Cognify: An On-Device, AI-powered Learning Assistant

by

Siyong Huang

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2025

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2025

Authored by:	Siyong Huang Department of Electrical Engineering and Computer Science August 22, 2025
Certified by:	Song Han Associate Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Cognify: An On-Device, AI-powered Learning Assistant

by

Siyong Huang

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Large Language Models (LLMs) have proven highly effective for a wide range of natural language processing tasks, but their size and compute requirements often restrict their use to powerful cloud-based infrastructures. In recent years, significant progress has been made in shrinking LLMs while maintaining performance levels comparable to much larger models. We are approaching the point where the capabilities of massive, multi-billion parameter models can be realistically replicated on consumer-grade devices. This thesis builds upon that foundation by developing an AI-powered note-taking application that runs entirely offline, using only the compute resources available on a personal laptop. The application is designed to listen to lectures alongside the student and provide support in real-time—through transcription, notes generation, and enabling context-aware search. Achieving this level of interactivity locally introduces challenges in reducing end-to-end latency, which this project addresses through both model-level optimizations and the design of efficient prompting and inference algorithms. A demo of the app can be found on Youtube[1].

Thesis supervisor: Song Han
Title: Associate Professor of Electrical Engineering and Computer Science

[1] https://youtu.be/mQAhfAF1wr4

Acknowledgments

I want to express my gratitude to Professor Song Han for his exceptional mentorship and vision throughout this project. His expertise in efficient deep learning and model compression shaped the technical direction of this thesis but also inspired me to think deeply about building AI systems that are powerfully practical.

I also want to thank my mentor Guangxuan Xiao for his exceptional guidance and patience throughout this journey. Thank you for all the trust and encouragement that made this project so enjoyable, and for always being available to help and advise me.

Thank you to my collaborator Eric Ge for your invaluable contributions in this project. Your talent and passion for science and engineering have set a strong foundation for everything you'll achieve.

My friends—whether from MIT, back home, or scattered across the country—have been a constant source of support and inspiration. Their encouragement kept me motivated and grounded throughout the journey of writing my thesis.

I owe my deepest gratitude to my family for your unconditional love and support. Your encouragement and faith in my abilities have shaped who I am, motivating me to push myself and achieve beyond what I thought was possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The emergence of large language models (LLMs) has fundamentally transformed how users interact with software, enabling capabilities such as search, writing assistance, coding, transcription, and summarization. These features have been rapidly integrated into both consumer and enterprise applications.

Most user-facing applications that incorporate LLMs rely on cloud-based infrastructures to handle computation. Web applications such as ChatGPT and Claude perform all inference on the provider's servers, transmitting user input remotely and streaming responses back. Similarly, desktop applications such as Cursor (an AI-powered code editor) and Notion (an AI-enhanced productivity suite) typically offload computation via API calls to cloud-hosted LLMs offered by providers such as OpenAI, Anthropic, or Google.

This architecture enables developers to leverage state-of-the-art models without requiring significant local resources, but it entails trade-offs in latency, privacy, cost, and dependence on internet connectivity. In contrast, this thesis investigates the feasibility of performing all LLM inference locally on consumer-grade hardware, with no external dependencies.

For users in environments with unreliable internet access or privacy requirements, such as classrooms, legal practices, or healthcare settings, cloud dependence can be a barrier to adoption. Even with stable connectivity, remote compute resources may incur high costs or

be temporarily unavailable.

Recent advances in model compression and inference optimization have begun to make it possible to deploy LLMs with substantially reduced compute requirements. Open-source projects such as Llama.cpp and Whisper.cpp illustrate that reasonably capable language and speech models can now run on consumer-grade devices.

This thesis aims to bridge that gap by developing a real-time, AI-powered note-taking assistant that operates entirely offline. Designed for students attending lectures, the system listens passively, transcribes speech, highlights key points, generates summaries, and enables context-aware search—all on the user's personal laptop. In doing so, this project explores the intersection of model compression, efficient prompting, and user-centric software design, offering a blueprint for AI tools that are private, portable, and responsive.

Chapter 2

Background and Related Work

2.1 Inference Optimization

To enable efficient on-device deployment of large language models, recent research has focused on inference optimization techniques. Methods such as quantization and speculative decoding reduce computational and memory requirements while maintaining model performance, forming a foundation for offline AI applications.

2.1.1 Quantization

Quantization is a technique that reduces the precision of model weights and activations to improve inference speed and decrease memory usage. It has proven to be highly effective, reducing memory footprints by up to 8× compared to full-precision models and achieving comparable speedups [1, 2].

While quantization can introduce accuracy losses, these errors may accumulate across the layers of a model. To address this, researchers have developed methods to mitigate its impact, such as carefully rounding model weights [3] or by smoothening out large activations into the weights [4].

Quantization has demonstrated effectiveness across various model architectures and has

been an active area of research in both convolutional neural networks (CNNs) [5] and transformers [6–9].

This work leverages quantization techniques to achieve the performance of more capable models while remaining within memory and computational constraints.

2.1.2 Speculative Decoding

Different model sizes present inherent trade-offs: larger models generally produce more accurate and contextually rich responses, but at the cost of slower generation speeds. Speculative decoding leverages the speed of smaller models by allowing them to generate candidate tokens, which are then verified by a larger model.

Because token verification is faster than token generation, this approach can reduce inference time when the smaller model predicts the correct tokens with high probability [10, 11] . In such cases, the larger, slower model generates fewer tokens overall, improving efficiency without sacrificing output quality.

2.2 Existing Infrastructure for On-Device AI

Building on inference optimization techniques, several software frameworks and tools have emerged to enable efficient on-device deployment of large language models. Notable examples include Llama.cpp and Whisper.cpp, which provide lightweight inference engines, as well as Ollama and LlamaIndex, which extend these capabilities with APIs and frameworks for embedding, retrieval, and knowledge management.

2.2.1 Llama.cpp and Whisper.cpp

Llama.cpp and Whisper.cpp [12, 13] are C++ based inference engines designed for efficient on-device deployment of large models. Llama.cpp supports a variety of LLM architectures, whereas Whisper.cpp is tailored specifically for OpenAI's Whisper speech recognition model.

[14].

Both inference engines operate with the GGUF and GGML file formats and include scripts to automatically convert between GGUF/GGML and Hugging Face models. These inference engines implement GPU support across multiple platforms and environment, including CUDA, Apple Metal, and others, allowing them to fully utilize the compute resources available on a user's local device.

Llama.cpp supports multiple quanitzation schemes, including AWQ [3]. However, it can convert Hugging Face models only into standard `Q4_0`, `Q4_1`, `Q8_0`, etc. formats as well as GGUF-specific K-quants. To produce an AWQ-quantized model in GGUF format, the model must first be tuned in FP16, and then converted into `Q4_1` format.

