FELDM
The Data
Collective.

Building a local, privacy-respecting meeting transcript and summarization tool – a technical journey:

TOPICs: Large Language Models, Prompt Engineering, GPT, Meta, NLP, Meeting Summary, Privacy, GenAI

Authors:

João Tozato
Dr. Matthias Böck

I. Introduction:

One of the most common practical applications of Large Language Models (LLMs) is text summarization, which can contribute to several other fields, including retrieval-augmented generation (RAG), data analysis and long-document Q&A.

The focus of this whitepaper narrows to examine one specific, yet significant, application, meeting transcription and summarization. We will explore how LLMs can automate this manual task and improve overall efficiency and output quality.

In this whitepaper, we share our technical journey to build a privacy-respecting, locally hosted meeting summarization tool. Using open-source frameworks like Whisper and open LLMs, we aim to give technical guidance on how to actually build such a system and to share our journey with ups and downs following the stream of recent advances in Artificial Intelligence.

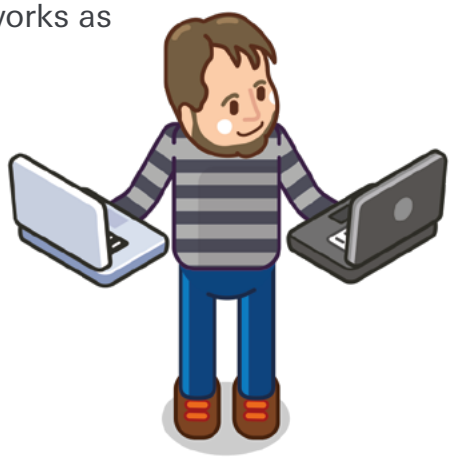
II. Why (another) meeting summarization?

Meeting notes are an important key to aligning and documenting the most important outcomes and are additionally an essential piece to asynchronous communication. Traditionally, the contents of organizational meetings are captured manually, often requiring a designated individual to transcribe and summarize key points - a method that is time-intensive, biased on the perspective of the scribe and partially preventing the scribe from participating in the discussions. Various stand-alone generative AI tools or built-in tools such as the Microsoft Copilot within Teams, allow us now to transcribe our meetings in real-time (also to different languages) and to give us a summary of these transcripts. This will be more and more conveniently integrated into our office processes but raises the question if we want to record, store and analyze all our communications.

Imagine a casual exchange about your weekend or a confidential side note to your colleague during a meeting, which will be also added to the meeting notes and stored as transcript and potentially audio file. Knowing that everything will be recorded also changes the dynamics of our communication and responses will be more controlled and the meeting tone more serious. For the same reasons participants will be more reluctant to use such a tool. Moreover, relying on third-party services introduces risks, from data breaches to regulatory violations (e.g. GDPR).

Our aim was to build a locally hosted solution that encompasses the real time transcription of meetings from an audio feed and then generates meaningful summaries using open-source tools on limited local hardware. Our solution should remove all data (audio, transcript) after having created the summary and achieve the following benefits:

