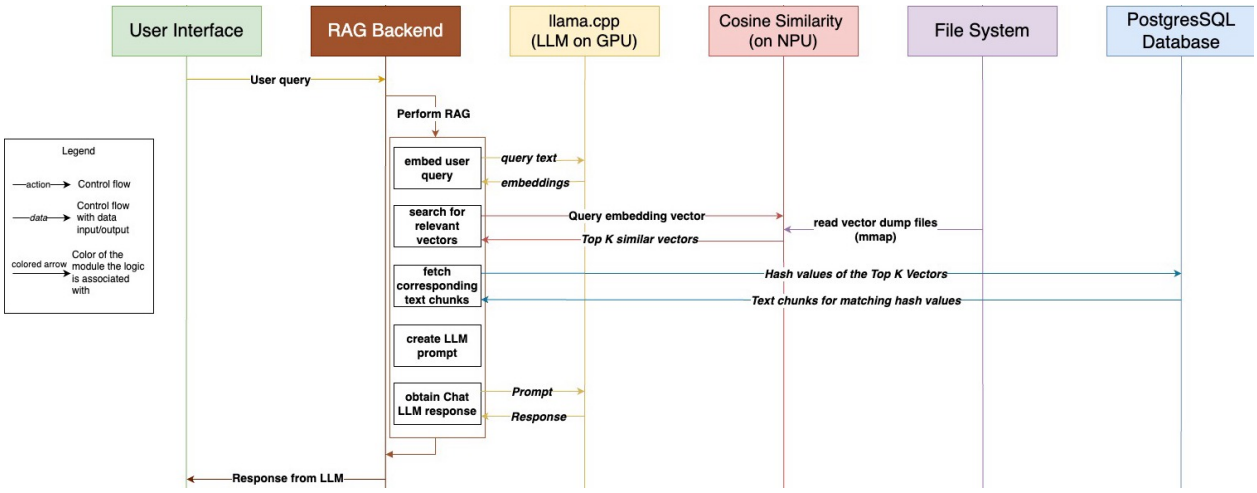


Figure 4.14: RAG Output evaluation metrics [6]

Once the corpus is embedded, the user may input a query. The following steps are executed:

- The user query is embedded using the same embedding model.
- The embedded query vector is sent to the **Cosine Similarity** module running on the NPU.
- A top-k similarity search is performed against memory-mapped vector files using the NPU, returning hash values of the best matches.
- These hashes are used to retrieve the corresponding text chunks from the **PostgreSQL Database**.
- A prompt is constructed and sent to the **LLM** via `llama.cpp`.
- The generated response is sent back to the **User Interface**.

This phase exemplifies the retrieval-augmented generation paradigm, grounded entirely in the user's local corpus and executed with optimized hardware utilization.

Chapter 5

EXPERIMENTATION-TODO

This chapter covers the experiment design and setup, describing in detail results obtained during different phases and tests. This chapter is organized as follows: first, we discuss the results of the training and its validation in Section 5.1, then in Section 5.2 and Section 5.3 we explore the DL model performance from the perspective of each component individually using the methods defined in Section **??**. Lastly, in Section 5.4, we analyze the performance of the DL model based on the two main evaluation metrics defined in Section **??** (MSE and execution time).

Results from experiments in this chapter are based on the dataset described in Section **??**. The experiments performed can be categorized into two main types based on the obstacle shape used:

1. Circular obstacle: the obstacle is randomly positioned in the simulation space, and its size is given by radius $\mathbf{r} = qH$, where $q \in [\frac{1}{9}, \frac{1}{5}]$ and H is the height of the simulated region.

2. Elliptical obstacle: the obstacle is also randomly positioned in the space, and its size is given by semi-major axis $\mathbf{a} = qH$, where $q \in [\frac{1}{5}, \frac{1}{3}]$ and H is the simulated region height, and the semi-minor axis $\mathbf{b} = p\mathbf{a}$, where $p \in [\frac{1}{5}, \frac{1}{4}]$. The ellipse is also tilted with an angle $\boldsymbol{\alpha} \in [-30°, 30°]$ with respect to $\mathbf{a}$ and the Cartesian x-axis.

The ranges of the obstacle dimensions and positions were chosen to fully fit the shape inside the simulated space without touching its perimeter. The dimensions of the Ellipse object were chosen to maintain a shape similar to that of an airfoil as described in Section **??**,

and its inclination range to represent common wing "angle of attack" including the critical or stalling "angle of attack" (typically between 18° and 25° degrees)[**?**].

5.1 Training Results

The neural network was trained for 500 epochs, across which, the Mean Squared Error (MSE) loss functions results were collected for both the Autoencoder and the Generator. The training error is captured in Both plots in Figure 5.1 show that the is generally lower than the validation error; this is to be expected given the proposed model was evaluated with sequences used during the training process.