2.2.2 Ollama

Ollama is a wrapper around Llama.cpp that extends its capabilities and streamlines the user experience. It provides direct API support for tasks such as creating embedding models, performing function calls, and integrating with frameworks like LangChain [15].

2.2.3 LlamaIndex

LlamaIndex is a flexible framework for building knowledge assistants by LLMs. It supports Ollama as an embedding model provider, and enables a variety of search and retrieval tasks.

2.3 Existing Note-Taking Tools

Digital note-taking tools play a critical role in learning, research, and professional productivity. Both traditional and AI-enhanced tools have emerged to support note organization, retrieval, and summarization.

2.3.1 Traditional Note-Taking Tools

Popular tools such as Microsoft OneNote, Evernote, Notion, and Obsidian provide flexible digital Notebooks with multimedia support. Notion extends this functionality with databases, cross-referencing, and highly customizable organization, while Obsidian emphasizes local storage and networked note-taking using markdown-based workflows. Although these tools excel at structuring and linking content, they generally do not offer advanced AI-driven summarization, keyword extraction, or real-time transcription.

2.3.2 AI-Powered Note-Taking Tools

Recent AI-powered solutions, including Otter.ai, Mem, and Notion AI, integrate machine learning to enhance the note-taking experience. Otter.ai provides real-time transcription and audio indexing for lectures and meetings, but relies on cloud servers, which can introduce latency and privacy concerns. Mem leverages AI to summarize notes and surface relevant content, and Notion AI offers LLM-based content generation and summarization within the Notion workspace. Despite these capabilities, most AI-driven tools require internet connectivity and do not operate fully offline, limiting usability in privacy-sensitive or connectivity-constrained environments. Across both traditional and AI-enhanced tools, a common limitation is that AI capabilities are minimal or heavily dependent on cloud based infrastructure. This gap motivates the development of the offline, real-time, AI-powered note taking assistant presented in this thesis.

2.4 Streaming Techniques

Recent work in streaming summarization has explored methods for processing long sequences of text with large language models (LLMs) in real time [16–18]. These approaches aim to overcome the computational and memory challenges that arise when processing lengthy

inputs, such as lectures or meetings, without sacrificing model performance.

2.4.1 StreamingLLM

Large language models have a context length that limits the maximum number of tokens they can process at once, typically ranging from 8,000 to 128,000 tokens. Increasing the context length, however, results in a quadratic slowdown in inference time. For tasks such as lecture transcription or long-form summarization, where inputs can span multiple hours, it is essential to perform inference efficiently, independent of input length.

To address these limitations, StreamingLLM truncates input tokens and employs attention sinks to manage context, enabling the model to process arbitrarily long sequences without exceeding memory or computational constraints [19].

Chapter 3

Cognifiy: An Offline, Real-Time AI-Powered Note-Taking Assistant

3.1 Design Principles

Most existing AI applications offload computation to external cloud provider, whether by querying APIs from foundation model companies like OpenAI or Anthropic, or accessing remote GPU clusters. Cognify, in contrast, is designed to run entirely on the user's device, which motivates three core design principles: real time performance, efficiency and precision.

Real-time performance

Cognify is intended to operate in real-time, keeping pace with fast-moving lectures without falling behind. Because the system is envisioned primarily for classroom use, it must respond promptly to current topics of discussion and remain aligned with the student's experience. If a student falls behind, Cognify needs to be able to catch the student up to the pace of the lecture. Substantial delays would create a mismatch between Cognifiy's notes and the ongoing lecture, reducing the utility of the system.

Efficiency

Efficient in Coginigfy refers to minimzing CPU and GPU usage, as well as energy consumption. Students often use the application in environments without a power supply, so inference must be lightweight and unobtrusive. This includes minimizing the number of tokens sent to and generated from the models. While techniques such as parallelization and batching can reduce latency and support real-time performance, it is equally important that inference does not overwhelm the device, ensuring a smooth and pleasant user experience.

Precise

Precision encompasses both the accuracy and conciseness of the model's output. Cognifiy's LLMs must produce responses that are correct and useful, but they should also use words sparingly. Excessive or inaccurate information can confuse students, whereas concise, accurate notes help them understand the material efficiently. Providing inadequate information can be mitigated by the student consulting the lecture directly, but incorrect information risks leading them astray.

All three principles—real-time performance, efficiency, and precision—share the common goal of enhancing the student's learning experience. Throughout Cognifiy's development, trade-offs among these principles inform design and implementation decisions, ensuring that the system remains both effective and practical in a classroom setting.

3.2 System Overview

The Cognify desktop application is built using ElectronJS, a widely adopted framework that provides cross-platform compatibility and a rich ecosystem of libraries and integrations, allowing for rapid development and iteration. The majority of the application codebase is written in TypeScript, with the front end leveraging React for rendering the user interface. The application is organized into three core processes:

1. **Main Process.** This process handles system-level operations, including accessing the file system, executing terminal commands, and utilizing available GPUs. It is also responsible for managing the renderer process.

2. **Renderer Process.** As the user-facing component of the application, the renderer process handles all frontend rendering and interaction. The bulk of the CPU-bound computational tasks are implemented here as well.

3. **Preload Process.** Acting as a bridge between the main and renderer processes, the preload process ensures secure communication and controlled access to desktop resources.

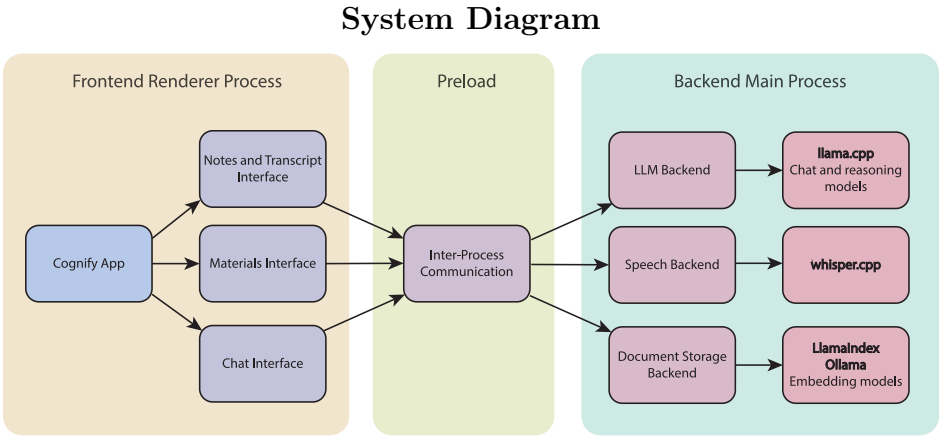

Figure 3.1: The renderer runs Cognify's user interface, which consists of the notes and transcript, materials, and chat pages. These communicate with the backend through the inter-process communication bridge set up by the Preload process. The backend accesses the device's GPUs to runs LLM, speech recognition, and embedding models.

