INFR11269 MACHINE LEARNING SYSTEMS

Homework 1: ASR on your own CUDA kernels

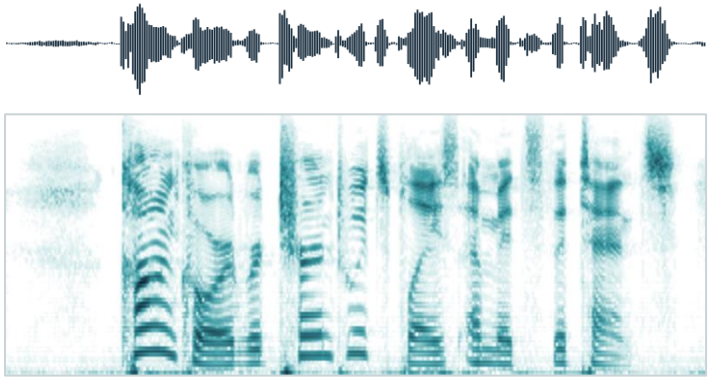


The model, the tasks and the benchmark
2026/27

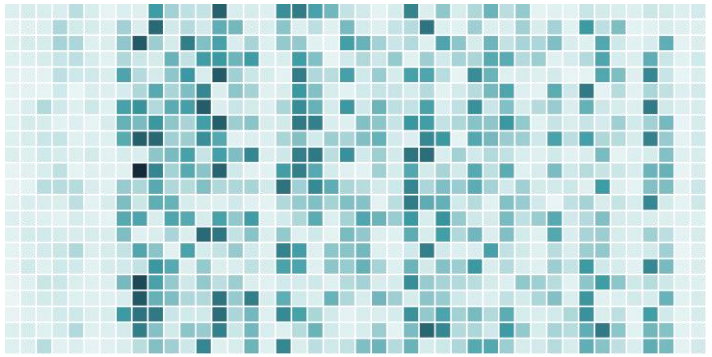
github.com/ed-aisys/edin-mls-26-autumn

What does an ASR model do?

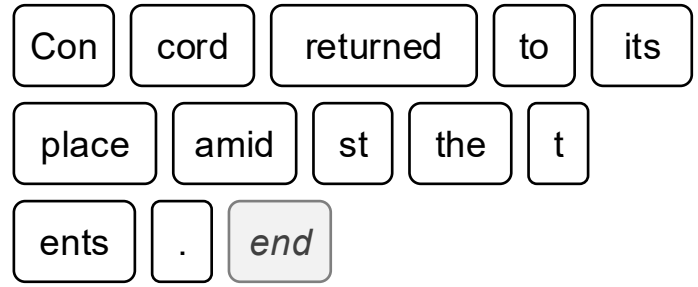
Automatic speech recognition: speech in, text out. GLM-ASR-Nano does it in three steps.



The real recording: waveform, then spectrogram



Encoder output (illustration)



The 13 tokens it writes for the recording

1. Hear

The recording is about 56,000 numbers: 3.5 s at 16 kHz. On the CPU it becomes a mel spectrogram: a picture of which pitches are loud at each moment.

2. Listen

The audio encoder reads the whole spectrogram and describes every 20 ms of sound with 1280 numbers, using the sounds around it as context.