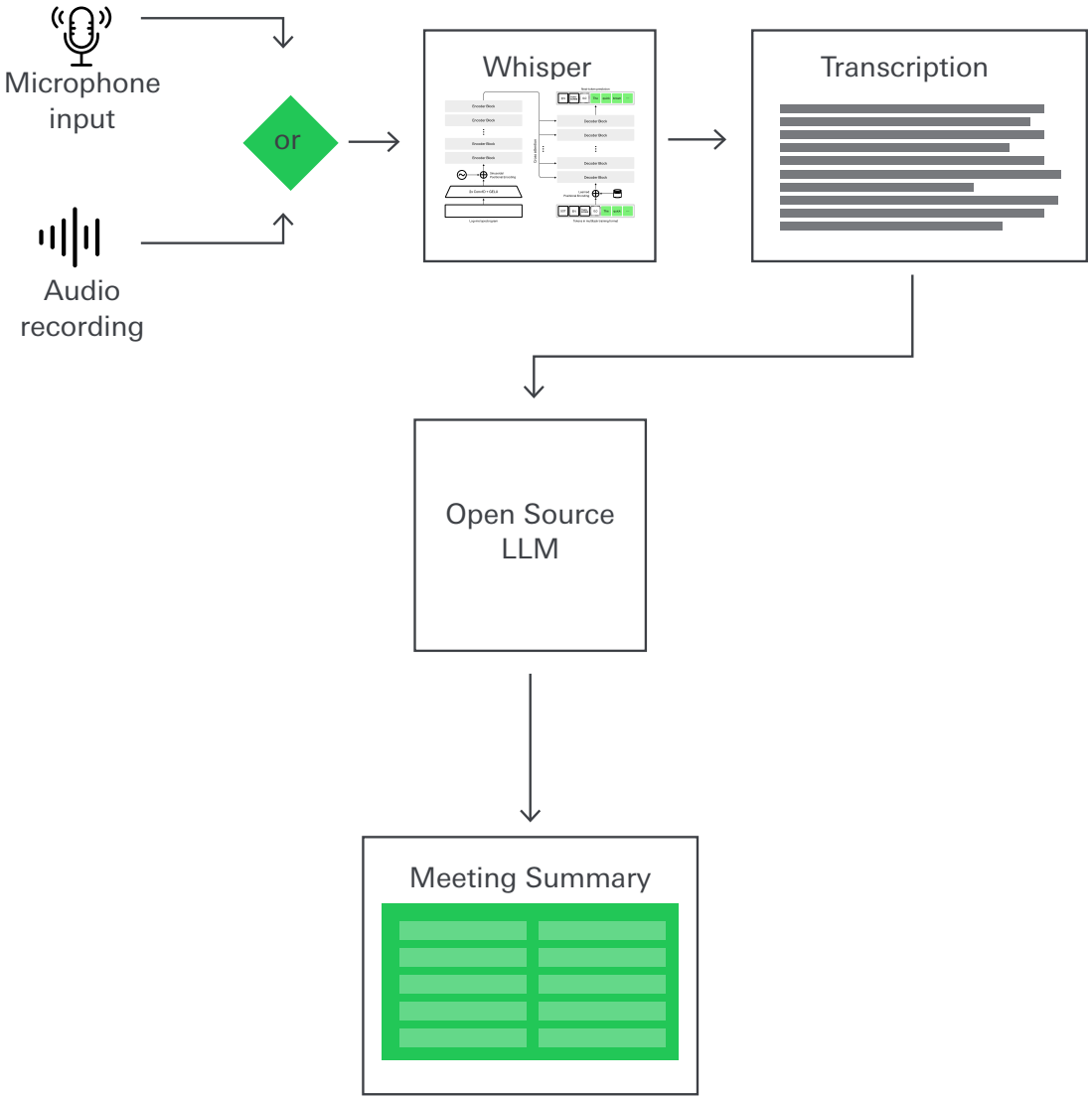
- Enhanced privacy and data control: By hosting the transcription and summarization tool locally, we ensure that sensitive meeting data remains on-premises, reducing the risk of exposure to third-party services and giving organizations full control over their data.
- Customizable and flexible: An own solution allows for greater customization such as adjusting the model for domain-specific language, customizing summarization criteria, and integrating with existing workflows.
- Improved trust and user adoption: Knowing that meeting content isn't automatically stored or analyzed externally can increase user trust and willingness to adopt the tool.
- Hands-on experiences: Testing and learning how to make use of the recent model advances and newly created frameworks as well as to be aware of their limitations.



In the following we want to showcase the possible ways to make use of state-of-the-art (and open source) LLMs and NLP tools to tackle this problem of speech-to-text and abstract summarization.

This approach consists of the following: the system captures live audio streams from meetings, transcribes them on-the-fly, and then creates concise summaries that retain pivotal details and the surrounding context. The following diagram gives an overview of all included steps:

Figure 1: End-to-end workflow of the locally hosted meeting transcription and summarization system. The process includes capturing live audio, transcribing it with Whisper, and generating summaries using open-source LLMs.