This architecture separates concerns between system-level operations, user interface rendering, and secure communication, providing a robust foundation for Cognify's offline, real-time AI functionality. The three UI pages are depicted in figures 3.2, 3.3, and 3.4.

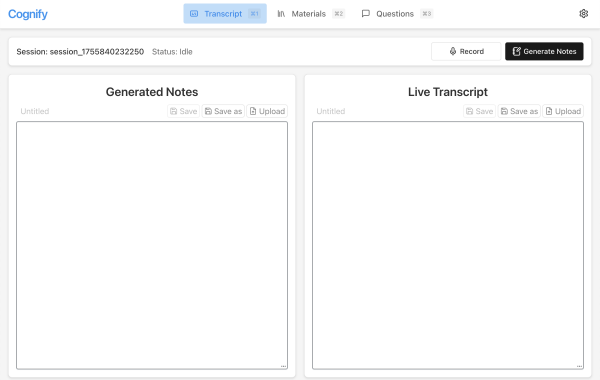

Figure 3.2: UI of the Notes and Transcript Interface

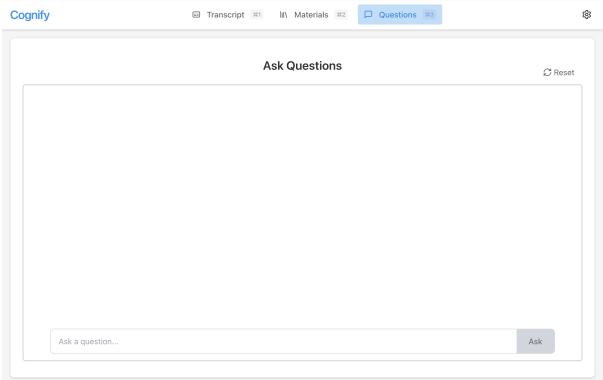

Figure 3.3: UI of the Chat Interface

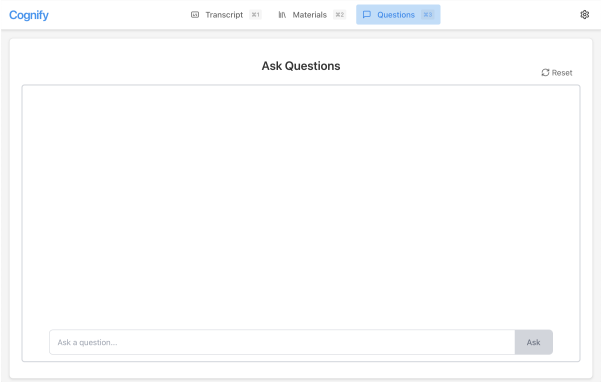

Figure 3.4: UI of the Materials Interface

3.3 Features

3.3.1 Real-Time Live Transcription

Cognify passively listens alongside students during lectures and transcribes speech in real time. Users can select any audio input device through the settings menu and begin recording by clicking the **Record** button, which streams audio directly into the **Transcript** panel.

Audio transcription is performed using the whisper.cpp inference engine. Since Whisper was originally trained on clips of up to 30 seconds, streaming requires dividing the audio into smaller chunks and processing them sequentially. Consecutive chunks are designed to overlap slightly to prevent words at chunk boundaries from being skipped. This overlap can introduce

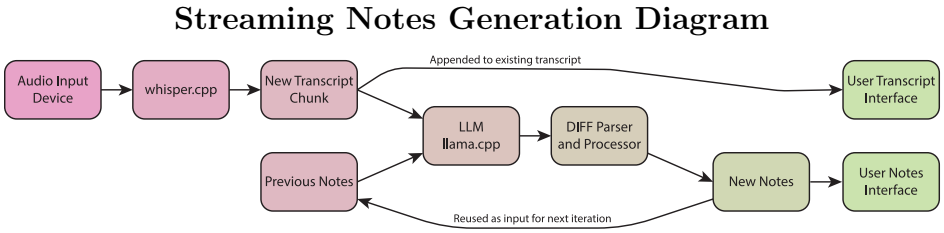

Figure 3.5: The audio device streams the transcript in using whisper.cpp, which is then buffered to produce transcript chunks of around 256 characters (3-5 transcript lines). A LLM then produces modifications to the old notes based on information in the new transcript chunk. We parse and apply these modifications to produce the new notes, and these new notes are displayed to the user and then fed back into the streaming system.

repeated words or phrases, which are removed using a simple deduplication algorithm that eliminates duplicates at the start of a chunk if they match the end of the previous chunk exactly.

To maintain real-time performance, Cognify uses a chunk size of three seconds with 200 milliseconds of overlap between consecutive chunks. This configuration strikes a balance between transcription quality and latency while naturally segmenting the transcript into lines of approximately 5-10 words.

The transcript is updated live in the right-hand panel. Users can edit and save transcripts to their local filesystem for later reference, as well as upload existing transcripts to facilitate study and review.

3.3.2 Automatic Streaming Notes Generation

Another key feature of Cognify is automatic notes generation. After a lecture, students may want to review material without reading the entire transcript or navigating long recordings. To streamline this process, Cognify produces structured and concise notes, allowing students to quickly locate relevant information and continue studying efficiently.

Lectures can span multiple hours, generating transcripts with tens of thousands of words. Feeding an entire transcript into an LLM is impractical, especially for smaller models used in Cognify, as this would exceed context length limits. To address this, transcripts are split into

smaller chunks, and notes are generated incrementally to capture all relevant information from each chunk.

This chunked approach also enables real-time note generation, keeping notes aligned with the ongoing lecture. Students can access summaries of material covered so far, facilitating connections between topics and enhancing comprehension.

To generate notes from transcript chunks, Cognify maintains an evolving notes set. Notes for the first chunk are generated normally, as if it were the entire transcript. For subsequent chunks, an engineered system prompt instructs the LLM to update existing notes by integrating information from both the previous notes and the new transcript chunk.

Chunk boundaries can pose challenges. If a lecturer is interrupted mid-idea, the model may struggle to summarize the content accurately. To mitigate this, Cognify includes a short history of preceding transcript chunks in the prompt, providing the LLM with sufficient context to capture ideas spanning multiple chunks.