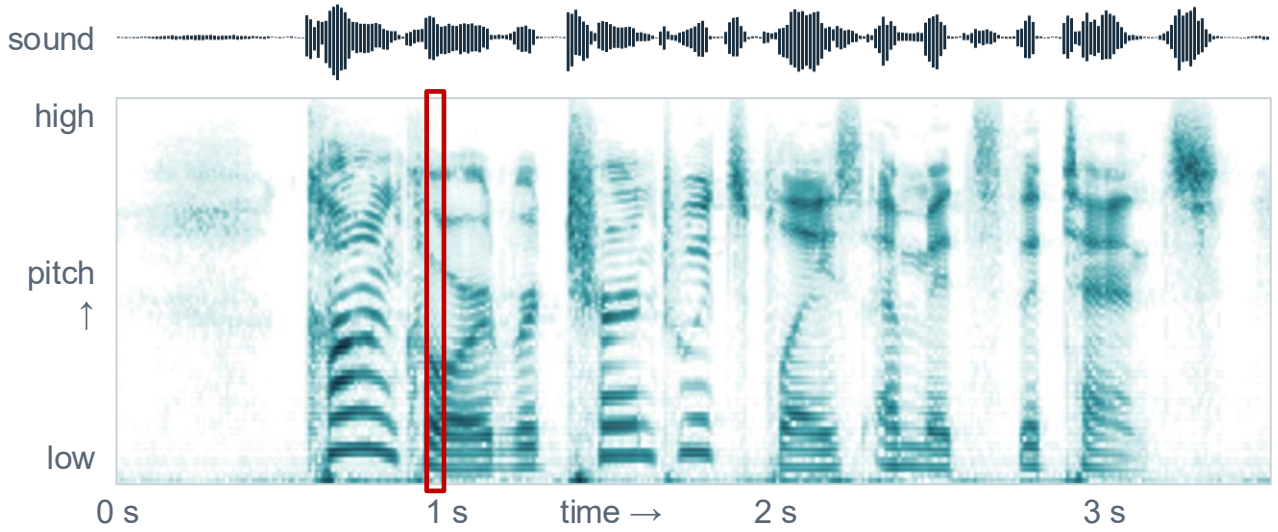
3. Write

The text decoder, a language model, writes the transcript one token at a time. A token is a word or a piece of one; the last one means “end”.

Steps 2 and 3 run on the GPU. Their arithmetic is what your kernels compute.

What is a mel spectrogram?

A picture of the sound: which pitches are loud at each moment. The model reads it like an image.



one column, zoomed in



Like a music equalizer

One column is a 10 ms snapshot: how loud each of 128 pitch bands is, low at the bottom, high at the top. 100 snapshots a second, side by side, make the picture.

How to read it

Time runs left to right, pitch bottom to top, and darker means louder. Vowels make dark stripes low down; “s” and “t” make smears high up.

Why “mel”

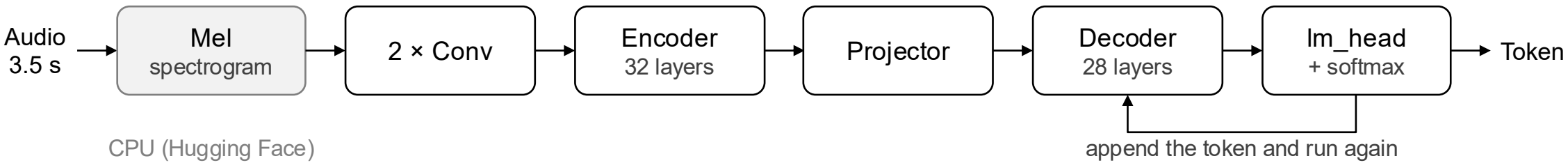
Ears tell low pitches apart far better than high ones: 100 vs 200 Hz is obvious, 7100 vs 7200 Hz is not. So the bands are narrow low down and wide high up.

How it is made

Cut the sound into 25 ms slices, one every 10 ms. Measure each pitch in every slice (a Fourier transform), add them up into 128 bands, take the log.

Every recording is padded to 30 s, so the model always gets 128 × 3000 numbers; this one fills the first 350 columns.

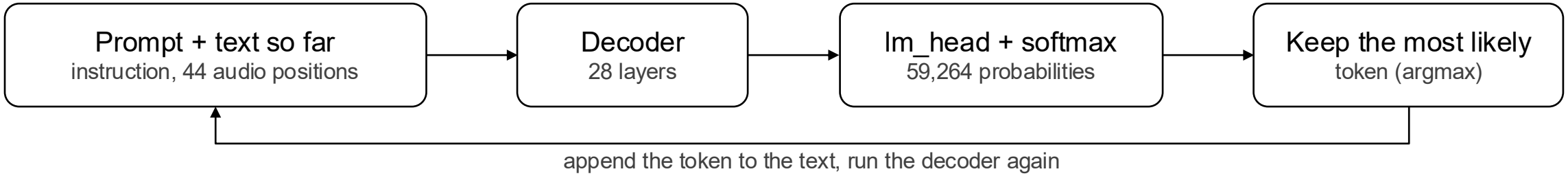
GLM-ASR-Nano, stage by stage



Stage	Purpose	Output shape	Your kernels
Mel spectrogram	Turn the sound into a picture of pitch over time	128 × 3000	none (CPU)
2 convolutions	Spot short sound patterns; halve time to 20 ms steps	1500 × 1280	im2col + Task 3
Audio encoder	Understand each sound in the context around it	1500 × 1280	Tasks 1–4
Projector	Shrink the audio 4× into something the decoder can read	44 × 2048	Task 3
Text decoder	Read the audio and the text so far; decide the next token	59 → 71 × 2048	Tasks 1–4
lm_head + softmax	Turn that into a probability for each of the 59,264 tokens	1 × 59,264	Task 3, Task 2

About 2.3 B parameters, stored as float32: about 9 GB of GPU memory.