III. Our technical setup

Our server has the following specifications:

- Processors: Dual Intel® Xeon® E5-2620 v4 (32 threads)
- RAM: 128GB
- GPU: NVIDIA RTX 4090 with 24GB VRAM

Despite having a lot of system memory and CPU cores, we are constrained by GPU memory. This limitation directly affects our ability to run both the transcription model and the large language model simultaneously on the GPU, which is a requirement for generating the transcriptions and summaries fast and parallel for different meetings. To mitigate these resource constraints, we prioritized efficient model configurations, including memory optimization techniques and strategic GPU utilization. This approach allowed us to balance hardware limitations with the system's performance goals.

IV. 3 steps to Summarization

Creating a summarization pipeline only consists of three major building blocks:

A. Generating multilingual transcripts from given audio
B. Set up of your own locally hosted large languagemodel (LLM)
C. Building the meeting summarization pipeline

We will give an overview of our chosen components and give details of how we set them up, what worked but also what did not work for us. Challenges along the way included:

- Choosing the right model and the right model size
- Efficient use of limited local hardware resources
- Setting up a proper evaluation framework

A. Step 1:

Whisper – generate multilingual transcripts on your own system

For our speech-to-text engine, which creates the transcript based on a given audio file, we used Whisper. Whisper is one of OpenAI's open-source multitasking speech recognition models designed for multilingual speech recognition, speech translation and language identification. It is built using a transformer architecture and uses an encoder-decoder structure together with attention mechanisms to process audio recordings. This model has been trained on 680,000 hours of multilingual data from several sources, and the sheer size and diversity of its training data makes Whisper a very robust model in transcription accuracy. Whisper is best performing on English data sets but can also handle several other languages such as German, French or Spanish. Particularly, German is important for several of our meetings.

Since Whisper was open sourced from its release, the open-source community quickly built upon it, notably with the release of whisper.cpp. This C++ library optimizes the model for low latency and high throughput inference and even includes quantization features to allow for less memory and disk usage. All of this makes the whisper model lightweight, efficient and highly portable, since it can run even on very constrained hardware (e.g. a smartphone or even a Raspberry Pi!).

Whisper is available in various model sizes, from tiny to large, which need to be chosen based on the trade-off between transcription accuracy and processing time. In our experience, the small model delivered satisfactory results for English. However, for German, we needed the large model to achieve reasonably accurate transcriptions. One word of advice, no matter which release of Whisper you are using, it is still a probabilistic model and prone to make mistakes.

Using whisper.cpp has allowed us to quickly experiment with different model sizes and performance trade-offs, which helped us to assess how well the model sizes meet our needs across these multilingual transcription tasks. However, we found that integrating whisper.cpp into our setup introduced several difficulties due to the lack of a straightforward Python API, which made the process more complex and required additional efforts, particularly when building the library to enable CUDA/GPU acceleration support.

To address these integration challenges, we also explored Faster Whisper as an alternative method to self-host Whisper on our premises. This library is built on the CTranslate2 inference engine, which is optimized for efficient, high-performance inference of transformer models and offers an easy-to-use, Python-friendly API.

In our experiments, as shown in Table 1, we found that this library was not only easier to set up and integrate into our existing workflows, but it also was more efficient in GPU memory allocation and transcription speed. For example, transcribing a one-hour meeting takes less than a minute and uses only about 1.8 GB of VRAM, making it exceptionally fast and resource-efficient for our meeting transcription tasks.

Table 1: Performance (latency and VRAM usage) comparison between Faster Whisper and whisper.cpp for transcribing a one-hour audio file.

	Latency (minutes)	VRAM (GB)
Faster Whisper	0.82	1.8
Whisper.cpp	1.58	1.9

In the next section we will have a look at the second step of how-to self-host a LLM and which frameworks are helpful.

B. Step 2:

Self hosting a LLM – efficient use of a locally owned model

To generate the summaries from the transcripts, we use a large language model (LLM) to process and extract the key points while retaining the context and important details of the meetings. By applying a LLM to the raw meeting transcripts, we can automate the summarization process, not only saving time but also ensuring consistency in capturing critical information from our internal meetings.