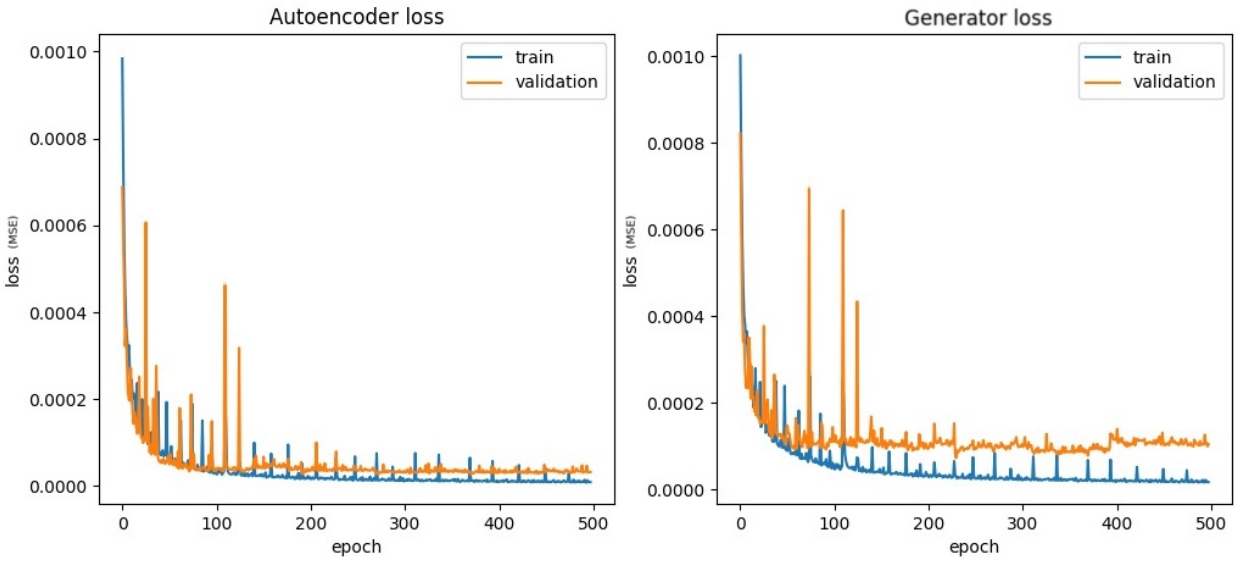


Figure 5.1: Autoencoder and Generator loss evolution during training

Figure 5.1 demonstrates the stability of the MSE loss function's evolution during training. The loss trend quickly descends, and after 200 epochs, the error stabilizes and converges to its final value. In addition, note that the slight difference between the training and validation errors signifies that the model does not excessively over-fit the training data. Regular error spikes appear during training, representing instances where the model encountered local

maximums before finding a local minimum. This happens when, during parameter optimization, a new combination of the network's weights is worse than the previous one, but then it improves again. This further underscores the model's reliability. These spikes decrease as the training advances, indicating a stable and reliable training process. Another critical aspect to note is that those spikes happen almost on the same epoch number and with a similar intensity for both of the model's components (Autoencoder and Generator), meaning that these components are working in collaboration to find the best result. This supports the election of the model architecture.

Table 5.1: Training and Testing errors (MSE)

	Autoencoder	Generator
Training	7.7727×10^{-06}	1.5429×10^{-05}
Validation	3.2173×10^{-05}	2.0414×10^{-05}

A comparison between training and validation errors is displayed in Table 5.1 for both the Autoencoder and Generator components of the model. The low error rate in all cases indicates the model's good performance. It is important to remember that validation samples were not used during training, which is noticeable in how lower the training errors are compared to the validation errors. Since the model has already "seen" the training samples to optimize its weights, it learns how to approximate the output based on those values.

In the following sections, we analyze the model's performance more deeply, examining each component's performance individually.

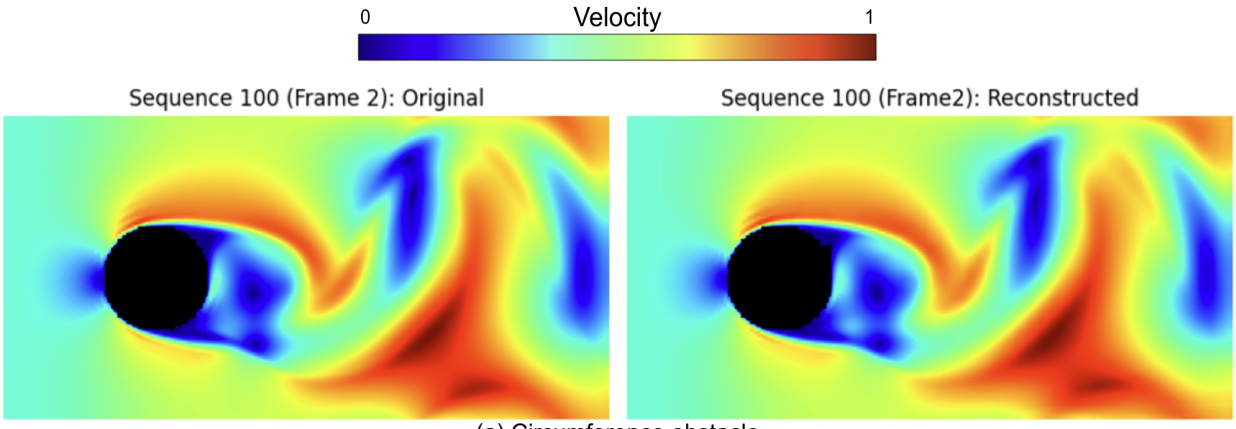
5.2 Autoencoder Results

To evaluate the Autoencoder, we need to verify that the Decoder can reconstruct the original information using the low-resolution representation of the data created by the Encoder. Because the input to the model is a set of frames according to the window described in