As the transcript grows, both prefill and generation token requirements increase roughly linearly. To improve efficiency, Cognify instructs the model to output only differences between existing and new notes, rather than regenerating the entire note set. The DIFF output follows a structured format:

```
ADD AFTER LINE X: [New content to add after line X]
MODIFY LINE X: [New version of line X]
DELETE LINE X:
```

This DIFF-based approach encourages smaller, more precise updates. Previous attempts to regenerate the entire notes set often resulted in dropped bullet points or sections, as the model struggled to reconcile the small size of the new transcript chunk with the existing comprehensive notes.

While the DIFF method improves incremental updates, it introduced new challenges. The model sometimes produced duplicate bullet points, likely attempting to move lines without

issuing the corresponding delete commands. Additionally, the generated notes tended to follow repetitive patterns, with shallow sections consisting of 1–2 bullet points per keyword, making them difficult to skim.

To address these macro-level issues, Cognify employs an agentic approach. After the first LLM proposes updates, a second model reviews and restructures the notes, removing redundancy and organizing content into hierarchical chapters, sections, and subsections. This two-step process significantly improves the quality, readability, and usefulness of generated lecture notes.

3.3.3 RAG-Powered Chat Interface

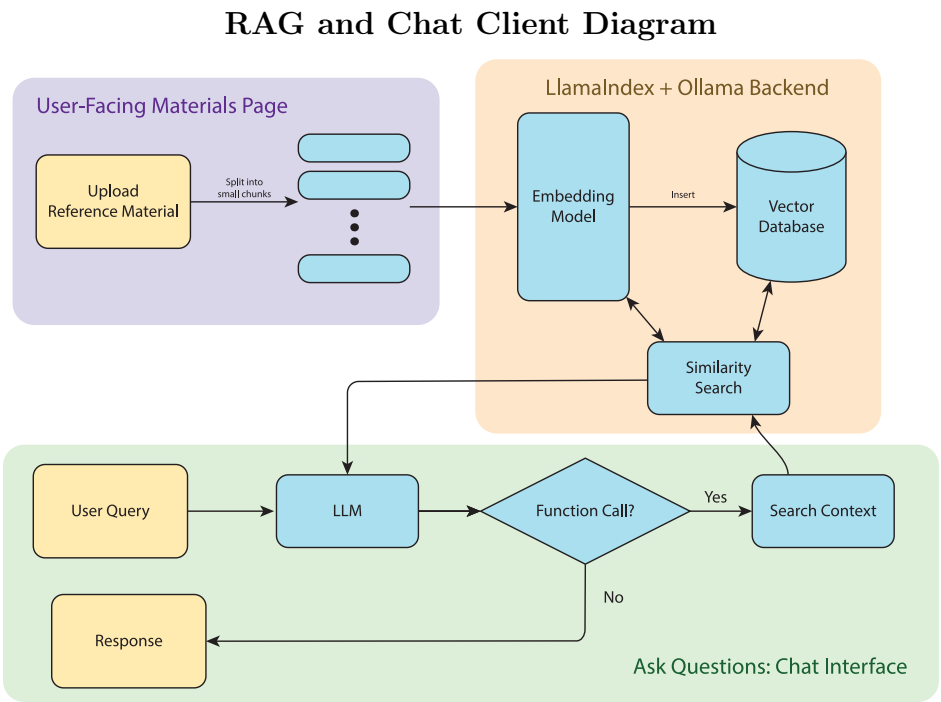

Figure 3.6: The user uploads materials in the Materials page, which gets split into small chunks that are embed and stored in the vector database. The backend consists of an embedding model, and a vector database powered by LlamaIndex. When the user asks a question, the LLM decides if it needs more context. If it does, we use a similarity search algorithm to find the most relevant documents and their most important chunks, and then feed it back into the model as context.

Cognify also features a chat interface that helps students clarify lecture material by

answering questions. Because the LLMs powering Cognify are relatively small, they cannot reliably generate accurate answers across a wide range of topics. To improve output quality and reduce hallucinations, students can upload lecture notes, slides, and other relevant documents that the model can reference as context.

To enhance performance in knowledge-intensive tasks, Cognify leverages Retrieval-Augmented Generation (RAG) [20]. RAG supplements the model with information retrieved from a document database, expanding its effective knowledge beyond what the model can memorize. This is particularly important for smaller models, which are constrained by size and context length.

Document retrieval is implemented using an encoder to generate embeddings for each document. These embeddings are stored in a vector database. When a query is issued, the query is embedded, and a top-k similarity search retrieves the most relevant documents for the model to reference.

In Cognify, the document database and retrieval mechanism are implemented using **LlamaIndex**, with embeddings generated via **Ollama**. The database is stored locally on the user's computer and persists across sessions, allowing the database to grow over time and provide increasingly informative context. Since documents can be large and LLMs have limited context lengths, documents are split into smaller chunks before storage in the vector database.

Although RAG can introduce latency to the Q&A experience, it is not always necessary for answering simple questions. To optimize efficiency, Cognify allows the model to assess question difficulty and determine when additional context is required by implementing function calling [21]. Function calling enables the model to access external computational tools or retrieve additional information dynamically. In Cognify, a function called `searchContext` performs document retrieval and provides the model with relevant context to answer user queries accurately. This approach significantly improves performance and reduces hallucinations.

A challenge arises when documents are split into many chunks with similar embeddings,

causing searches to retrieve multiple chunks from a single document, even if other documents contain relevant information. This is especially problematic for small models with limited context lengths. Cognify addresses this by storing summaries of large documents in the vector store. After selecting the most relevant summaries, a second retrieval is performed within the original chunks of each document to select the most informative portions.

This two-stage retrieval strategy provides two main advantages: (1) it drastically reduces the size of the retrieved context, enabling the model to incorporate information from multiple sources, and (2) it preserves essential details from each document, allowing the model to produce precise and accurate answers despite the constraints of a small model.

3.4 Challenges

3.4.1 Custom Javascript Bindings via WebAssembly

TypeScript is a wrapper around Javascript, and thus is an interpreted language that can struggle with performance during computationally intensive tasks. One approach to improve performance is to create bindings in a compiled language such as C++ or Rust and compile them to WebAssembly (WASM). WebAssembly is a low-level binary instruction format designed to integrate seamlessly with JavaScript-based web applications, providing near-native performance without relying on complex inter-process communication.

A key advantage of WASM is its universal support across modern web browsers. Unlike conventional compiled binaries, which often require system-specific recompilation and shared library dependencies, WASM modules execute reliably across platforms while achieving performance close to native binaries.

C++ WASM bindings are typically built using Emscripten, a complete LLVM-based compiler toolchain that converts C++ code directly into WASM modules. These modules can be imported into JavaScript or TypeScript applications, exposing JavaScript functions bound to the compiled C++ functions. Emscripten also generates type definitions to facilitate

smooth TypeScript integration.