Given that the contents of our internal meetings are often sensitive, we prioritize keeping this data within our own infrastructure, avoiding the need to send it to external servers. For such use cases, deploying an LLM on-premises offers several benefits in terms of control, privacy and customization, such as:

- No Vendor Lock-In: Using open-source models avoids dependence on specific vendors, offering more flexibility and cost savings by not depending on usage costs associated with proprietary models.
- Privacy and Data Control: Self-hosting ensures that all sensitive data remains entirely within the organization's infrastructure, reducing the risk of data leakage and enabling stricter control over data handling and retention policies.
- Keeping Up with the State-of-the-Art: Open-source models are increasingly more capable due to the fast pace of community contributions. Self-hosting allows us to explore the latest model developments, keeping track of tools, methods and model improvements as soon as they become available.

In a self-hosted LLM setup, there are two main components involved: the inference engine, and optionally, but highly recommended, an inference server. The inference engine is responsible for executing the model and managing the generation process efficiently, such that resource usage is optimal, and the responses are fast. An inference server can be seen as an extension of the engine, handling incoming and outgoing requests, typically through HTTP or gRPC, which makes integration with external applications easier and also enables scaling for multiple concurrent users.

When selecting a model to run within this self-hosted setup, especially in a single GPU server like ours, we had to consider several factors to ensure efficiency and maximize performance. The number of model parameters, text generation throughput, output latency, and hardware constraints all influenced our decision on a framework that could best meet our needs.

Furthermore, techniques like weight quantization were quite important, as they allowed us to reduce memory usage and improve processing speeds, making it feasible to deploy larger models on our limited hardware. These considerations were what guided us in identifying a combination of inference engine framework and model choice that would be optimal for our setup.

In the following we compare some of the currently most used inference engines and share our experience with them for our text generation tasks.

HuggingFace's Text Generation Inference (TGI)

This framework is designed specifically for serving large language models efficiently and on large scale setups, it has a relatively simple launcher to serve popular LLMs and several production-ready features such as tensor parallelism for multi-GPU settings, continuous batching of incoming requests to increase total throughput, as well as optimized code for inference, using Flash Attention and Paged Attention on widely used transformer architectures.

Another compelling aspect of Hugging Face TGI is its integration with the Hugging Face Hub model ecosystem, allowing easy access to several community contributed pre-trained and fine-tuned models. It also offers an OpenAI-compatible API, which makes it easy to integrate with applications already designed around proprietary models, such as GPT-4o.

However, TGI does require constant GPU allocation, meaning that the model remains in GPU memory once loaded, which can be a limitation in environments with shared and limited GPU resources. In our experiments, we were

able to fit up to a 13-billion parameter model within 24 GB of VRAM using 8-bit quantization and a context length of at most 8,192 tokens. This configuration yielded a throughput of around 70 tokens per second, making it quite fast in terms of generation speed.

Due to the high VRAM usage, TGI wasn't exactly suited for our use case, as we also needed to allocate the Whisper model in VRAM simultaneously. This requirement meant that we couldn't allow the LLM to occupy all the VRAM continuously, which led us to explore alternative inference engines better suited to our single GPU setup.

Llama.cpp

Another framework we explored is llama.cpp, a CPU-first, high-performance C++ library that allows for broad device compatibility. It is designed to work with several backends, including CUDA, Metal, Vulkan and CPU directly, making it feasible to host LLMs even on smaller devices like smartphones or low-powered computers. It also supports CPU offloading, which allows layers of the model to be shifted from the GPU to the CPU when VRAM is a limiting factor.

For example, in our single 4090 setup, llama.cpp can support models up to 56 billion parameters by using 4 bits-per-weight quantization, with a context length of 8192 tokens, resulting in a throughput of roughly 5 tokens per second and utilizing around 22 GB of VRAM. Despite allowing us to self-host larger models, this framework does have some relevant limitations. The manual build process for the library can require quite a bit of technical knowledge and effort, as users must compile the library and configure it for their specific hardware, which can make the overall setup challenging.

Furthermore, it also lacks a fully featured inference server, which adds an extra layer of complexity to deploy and manage models effectively. Overall, we also found out on our experiments that the text generation speeds are generally slower than those of GPU-first frameworks, impacting throughput, especially for larger models. These are the reasons why we decided against using llama.cpp for our setup, as the limitations outweighed the benefits for our specific use case.