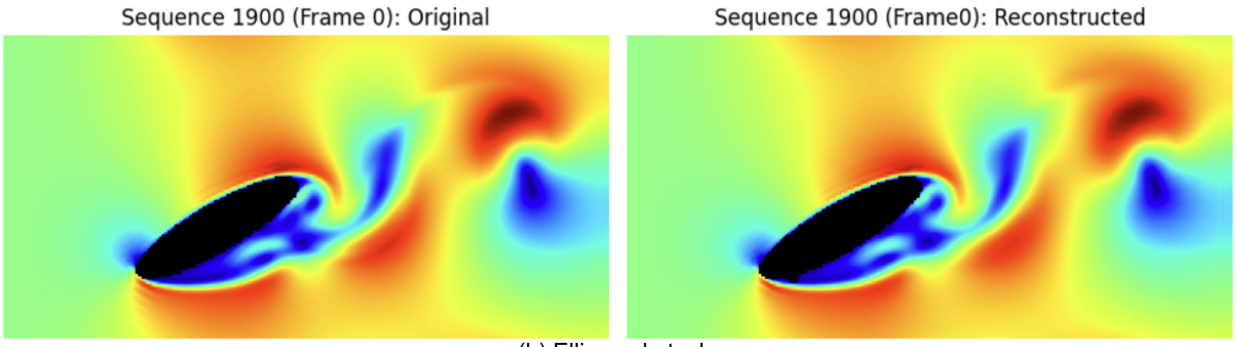
Chapter **??**, the Decoder output will also have those dimensions. This means that to evaluate the Decoder's output, we must compare all the frames in the set.

Using the Decoder's output sequence representing the fluid state, a *heatmap* of fluid velocity values was rendered to compare the results visually. Figure 5.2 shows examples of frames with each type of obstacle. In each case, we show the Original frame to the left and the Reconstructed frame output by the Decoder to the right. We can see that although there are some minor differences, both frames are almost identical. Similar results were obtained for the rest of the sequences. The differences found can be seen in small changes of intensity of the *heatmap* colors, which may indicate that the velocity values approximate the original ones but are either lower or higher than expected. Although there are some minor differences, the similarities between the Original and Reconstructed frames give us a clue that the Autoencoder is working as intended.

For the following evaluation method, we compare the velocity values between the Original and Reconstructed frames to verify their similarity. This is done by creating a histogram of velocities, i.e., a frequency count of velocity values on each grid cell in the frame. Figure 5.3 shows examples of those histograms for Origianl and Reconstructed frames containing each type of obstacle. The frame examples are the same as Figure 5.2 used in the previous evaluation method. On the x-axis, we have the range of all the velocity values in the frame. These values are between 0 and 1 because the data was previously normalized, as explained in Section **??**. On the y-axis, the frequency or occurrence of each velocity value is represented. To compare all the velocity histograms, we calculated the Jensen-Shannon distance between each frequency distribution. The resulting average distance between the original and reconstructed frames was 0.021, with a standard deviation of 0.014.