For this project, we developed C++ bindings for the Whisper.cpp and Llama.cpp modules. While these bindings functioned correctly and enabled inference for extremely small models, we encountered significant limitations when running larger models or longer context lengths. Currently, WebAssembly only supports 32-bit pointers, restricting memory usage to 4 GB—insufficient for many practical LLM workloads. Although 64-bit pointer support is under development, for this project we executed Whisper.cpp and Llama.cpp through their original implementations with minimal modifications. Consequently, most inference optimizations were implemented at the application level rather than through WASM bindings.

Chapter 4

Experiments and Evaluation

4.1 Dataset

Our dataset consists of 20 MIT lectures uploaded to YouTube. We scraped their raw audio data into `.wav` files and download the Youtube-provided auto-generated captions. We then cleaned the caption data by removing annotations, and created audio-text pairs for each lecture.

The dataset should be as faithful as possible to the intended purpose of the application. As such, the data comes from recordings of lectures that were actually taught at MIT. Lectures in the dataset can be anywhere from 30 minutes to 2 hours long, because it's important that the algorithms we developed work in these real-life scenarios. Since the lectures can be so long, their audio files can grow to up to nearly 1GB. This limits the dataset to include only a handful of lectures. We randomly choose a small sample of 20 lectures across the MIT OCW lecture space, and then modify the dataset to cover as wide a breadth of topics and difficulties as possible.

4.2 Evaluations

We evaluate Cognify in two environments.

1. We measure the theoretical performance of models used in the App
2. We measure the actual performance of Cognify in an environment resembling that of a student in a lecture

We achieved environment (2) by running experiments on Cognify while it is open on a 2021 Macbook Pro, complete with an Apple M1 Pro chip and 16GB Unified Memory. While the M1 Pro chip is excellent and incredibly performant, newer models with more sophisticated chips become increasingly capable every year. Perhaps the tradeoffs and limitations we face today will be alleviated by future advances in hardware.

By understanding the capabilities of small-scale models, we can evaluate tradeoffs between Cognify's efficiency and quality.

4.2.1 Transcription Quality and Latency

Metrics

Transcription quality can be measured via the Word Error Rate (WER), Match Error Rate (MER), and theh related Word Information Preserved (WIP) and Word Information Lost (WIL) metrics [22].

The WER metric is defined by the equation

$$\text{WER} = \frac{S + I + D}{N}$$

where S, I, and D are defined as the number of substitutions, insertions, and deletions in the minimum edit distance between the reference and predicted transcripts. N is the number of words in the reference transcript. This metric represents essentially how much many words a user would need to manually correct in the generated transcript to make it match. This is the most common evaluation method, and has been used many times before. We can see that state-of-the-art models have WER rates of $5 - 7\%$ [23], which can be achieved through

methods far beyond the computational resources and infrastructure that we have. Typically, models with 30% WER and below are considered to be reasonable, and models with 20% and below are pretty good [24].

The MER metric is similarly defined, except instead of dividing by the total number of words in the reference, we instead divide by the total number of words in the comparison. Specifically, it follows the formula

$$\text{MER} = \frac{S + I + D}{H + S + I + D}$$

where S, I, D are defined as before, and H is the number of correctly matched words (also known as hits). The main advantage of the MER metric is that it's symmetric, so it doesn't rely on the existence of a reference document. Additionally, it's less punishing if the reference text is shorter than the generated audio. This may make an impact because the whisper model can generate additional tokens such as `[blank audio]` and sound effects like `*laughs*` which could be ignored in a manually generated Youtube transcription.

Finally, the WIP and WIL metrics are defined by

$$\begin{aligned}\text{WIP} &= \frac{H}{N_1} \cdot \frac{H}{N_2} = \frac{H}{H + S + D} \cdot \frac{H}{H + S + I} \\ \text{WIL} &= 1 - \text{WIP}\end{aligned}$$

where N_1 and N_2 are the lengths of the reference and predicted transcripts. The WIP is essentially the product of the precision and recall of the transcription model, so their product represents the proportion of information that is both complete *and* correct, which seems to be a better indicator of how useful it is for the student.

Benchmarking

Each model, ranging from `ggml-tiny` to `ggml-medium` is ran on the full length of each of the lectures using the streaming algorithm, and the generated transcript is compared against the

reference transcript provided by Youtube.

The average performance of each of the models is recorded the table below.

model	model size	wer	mer	wil
ggml-tiny-q4_0	24.1MB	0.250023	0.231432	0.327206
ggml-tiny-q4_1	26.3MB	0.234503	0.218255	0.306492
ggml-tiny-q8_0	41.5MB	0.211942	0.196978	0.271527
ggml-base-q4_0	44.3MB	0.213587	0.198108	0.268371
ggml-base-q4_1	48.5MB	0.220198	0.200486	0.263924
ggml-tiny	74.1MB	0.213108	0.197905	0.272346
ggml-base-q8_0	78.0MB	0.204044	0.187345	0.246854
ggml-small-q4_0	138.7MB	0.166375	0.156495	0.204539
ggml-base	141.1MB	0.205332	0.188357	0.248127
ggml-small-q4_1	152.9MB	0.175047	0.163765	0.213687
ggml-small-q8_0	252.2MB	0.172385	0.161213	0.208845
ggml-medium-q4_0	423.9MB	0.165400	0.155491	0.201629
ggml-small	465.0MB	0.171554	0.160593	0.208353
ggml-medium-q4_1	469.1MB	0.166597	0.156772	0.203750
ggml-medium-q8_0	785.2MB	0.163595	0.153996	0.199081
ggml-medium	1.4GB	0.164047	0.154314	0.199444

Table 4.1: Comparison of Whisper models by size, quantization level, and accuracy metrics. The table is sorted in increasing order of model size.

Unsurprising, model performance increases with model size. The smallest model, `ggml-tiny` has the highest error rates across the board. However, it's quite surprising that the quantized `q8_0` models sometimes outperform their half-precision versions. We can verify with the model sizes that these 8-bit quantized models offer a significant performance improvement over their unquantized counterparts, but also comes with no loss of quality. Thus we choose to proceed with the 8-bit quantized models for further analysis.

4.2.2 Quantization in Notes Generation

The quality of generated notes is difficult to evaluate because there's no correct answer. Human evaluation is the gold standard, but it's costly because lectures are so long. Additionally, since people have different preferences on what notes should look like, a lot of data is needed for accurate analyses.

One simple and popular metrics for evaluating quality of notes is the ROUGE score [25]. This scoring system has multiple variations, but generally they all represent a metric of word overlaps between the original document and the summary or notes. For example, ROUGE-N counts the proportion of N-grams that are matching, and ROUGE-L is based on the longest common subsequence of the two documents. Unfortunately, one flaw of the ROUGE metric is that plagiarizing the original document results in a good score, even though some of our important priorities are to be efficient and concise.