Our Text Generation Engine of Choice: ExLlamaV2

After evaluating the previously discussed LLM inference frameworks, we chose ExLlamaV2 as our text generation engine due to its high-performance inference and efficient memory management features, which aligned well with our single GPU setup and met our speed and latency requirements.

ExLlamaV2 is an incredibly fast inference engine, specifically designed with several optimizations, such as Paged Attention, Flash Attention, continuous batching, and prompt caching, plus the EXL2 format, which allows it to store the weights in an efficient manner. All these features help enhance its overall throughput and reduce latency for text generation tasks. Nonetheless, unlike other frameworks, it operates as an engine only, so an external inference server, such as TabbyAPI, is required to orchestrate and handle incoming requests and outgoing model responses.

In our experiments, ExLlamaV2 demonstrated remarkable efficiency in handling lower-bit quantization formats, allowing us to fit larger models within the 24 GB VRAM constraint of our GPU. Below are some examples of its performance:

1. Llama 3.1 (8 Billion Parameters):
    a. Quantization: 8 bits-per-weight
    b. Context Length: 32,768 tokens
    c. VRAM Usage: 9 GB
2. Mistral Nemo (12 Billion Parameters):
    a. Quantization: 8 bits-per-weight
    b. Context Length: 16,384 tokens
    c. VRAM Usage: 12 GB
3. Mixtral 8x7B (56 Billion Parameters):
    a. Quantization: 4 bits-per-weight
    b. Context Length: 4,096 tokens
    c. VRAM Usage: 22 GB

This efficiency makes ExLlamaV2 highly suitable for single GPU setups, accommodating both high-performance and resource-constrained scenarios.

Moreover, this framework doesn't require persistent GPU memory allocation, allowing the models to be loaded or unloaded quickly as needed with a single API call. This feature frees up the GPU for other applications and makes it quite a compelling option for environments where GPU resources are shared between multiple applications. However, it also has limitations when it comes to documentation and number of supported models, so users might need to convert Hugging Face models themselves if the desired model isn't available directly in the EXL2 format.

In our case, ExLlamaV2 provided the right feature set and balance between performance and memory efficiency, enabling us to run both Whisper and an LLM simultaneously in our single GPU setup. This allowed us to optimize our resource usage and ensured we wouldn't run into out-of-memory related issues, making it the ideal candidate for self-hosting an LLM on our own premises.

Overview of the LLM-serving frameworks:

Framework	Inference Engine	Upsides	Downsides
HuggingFace TGI	Feature-rich Inference API, multiple endpoints, support for Hugging Face's Chat Interface	Support for fine-tuned models; Custom prompt generation; Simple to switch Between models	Limited in single GPU setups; Persistent memory usage required; High hardware requirements for larger models;
Llama.cpp	CPU-first, high-performant C++ library	Compatible with most devices; Allows for running large models on modest hardware using RAM instead of VRAM.	Text generation is slower on GPU; Setup process can be complex; Limited supported model formats (GGML, GGUF)
ExLlamaV2	GPU-first, highly optimized for fast LLM inference framework	Allows multiple concurrent requests; Supports different bit-quantization formats; High text generation throughput;	Requires specific EXL2 model format; Limited to GPU-only setups; Smaller range of models to choose from;

C. Step 3:

Summarization with LLMs – creating a productive pipeline

For the transcript summarization workflow, we chose to work analogous to a map-reduce approach. This process is needed due to a particular limitation of language models which is the context window. This context window basically refers to the maximum amount of text that LLMs can consider as input for the generation of a response. Therefore, when dealing with longer text inputs, such as longer documents or, in our case, longer meeting transcripts, it's not possible to simply feed the input text in its entirety to the LLM. Closed models such as ChatGPT, Claude or Gemini come with rather big context windows but for locally hosted models this comes at the expense of bigger GPU and memory consumption.

That's where a map-reduce like approach is useful, since it effectively allows for managing long text inputs and getting around this context window limitation. Here's a basic outline of how this approach works:

1. Map step: In this phase, the document gets divided into smaller chunks, which are smaller, more manageable sections that fit within the language model's context window. Each of these chunks is processed independently to understand its content, context and extract their key points. In our particular use case for summarizing meeting transcripts, we found out that by prompting the model at this stage to act as a note taker, ensuring that the essential aspects and context are captured, making it easy for a person who wasn't at that meeting to understand the outcomes and follow-up actions.