(a) Circumference obstacle



(b) Ellipse obstacle

Figure 5.2: Original vs Reconstructed frames

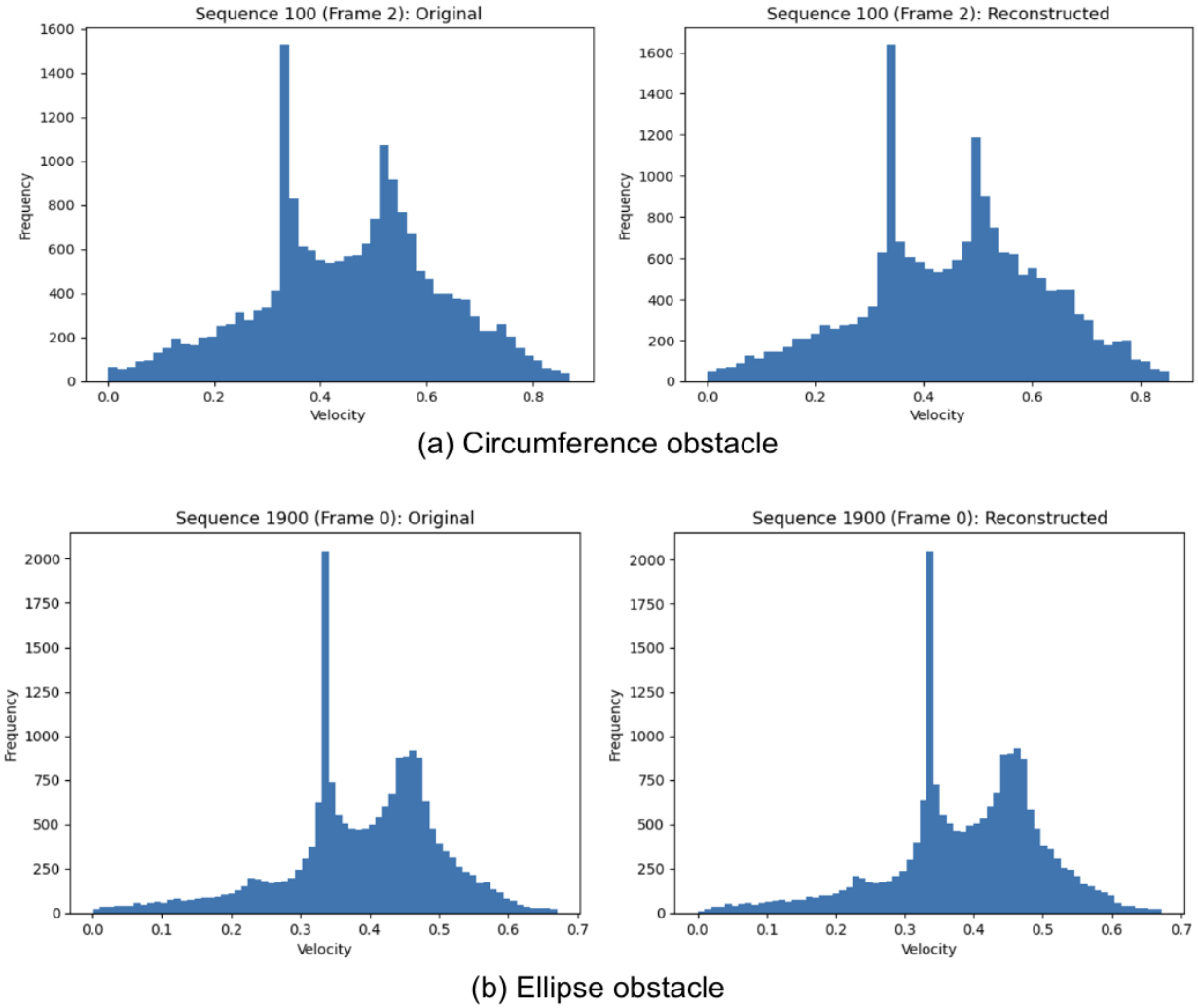


Figure 5.3: Original vs Reconstructed frames velocity histograms

Both evaluations tell us that the Autoencoder successfully reduces the data's dimensionality so that the original data can be reconstructed using that low-resolution representation. This means that the training of this model's component was successful. Although there are some minor errors, the fluid flow structure remains correct, and the approximations of the velocities are very close.

The Autoencoder is a vital component of the model because it guarantees that the model

successfully extracts enough information to represent the original data. This process of reducing the amount of information with a lower representation is important to support the next phase, which is the generation of the next fluid state.

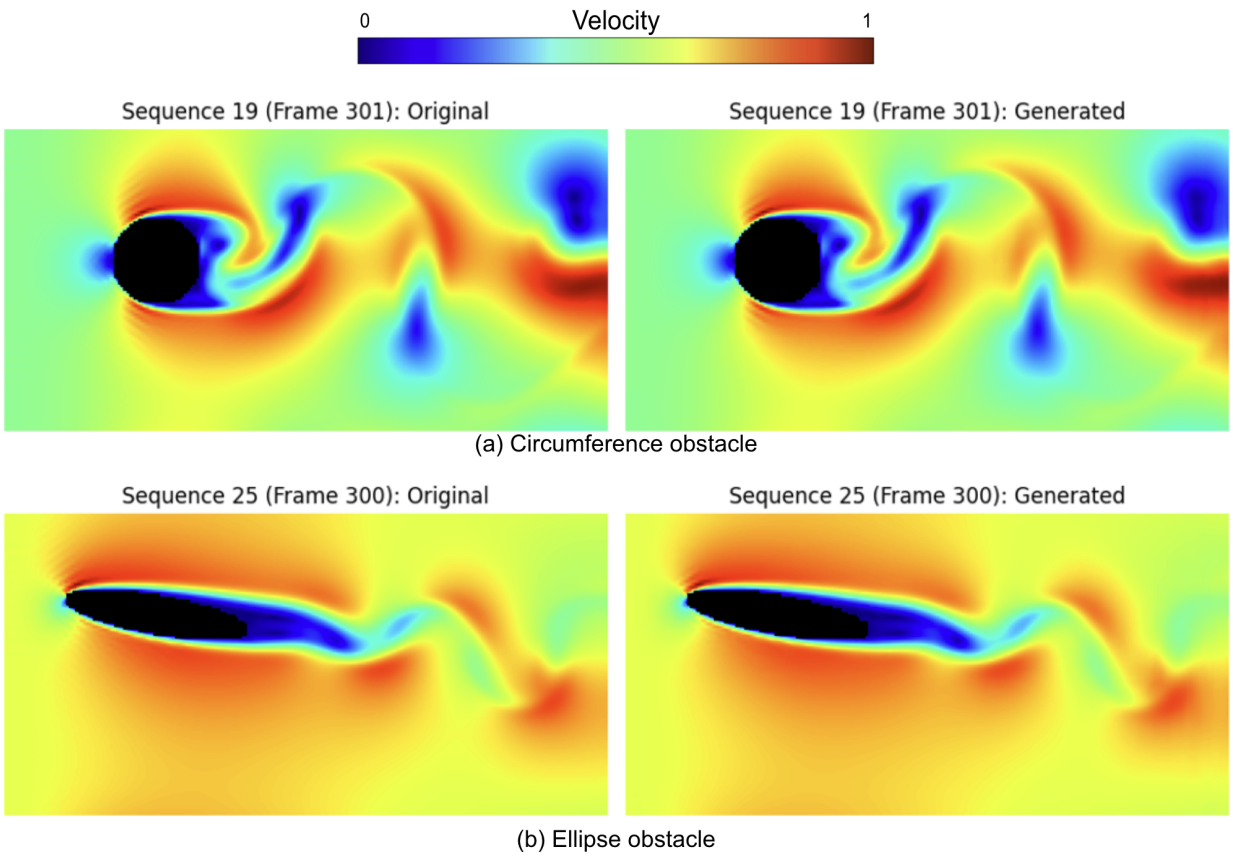
5.3 Generator Results

The Generator's goal is to generate the next fluid flow state using as an input the low-resolution representation created by the Encoder. To evaluate this component, the resulting frame is compared against the expected frame taken from the dataset created by the numerical simulation. For this evaluation we used similar methods than the Autoencoder evaluation.