For these reasons, we choose to additionally use LLMs to rate the quality of the generated notes. Specifically, we use OpenAI's GPT 4 model. We believe that LLM as a judge is effective for analysis in this case because of the vast difference in model size. GPT 4 is nearly 1000× the size of our smallest models, so we expect it to be also much more capable, and rate the generated notes in a reasonable way.

Notes generation is evaluated using the reference transcript to ensure consistency and to decouple the performance of notes generation from the performance of the transcription model.

model	rouge-1	rouge-2	rouge-L	judge score
`Qwen3-4B-awq-q4_1`	0.341021	0.097704	0.335839	7.972628
`Qwen3-4B-f32`	0.356851	0.105455	0.349231	7.987226
`Qwen3-4B-q4_1`	0.369772	0.125753	0.362905	7.958029

Table 4.2: ROUGE and LLM-as-a-judge score for quantizations of the Qwen3-4B model. These are averaged across all partially generated notes across all 20 videos in the dataset. The `Qwen3-4B-awq-q4_1` model is AWQ quantized, and `Qwen3-4B-q4_1` is quantized using a GGUF-provided quantization algorithm.

We can see in table 4.2 that the GGUF quantized model has the highest ROUGE scores across the board. However, we know that ROUGE scores can be artificially inflated by repeating back the transcript. Instead, if we look at the the judge score, which has been prompted to value conciseness, we see that the original, unquantized model performs the best, and the AWQ quantized version does outperform the GGUF quantized version. As a

result, we prefer to use the AWQ quantized model over the GGUF quantized model.

4.2.3 End to End Note Generation Latency

In order to analyze the performance of the application in a lecture environment, we perform end to end latency benchmarks. We open the application on a laptop, enable recording and notes generation, and then play a recorded lecture from a phone located elsewhere in the room. Since these experiments take a really long time to run, we only simulate the first 15-20 minutes of one of the lectures. We also expect that the latency varies significantly less than quality across the different lectures.

The app has a built-in monitoring system that tracks several key timing data. It tracks what time each audio segment is received by the computer, which is the time that the student hears this part of the lecture. Next, after the audio segment is transcribed into text by the speech recognition model, we record the time that it is displayed onto the app for the student to see. Because of the streaming transcription algorithm, this delay is at minimum 3 seconds, because that's how long the transcription window length is. In order to make the application feel more real-time, we can choose to shorten this window or optimize the inference to bring the true delay as close to the transcription window length as possible. Next, we track how long it takes for this line of the transcript to be incorporated into the notes. We detect that this incorporation happens as soon as the streaming notes generation algorithm responds with the new set of notes after receiving this transcript line. Because multiple transcript lines can be bucketed together, the generation times of these transcript lines will be the same. This is why we see a staircase graph in the data.

Graph 4.1 displays three lines, indicating the timing information described above. The x-axis is numbered starting from $i = 1$ indicating the i-th transcript line. There are three data points for each transcript line, at each of the three lines. They correspond to when the first audio sample is received, when transcript line i is generated by the transcription algorithm, and when line i is incorporated into the notes by the notes generation algorithm.

Their y-coordinates indicate these times.

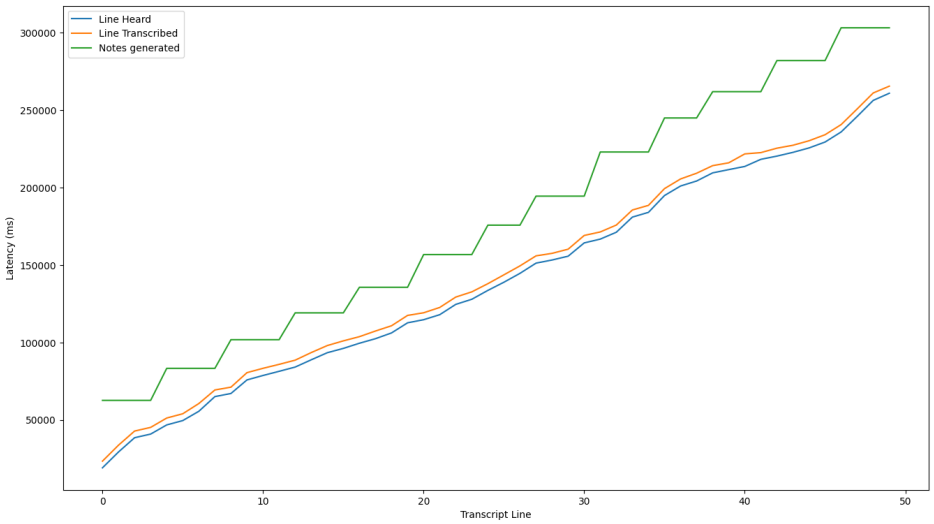

Figure 4.1: Diagram showing latency of transcription and notes generation

We can see that transcription is very quick, and consistently happens within a few seconds of hearing the audio input. On the other hand, there a larger delay in notes generation, and this delay grows slowly as the lecture goes on. This is because the number of input tokens into the model slowly grows as the notes get longer. Thanks to diff generation, the number of generation tokens remains roughly constant, so the delay is contained as much as possible.

4.2.4 Examples of RAG

We test the effectiveness of RAG-generated outputs by comparing how the LLM answers prompts when it has the necessary context, and when it does not have the necessary context. We expect that if asked a difficult question, the model will be able to identify that it does not know the answer, and will thus choose to retrieve context. The following experiments are performed with AWQ-quantized Qwen-4B models, with a temperature of 0.7.

First, we test if the chat bot can properly use RAG to accurately report details about the SmoothQuant paper [4]. We ask a very general question so the model can choose to be vague instead of using RAG.

We can see in figure 4.2 that the model recognizes its own lack of knowledge regarding

SmoothQuant, and thus takes a guess as to what it is. It ends up hallucinating what the paper is about, but it does successfully recognize that it is related to quantization. This suggests that the model is aware of when it is missing information, and can meaningfully decide when it needs additional context. Sure enough, when given the ability to search context, the model searches for the relevant information and provides a detailed, accurate response for the user, as depicted in figure 4.3.

Next we examine cases when the model is asked a very specific, technical, math question. We chose to ask about a novel result from a recently released paper that is not in the training data of the model: the bounds for the Chvátal-Sankoff constant in a specific case [26]. Since this is not an open-ended question, we wonder whether the model will hallucinate a response or not provide an answer.