2. Reduce step: The summaries from each chunk obtained from the last step are then combined and are fed again to the LLM to generate a final, coherent summary. This process involves identifying overlapping or redundant information from the smaller summaries, making sure that the main themes of the meeting are accurately represented, and finally creating a summary that effectively captures the key aspects of the entire input transcript. It is also in this step where we "steer" the output format of the final summary, e.g. how many paragraphs should it contain, whether it should add a list of the main themes and discussion points, etc.

IV. So how did we include the users and how was the process of integrating the tool into our workflows?

Building data products requires a user-centered approach to ensure they are usable, feasible and desirable. Following this principle for our meeting summary, the actual users were included into the development process from the beginning. To bring data products to life they should be accompanied via the following:

1. Ongoing evaluation and user feedback
2. Building user's trust
3. Workflow integration and scaling

To evaluate the meeting transcription and summarization tool we started to use it within the data product team's regular meetings. These weekly meetings provided a practical setting to test the tool in real-world scenarios and assess its ability to capture key discussion points. Since we already had assigned minute-takers for every meeting, they were also asked to evaluate the tool's summaries according to:

- accuracy: is the summary factually correct
- clarity: readability and structure of the summary
- relevance: are essential topics included or omitted

This structured feedback helped us to identify the areas where the tool needed the most improvement, especially in writing nuanced, yet concise details while also aligning with the established structure of manual meeting notes.

Based on this feedback, we iteratively refined the model's prompts and adjusted the format of the summaries to better align with our expectation of a comprehensive meeting summary. This entire process not only helped to improve the tool's outputs but also allowed us to tailor its functionality to our specific needs and to account for the unique structure of our internal alignments.

The transparency on the capabilities but also shortcomings of the tool helped to align the user's expectations and how they could integrate the tool into their

workflows. We additionally explained in dedicated sessions the functionalities as well as the underlying technologies, and how the data is handled with a focus on privacy (e.g. deletion after the summary creation). The combination of education and transparency led to gaining our user's trust.

Additionally, we gradually rolled out and tested our summarization tool for different meeting formats also within different departments with a similar approach. Summary prompts can now also be customized by the users. A next step will be for us to create further templates for general meeting formats such as breakfast (we have each week a company breakfast with updates around the company, specific hub team weeklies or monthlies and several other artifacts).

Improving and learning is still an ongoing task as well as developing new workflows, which help us to better and faster document our meetings. Looking ahead, we are also exploring how to integrate these summaries into a knowledge base to make the content more accessible across teams.

VI. What did we learn on this journey and what where the biggest hurdles?

From a technical perspective, setting up the entire self-hosting infrastructure required initially a lot of time and resources. Managing the limitations of a single GPU setup with restricted VRAM proved to be quite challenging, as hosting multiple models, such as the transcription and summarization ones, required an extensive experimentation and optimization to make sure they could run at the same time without running into out-of-memory issues.

Keeping up with the state-of-the-art advancements in LLMs and serving frameworks was also another challenge of this entire process. This is due to the difficulty of finding a proper balance between exploring new models and techniques and maintaining the stability of our current setup. On one side, the fast pace of the field and model releases offered more opportunities to improve overall performance and efficiency. On the other hand, putting these new models to work can also demand significant efforts, from adapting the existing

infrastructure to fit the new requirements, to extensively testing their compatibility and reliability compared to the existing workflows.

Key takeaways for us:

1. User engagement drives success.

Though a general rule for the development of data products, this was one of the key elements for us.