Figure 5.4 shows a comparison between a generated frame velocity *heatmap* at the right, and the expected frame at the left. The images show a result example for each of the possible obstacle types. Similar to the Autoencoder results, very small differences appear in the *heatmap* colors, however, both images are almost similar.

We created the velocity histogram for each generated frame and compared it to the original frame. Figure 5.5 compares 2 examples of the velocity histograms. Then, we calculated the Jensen-Shannon distance between the velocity distributions of the frames in all the sequences. This results in an average distance between the expected and generated frames of 0.043 with a standard deviation of 0.010. The low average value of the distance metric between both distributions indicates that the original and generated frames are very similar.

These evaluations show that the model can successfully approximate the next state in the fluid flow sequence. The same level of accuracy in both evaluation methods was observed across all the cases in the testing dataset. Although there are some differences between the expected and generated frames, the model can replicate the evolution of the fluid flow structure across the sequence simulation time.



(a) Circumference obstacle

(b) Ellipse obstacle

Figure 5.4: Original vs Generated frames

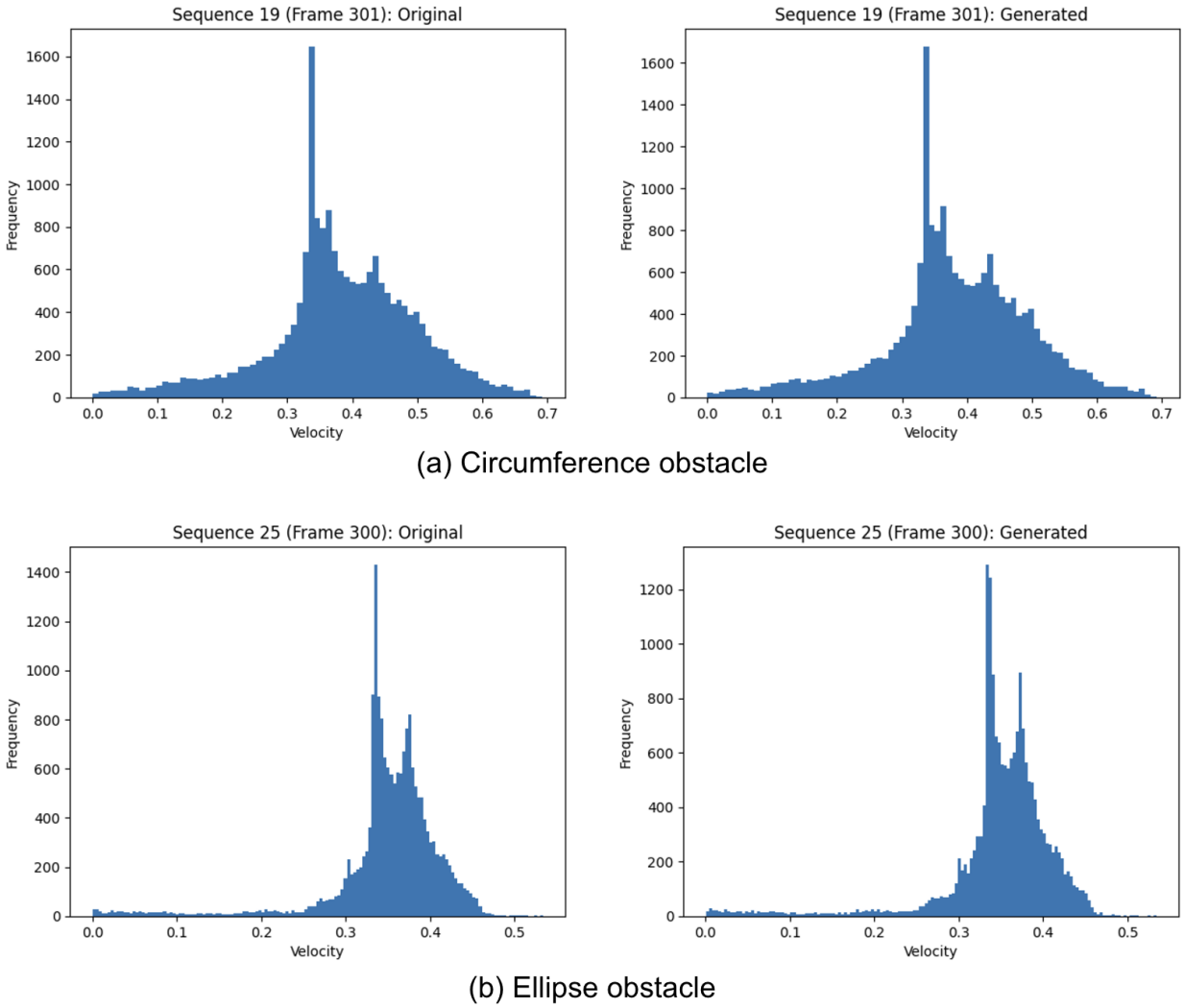


Figure 5.5: Original vs Generated frames velocity histograms

5.4 Model Performance

In this section we explore the results of the two main metrics chosen to evaluate this models performance as explained in Section **??**. These metrics are: the Error measured with MSE, and the execution time of the simulation measured in seconds.