Figure 4.4 shows that, without RAG, the model does not provide the right answer, and expresses no doubt in its hallucinated response. This can confuse the user, or potentially even mislead them. When the model does have access to RAG, but the user has not uploaded the relevant files, we can see in figure 4.5 that the model tries to retrieve context but gets the wrong document (the SmoothQuant paper). Although it responds with paragraphs of unrelated information, it does not provide the user with an incorrect answer, but instead asks for more information. This could prompt the user to upload the correct documents, and is overall beneficial. When provided with the correct documents, we can see in figure 4.6 that the model correctly identifies the bound, and references where it received the information from.

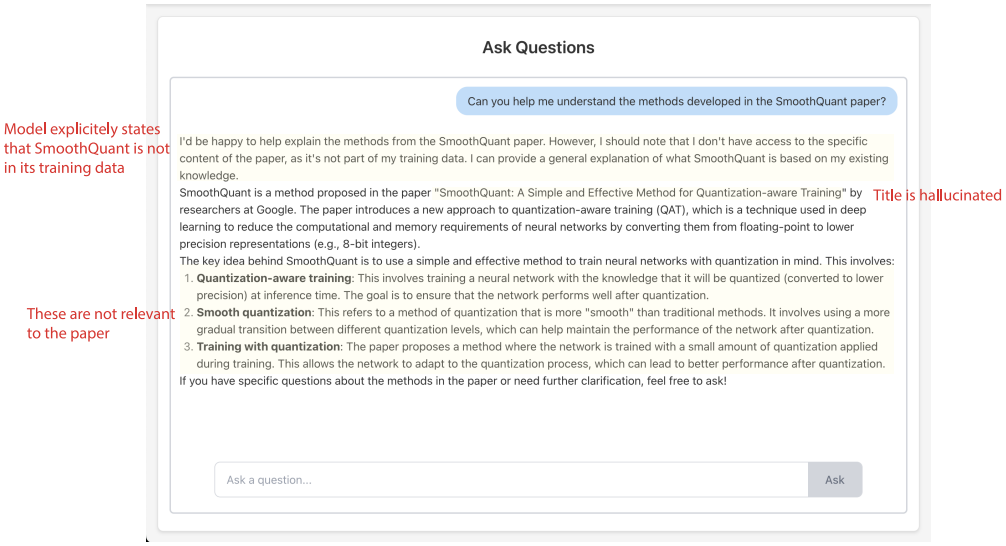

Figure 4.2: Model output without RAG

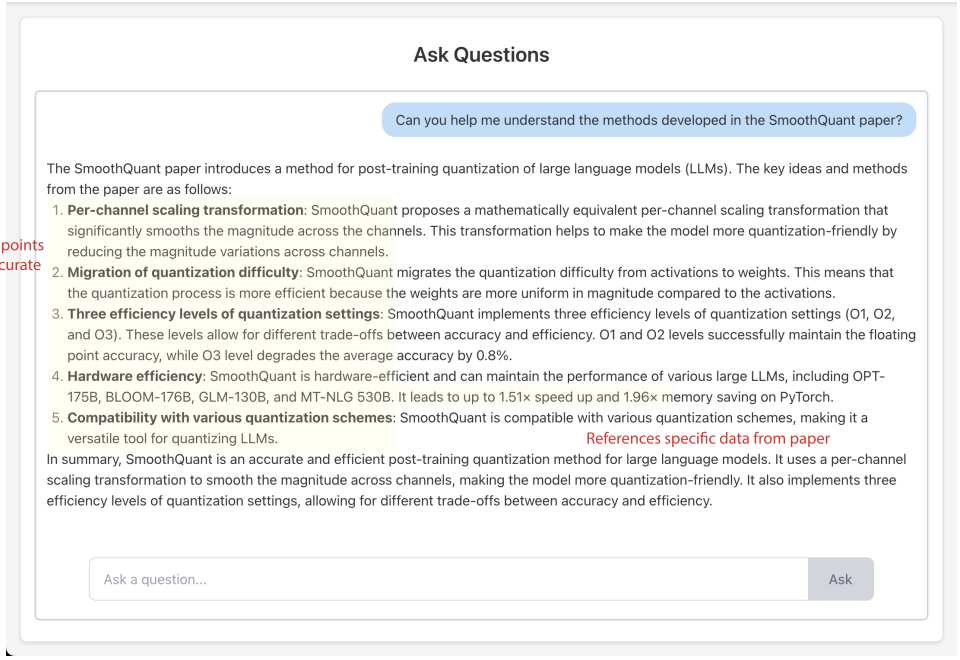

Figure 4.3: Model output with RAG

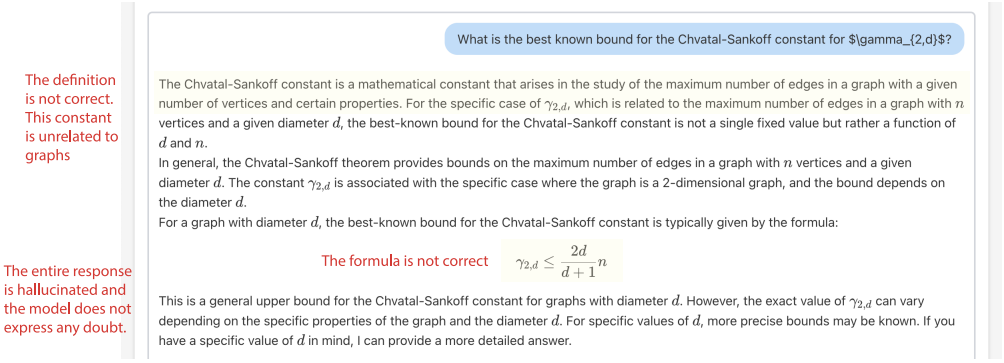

Figure 4.4: Model output without RAG

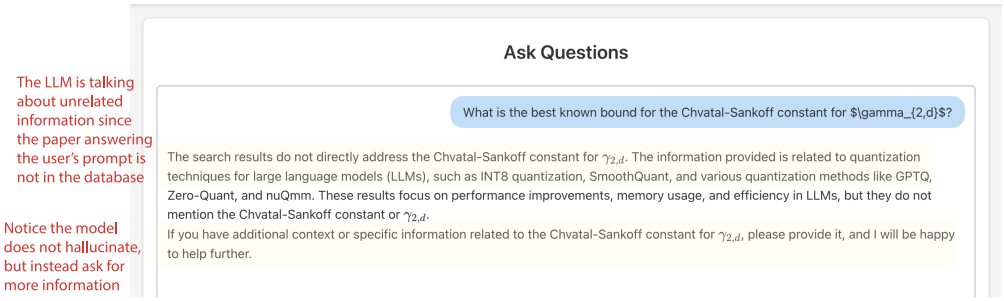

Figure 4.5: Model output with RAG, but with the wrong documents

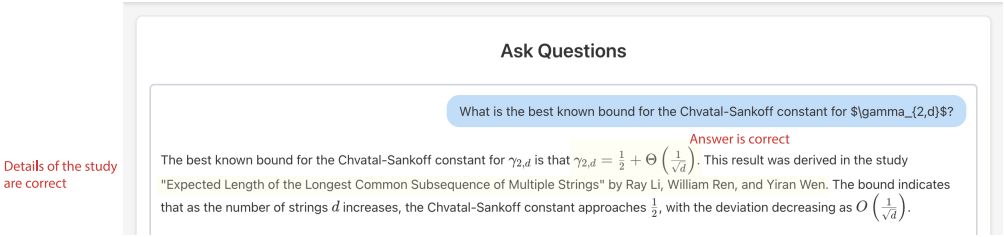

Figure 4.6: Model output with correct RAG

Chapter 5

Conclusion

Large language models are becoming increasingly efficient as optimization techniques advance, and commercial-grade laptops continue to improve in performance. While most modern applications still rely on third-party providers for LLM inference, a new class of offline, AI-powered applications is emerging. Cognify serves as a benchmark for this middle ground, combining low-level inference optimizations with intelligent generation algorithms to maximize performance in computationally constrained environments.

Cognify implements three core features: live transcription, streaming notes generation, and a RAG-powered Q&A interface. These features were evaluated in simulated lecture environments to assess performance and quality, highlighting the trade-offs inherent to operating in a private, resource-limited setting.

Looking forward, Cognify could expand beyond desktop platforms into mobile devices, where performance constraints are even more stringent. Incorporating multi-modal models capable of interpreting images, diagrams, and other media could further enhance capabilities, though at higher computational cost. Ultimately, Cognify demonstrates that local inference is feasible and provides a foundation for the development of future offline AI applications.

References

[1] S. Han, H. Mao, and W. J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV]. URL: https://arxiv.org/abs/1510.00149.

[2] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han. *HAQ: Hardware-Aware Automated Quantization with Mixed Precision*. 2019. arXiv: 1811.08886 [cs.CV]. URL: https://arxiv.org/abs/1811.08886.

[3] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han. *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*. 2024.

[4] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. "SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models". In: *Proceedings of the 40th International Conference on Machine Learning*. 2023.

[5] Y. Zhong, Y. Huang, J. Hu, Y. Zhang, and R. Ji. *Towards Accurate Post-Training Quantization of Vision Transformers via Error Reduction*. 2025. arXiv: 2407.06794 [cs.CV]. URL: https://arxiv.org/abs/2407.06794.

[6] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. arXiv: 1712.05877 [cs.LG]. URL: https://arxiv.org/abs/1712.05877.