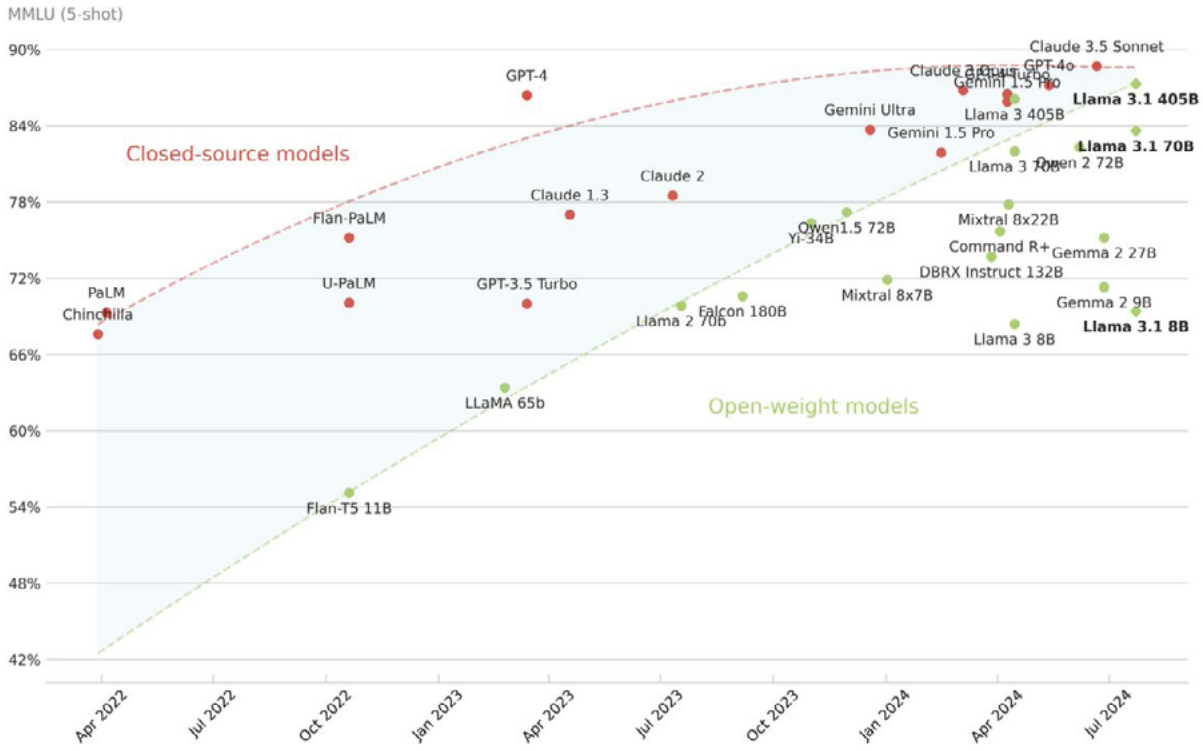
2. Balancing innovation with stability.

Staying up to date with LLM advancements is essential but requires constant screening of the latest developments as well as weighing the actual benefits vs efforts of updating our tool.

3. Hardware limits spark creativity.

Building on a constrained hardware environment pushed us to explore creative solutions, such as leveraging lower-bit quantization and memory-efficient inference engines.

Figure 2: Closed-source vs open-weight models performance on MMLU (5-shot) benchmark between April 2022 and July 2024 (copyright @maximelabonne)



VII. Conclusion/Outlook:

Even though the big players from Microsoft to Google or OpenAI are moving fast in offering out of the box and in their platforms integrated solutions, there is still a benefit in exploring the potential of your own premises. Our tool allowed us to explore the actual performance of open LLMs and how we can use them internally. At the moment, the big paid models such as GPT-4o or Gemini outperform the open local models and offer cheap API calls to build customized solutions. Still, one should consider the fast evolution of the open LLMs and consider what might be possible in the not too far future. Despite that no data leaves our own servers and we are immediately deleting any audio or transcript data, which is not needed anymore.

In the end, the main task was to reduce the amount of effort for taking notes and also to improve the quality. To achieve this, we worked in close iterations with our internal users to reach a solid understanding of how they would benefit from such a summary but also to show and tell about the capabilities and limitations of generative AI tools. Both aspects are key to a successful role out to gain trust in the solution and confidence in making use of it. A human still needs to be in the loop to critically assess the summary before it is shared with others.

With this in mind, we introduced the summarization tool now into several of our weeklies and other meeting formats, which helped us to significantly speed up the note taking.

If you need support in developing privacy-respecting AI products for your organization or are interested in collaborating on open-source solutions, we'd love to hear from you.

FELDM

The Data
Collective.