5.4.1 Error Measurements

Figure 5.6 shows the results of the MSE metric. The results are divided into three groups, one for all the shapes in the dataset together, one with only Circumferences obstacles, and another for Ellipses obstacles. The minimum, maximum, and average error values are plotted for each group. It is important to mention that the dataset is balanced, meaning that the amount of examples with each obstacle type is the same, which is essential to ensure fairness in the results.

The following analysis can be done by looking at the MSE plots in Figure 5.6. We can see that the minimum and maximum errors across the entire dataset are both in simulations with an ellipse obstacle. Additionally, the difference between the minimum and maximum error is lower for the circumference than the ellipse obstacle. This could be caused by circumference obstacles presenting less variability in their shapes, with only a change in the radius, while ellipses obstacles have more diversity in their shapes. This variability in the obstacle shapes makes it more challenging for the model to learn how to simulate the ellipse objects. However, because the average errors are similar between the two types of obstacles, we can conclude that no specific obstacle shape is significantly more difficult for the model to simulate.

5.4.2 Execution time

As explained in Section **??**, the goal of this model is to reduce the execution time of the simulation while maintaining a low error to preserve the pattern structure of the fluid flow in the generated sequence. Table 5.2 compares execution time between the simulation and the

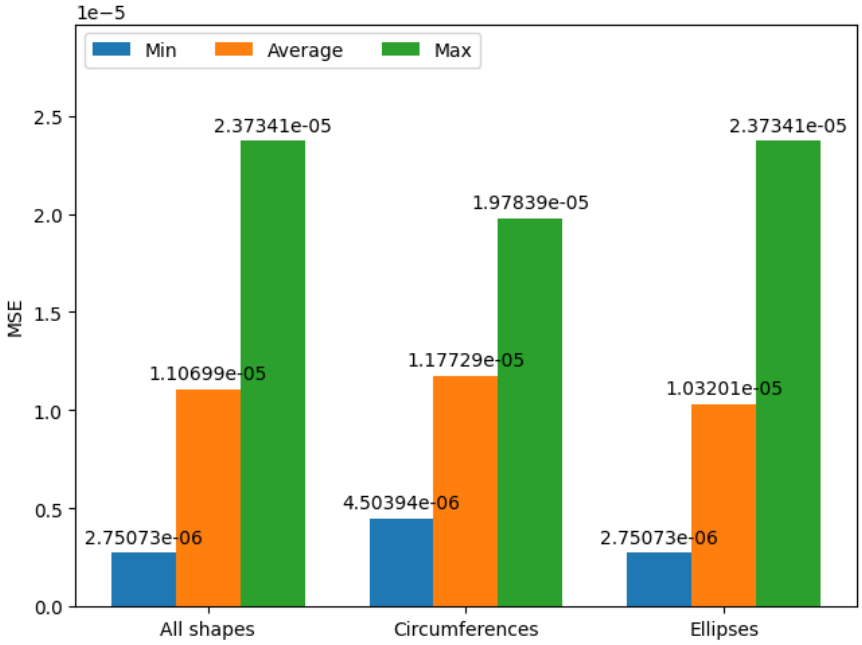


Figure 5.6: Model MSE error metric

DL Model. The simulation took, on average, 191 seconds (3.2 minutes), while the DL model took, on average, only 42 seconds. This represents a 4.5 times improvement in execution speed over the numerical simulation. This result shows that using this DL model improves the simulation's performance, reducing the total execution time while successfully simulating the evolution of the fluid flow.

Table 5.2: CFD simulation vs DL Model execution time

	Average Execution Time
CFD Simulation	191
DL Model	42

Chapter 6

CONCLUSION-TODO

6.1 Conclusion

In this research, we analyze the use of ConvLSTM to implement a deep-learning model for turbulent fluid flow simulations. We propose a neural network architecture to create a Reduced Order Model (ROM) and generate a fluid flow by predicting its spatiotemporal dynamics. This architecture is implemented completely using the ConvLSTM network for all its components. The model was trained and tested using a dataset of fluid flows interacting with circumference and ellipses obstacles of diverse sizes and positions in a two-dimensional space. The data was generated using a Direct Numerical Simulation (DNS). The performance was evaluated using the MSE error metric, achieving results comparable to a DNS simulation. When it was evaluated using the training and testing data, the model achieved similar error metrics, meaning that it was able to generalize well to unknown data. It is also worth mentioning that the error was balanced across the two types of obstacles. Additionally, the simulation execution time with the DL model was four times faster than the DNS. These results demonstrate that a neural network model proposed, combining an autoencoder and a generator implemented with a ConvLSTM, is suitable for predicting turbulent fluid flows and can accelerate Computational Fluid Dynamics simulations.