[7] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer. *I-BERT: Integer-only BERT Quantization*. 2021. arXiv: 2101.01321 [cs.CL]. URL: https://arxiv.org/abs/2101.01321.

[8] Y. Bondarenko, M. Nagel, and T. Blankevoort. "Understanding and Overcoming the Challenges of Efficient Transformer Quantization". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 7947–7969. DOI: 10.18653/v1/2021.emnlp-main.627. URL: https://aclanthology.org/2021.emnlp-main.627/.

[9] H. Wang, Z. Zhang, and S. Han. "SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Feb. 2021. DOI: 10.1109/hpca51647.2021.00018. URL: http://dx.doi.org/10.1109/HPCA51647.2021.00018.

[10] Y. Leviathan, M. Kalman, and Y. Matias. "Fast inference from transformers via speculative decoding". In: *International Conference on Machine Learning*. PMLR. 2023, pp. 19274–19286.

[11] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao. *Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads*. 2024. arXiv: 2401.10774 [cs.LG]. URL: https://arxiv.org/abs/2401.10774.

[12] G. Gerganov and contributors. *llama.cpp*. URL: https://github.com/ggml-org/llama.cpp.

[13] G. Gerganov and contributors. *whisper.cpp*. URL: https://github.com/ggml-org/whisper.cpp.

[14] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever. *Robust Speech Recognition via Large-Scale Weak Supervision*. 2022. arXiv: 2212.04356 [eess.AS]. URL: https://arxiv.org/abs/2212.04356.

[15] J. M. Michael Chiang and contributors. *Ollama*. URL: https://ollama.com.

[16] N. Chhibbar and J. Kalita. *Automatic Summarization of Long Documents*. 2024. arXiv: 2410.05903 [cs.CL]. URL: https://arxiv.org/abs/2410.05903.

[17] S. Cho, F. Dernoncourt, T. Ganter, T. Bui, N. Lipka, W. Chang, H. Jin, J. Brandt, H. Foroosh, and F. Liu. "StreamHover: Livestream Transcript Summarization and Annotation". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 6457–6474. DOI: 10.18653/v1/2021.emnlp-main.520. URL: https://aclanthology.org/2021.emnlp-main.520/.

[18] S. Narayan, S. B. Cohen, and M. Lapata. "Don't Give Me the Details, Just the Summary! Topic-Aware Convolutional Neural Networks for Extreme Summarization". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Ed. by E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 1797–1807. DOI: 10.18653/v1/D18-1206. URL: https://aclanthology.org/D18-1206/.

[19] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis. "Efficient Streaming Language Models with Attention Sinks". In: *ICLR* (2024).

[20] P. Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401.

[21] Y.-C. Chen, P.-C. Hsu, C.-J. Hsu, and D.-s. Shiu. *Enhancing Function-Calling Capabilities in LLMs: Strategies for Prompt Formats, Data Integration, and Multilingual Translation*. 2024. arXiv: 2412.01130 [cs.CL]. URL: https://arxiv.org/abs/2412.01130.

[22] A. C. Morris, V. Maier, and P. Green. "From WER and RIL to MER and WIL: improved evaluation measures for connected speech recognition". In: *Interspeech 2004*. 2004, pp. 2765–2768. DOI: 10.21437/Interspeech.2004-668.

[23] C.-C. Chiu et al. *State-of-the-art Speech Recognition With Sequence-to-Sequence Models*. 2018. arXiv: 1712.01769 [cs.CL]. URL: https://arxiv.org/abs/1712.01769.

[24] C. Park, M. Chen, and T. Hain. *Automatic Speech Recognition System-Independent Word Error Rate Estimation*. 2024. arXiv: 2404.16743 [cs.CL]. URL: https://arxiv.org/abs/2404.16743.

[25] C.-Y. Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013/.

[26] R. Li, W. Ren, and Y. Wen. *Expected Length of the Longest Common Subsequence of Multiple Strings*. 2025. arXiv: 2504.10425 [math.CO]. URL: https://arxiv.org/abs/2504.10425.