6.2 Contributions

6.2.1 Data preparation and training methods

The dataset we created for this research builds upon others used in CFD research by including a greater diversity of obstacles with different shapes and positions in the simulated space. Additionally, it takes into account a common and important shape in fluid dynamics, which

is the airfoil shape, by using a simple approximation of it.

The methods defined to preprocess the raw data of simulated fluid flows are essential to prepare the dataset to be used for the model's training process. This greatly influences the final model performance. The methods we describe in this study can be easily extended and adapted to other applied cases, like longer simulated sequences or different window $\mathcal{W}$ sizes. Additionally, this could be applied to any neural network training problem that relies on a multidimensional time-series dataset.

6.2.2 Model arquitecture

This research demonstrates the effectiveness of the ConvLSTM type of neural network in CFD by presenting a model architecture that is totally implemented with ConvLSTM and can achieve similar results to a DNS of fluid dynamics. This architecture was successful in other domains involving multivariate time series, like weather prediction, and we show how a solution in CFD can completely rely on this neural network.

The DL model presented further demonstrates how a data-driven approach with an end-to-end model can be used to generate fluid flow simulations in CFD applications. Additionally, the model architecture shows how useful is to include the creation of ROMs with an Autoencoder as a component of the DL model. The proposed model could represent the input data with half of the initial dimension and still reconstruct the original data. By including the dimensionality reduction as part of a unified model, the entire CFD process is simplified because it reduces the steps in the process to only one. Additionally, this research showed how much faster is to execute the neural network model compared with the DNS. By obtaining an improvement in the execution time of about 4 times faster, the research proves that this model can accelerate scientific computing simulations, including CFD.

6.3 Limitations

Some limitations exist in this work that are worth mentioning:

- Available hardware resources: the available hardware had 2 GPUs with 16GB of RAM each. This limits the size of the model and the dataset that could be loaded in memory. Because the model and the dataset have to share the GPU memory simultaneously, it limits the outcome of the model training. While training, the process used about 95% of the available memory on each GPU.

- Dataset: The dataset only has a single obstacle of two possible types with a simple shape. Furthermore, the shapes are static during the simulation. Additionally, all the fluid flows considered have the same Reynolds number.

- Comparing the results to other solutions: because the dataset was different than equivalent methods, there was no way to compare the results without implementing all the other solutions again.

6.4 Future work

This work opens plenty of opportunities to extend this research. These opportunities are related to scope limitations that exist in this work and possible improvements to this research. Future work could focus on extending the dataset to consider a wider range of shape types and complexity. Additionally, moving obstacles could be considered for the dataset. Including fluid flow sequences with multiple obstacles or complex obstacles that use simple ones as building blocks could also be interesting to test. Additional work could consider more complex turbulent flows with different Reynolds numbers and evaluate how well the model results generalize to those cases. In future research, it might be worthwhile to investigate how the window size m affects the accuracy and performance of the model; this could involve determining the point of diminishing returns when increasing the size of the window while giving the model a small amount of input information.

Computational Fluid Dynamics is already a mature field. It utilizes principles from fluid dynamics that are at least 300 years old and relies on computer simulation algorithms and

methods that are about 60 years old. However, recent research like this one shows that progress can still be made with the introduction of Deep Learning to the field.

BIBLIOGRAPHY

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30:5998–6008, 2017.

[2] Nabi. All you need to know about llm text generation, 2024.

[3] Hollemans. Apple m1 chip architecture, 2020.

[4] Glenn. Mastering Retrieval-Augmented Generation (RAG) Architecture. `https://blog.stackademic.com/mastering-retrieval-augmented-generation-rag-architecture-unleash-the-power-of-larg` 2024. Mastering Retrieval-Augmented Generation (RAG) Architecture.

[5] LangChain. Rag tutorial. `https://python.langchain.com/docs/tutorials/rag/`, 2023. Accessed May 2025.

[6] Erika Cardenas and Connor Shorten. An overview on rag evaluation, 2023. Accessed: 2025-05-22.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[9] OpenAI. Gpt-4 technical report. `https://openai.com/research/gpt-4`, 2023. Accessed 2024.

[10] Yang Liu and Mirella Lapata. Text summarization with pretrained encoders. *arXiv preprint arXiv:1908.08345*, 2019.

[11] Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282*, 2021.

[12] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[13] Rishi Bommasani et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[14] Philipp Mattern, Felix Sattler, Hartmut Schmeck, and Bastian Pfitzner. Membership inference attacks against language models via neighbourhood comparison. *arXiv preprint arXiv:2306.04554*, 2023.

[15] Milad Nasr, Nicholas Carlini, Florian Tramer, and Reza Shokri. Scalable extraction of training data from (production) language models. *arXiv preprint arXiv:2311.17035*, 2023.

[16] Amy Deschenes and Meg McMahon. A survey on student use of generative ai chatbots for academic research. *Evidence Based Library and Information Practice*, 19(2):2–22, 2024.

[17] Mohammad Hosseini, Serge Horbach, Tanja Van den Broek, Boudewijn De Bruin, and Sietse Wieringa. An exploratory survey about using chatgpt in education, healthcare, and research. *medRxiv*, 2023.

[18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and train-

ing of neural networks for efficient integer-arithmetic-only inference. *arXiv preprint arXiv:1712.05877*, 2017.

[19] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[20] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kulkarni, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[21] Apple Inc. Apple neural engine (ane) performance across devices, 2024. Accessed May 13, 2025. Reports ANE performance ranging from 0.6 TOPS (A11) to 38 TOPS (M4).

[22] Georgi Gerganov. LLaMa.cpp. `https://github.com/ggerganov/llama.cpp`, 2023. C++ framework to run open source LLMs on local hardware.

[23] Ollama. `https://ollama.com`, 2023. Desktop application to download and run a specific LLM.

[24] Tinygrad. Apple neural engine reverse engineered for c++. `https://github.com/geohot/tinygrad/tree/master/accel/ane`, 2023. GitHub repository documenting reverse engineering of Apple's ANE.

[25] LLamaFile. `https://github.com/Mozilla-Ocho/llamafile`, 2023. Command-line app to package and run an LLM.

[26] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Matthias Gallé, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[27] Jay Alammar. The illustrated transformer, 2018. Accessed May 2025.

[28] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016.

[29] Talamdupula. A guide to quantization in llms. 2024.

[30] Li. Quantization tech of llms-gguf. 2024.

[31] Reiner Pope, Alex Tamkin, Ekin Akyürek, Suhas Dathathri, Danny Hernandez, and Andrew Goldie. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022.

[32] Apple Corp. Apple m1 overview. Technical report, Apple Inc., 2020.

[33] Apple Corp. *Apple Metal GPU Developer Guide*. Apple Inc., 2020.

[34] Tinygrad. Apple neural engine reverse engineered for c++, 2023.

[35] Apple CoreML. Coreml – converting pytorch model to coreml. `https://developer.apple.com/documentation/coreml/`, 2024. Accessed May 2025.

[36] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[37] Bashir. In-context learning, in context. 2023.

[38] LangChain. Text splitters. `https://python.langchain.com/docs/modules/data_connection/document_transformers/text_splitter/`, 2023. Accessed May 2025.

[39] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 3: Open foundation and instruction-tuned models. `https://ai.meta.com/blog/meta-llama-3/`, 2024. Meta AI, Accessed May 2025.

[40] Nelson F. Liu, Tianyi Shen, Faeze Brahman, Mona Diab, Noah A. Smith, and Yejin Choi. Long in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023.

[41] Wenhui Wang, Furu Wei, Li Dong, Hang Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *arXiv preprint arXiv:2002.10957*, 2020.

[42] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[43] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[44] Xiaohua Wang and Zhenghua Wang. Searching for best practices in retrieval-augmented generation. *arXiv preprint arXiv:2407.01219*, 2024.

[45] Shinya Sugawara, Hayato Kobayashi, and Manabu Iwasaki. On approximately searching for similar word embeddings. *arXiv preprint arXiv:1604.00417*, 2016.

[46] Harald Steck, Hieu Pham, Dawen Ding, Lam Thai, Hu Yue, Wei Han, Jeanne Soar, Mihir Sahasrabudhe, and Cosma Shalizi. Is cosine-similarity of embeddings really about similarity? *arXiv preprint arXiv:2403.05440*, 2024.

[47] Labelbox. Vector similarity search techniques. `https://labelbox.com/blog/how-vector-similarity-search-works/`, 2023.

[48] Felix L. Hsu. pgvector: Vector similarity search for PostgreSQL. `https://github.com/pgvector/pgvector`, 2023. Accessed: 2025-05-23.

[49] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. *arXiv preprint arXiv:1908.10396*, 2020.

[50] LangChain Contributors. Langchain: Building applications with llms through composability, 2023. https://www.langchain.com/.

[51] LlamaIndex Team. Llamaindex (gpt index), 2023. https://www.llamaindex.ai/.

[52] deepset AI. Haystack: Open source nlp framework for question answering, summarization, and more, 2023. https://haystack.deepset.ai/.

[53] Microsoft Research. Autogen: Enabling next-gen llm applications via multi-agent collaboration, 2023. https://github.com/microsoft/autogen.

[54] Raphaël Gor, Xiao Tan, Zi Ouyang, Zekun Zhang, Jinjie Wei, Yiheng He, Lei Xu, and Yuexin Wu. Routerllm: An expert routing system for cost-efficient inference of large language models. *arXiv preprint arXiv:2309.13285*, 2023.

[55] RAGAS Contributors. Ragas: Retrieval-augmented generation assessment, 2024. https://github.com/explodinggradients/ragas.

[56] Gerganov. Llama.cpp project. https://github.com/ggml-org/llama.cpp, 2020.

[57] Ollama. Ollama. https://ollama.com/, 2023. Accessed May 2025.

[58] Mozilla-Ocho. Llamafile. https://github.com/Mozilla-Ocho/llamafile, 2023. Accessed May 2025.

[59] ggml org. llama.cpp - simple example. https://github.com/ggml-org/llama.cpp/tree/master/examples/simple, 2024. Accessed 2025-05-26.

[60] ggml org. llama.cpp - server example. https://github.com/ggml-org/llama.cpp/tree/master/tools/server, 2024. Accessed 2025-05-26.

[61] ggml org. llama.cpp - embedding example. https://github.com/ggml-org/llama.cpp/tree/master/examples/embedding, 2024. Accessed 2025-05-26.

[62] ggml org. llama.cpp. https://github.com/ggml-org/llama.cpp, 2023. GitHub repository. Accessed May 2025.