How the decoder writes the text



The 13 runs for the test recording: the token each run keeps, and how many positions the decoder read in that run

run 1	run 2	run 3	run 4	run 5	run 6	run 7	run 8	run 9	run 10	run 11	run 12	run 13
Con	cord	returned	to	its	place	amid	st	the	t	ents	.	*end*
59	60	61	62	63	64	65	66	67	68	69	70	71

Greedy

Each run keeps only the most likely token: no sampling, no search.

No KV cache

Each run recomputes every position from scratch, so the input grows from 59 to 71.

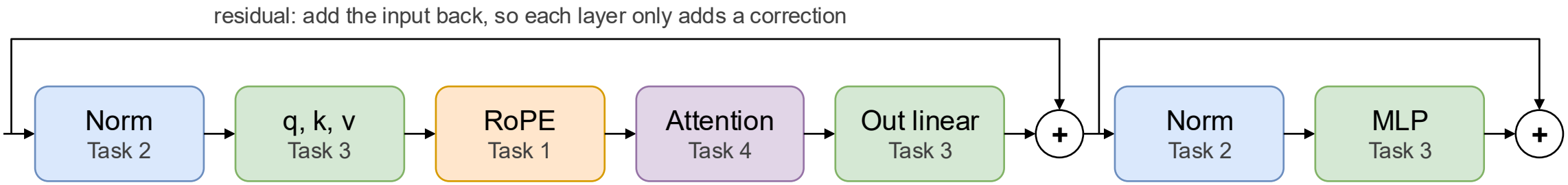
Stops at “end”

13 runs × 28 decoder layers, after one pass of the encoder.

The prompt is 59 positions: 15 tokens of instruction (“Please transcribe this audio into text”) and 44 of audio.

Inside one layer: where your kernels run

Every encoder and decoder layer has this shape. The linear layers (Task 3) do almost all the arithmetic.



Step	Purpose	Task
Norm	Keep the numbers at a steady scale, so dozens of stacked layers stay stable	Task 2
q, k, v	Give each position a question (q), a label (k) and a message (v) for attention	Task 3
RoPE	Tell attention where each position is, so that order matters	Task 1
Attention	Let each position gather information from all the others: this is where context comes in	Task 4
Out linear	Bring the gathered information back to the layer's width, then add the input back	Task 3
MLP	Process each position on its own; about two thirds of the parameters are here	Task 3

Calls per transcription: 2757 linear, 396 attention, 396 RoPE, 806 norm, 13 softmax.

What you need to do: four kernels

Task	File and test	What you write	The CUDA idea
1. RoPE	task1_rope.cu make test1	One thread per pair of numbers: rotate q and k in place	Index arithmetic; a memory-bound kernel
2. Reductions	task2_reduce.cuh, .cu make test2	Warp and block sum and max, then RMSNorm and a one-pass softmax	Warp shuffles, shared memory, barriers
3. Tiled GEMM	task3_gemm.cu make test3	The linear layer $y = x \cdot W^T + b$ in shared-memory tiles, with GELU fused	Tiling, coalesced reads, bank conflicts
4. Attention	task4_attention.cu make test4	Attention over tiles of 32 keys with an online softmax; GQA and causal mask	FlashAttention-style streaming

Fill in the blanks

Each blank is 1 to 15 lines. The code around them, the Python side and the tests are written for you.

Do the tasks in order: each builds on the last, and Task 4 calls your Task 2 warp reductions.

How to do it

One GPU per person: only --gres=gpu:1, one GPU shell or job at a time. Exit when you are done.

1. Get the code, set up once

Clone github.com/ed-aisys/edin-mls-26-autumn on the head node, then run setup_env.sh.

2. Get one GPU

The head node has no GPU. Each session, ask Slurm for one card, put nvcc on PATH and activate mls-asr.

3. Read the task guide

docs/taskN-*.md: what the kernel computes, which thread does what, and what each blank needs.

4. Fill in the blanks

Replace // YOUR CODE HERE, then run make and fix every compiler error.

5. Test, then run the model

make testN compares your kernel with a CPU version. When it says ALL PASS, run the model with your kernels so far.

```
# once, on the head node (mlp)
git clone \
  https://github.com/ed-aisys/edin-mls-26-autumn.git
cd edin-mls-26-autumn/hw1-asr
bash setup_env.sh

# every session: ONE GPU (gpu:1)
srun -p Teaching --gres=gpu:1 \
  --cpus-per-task=2 --mem=16G \
  --time=01:00:00 --pty bash
export PATH=/opt/cuda-12.8.0/bin:$PATH
conda activate mls-asr
cd ~/edin-mls-26-autumn/hw1-asr

# every task, e.g. task 2
make && make test2
./run.sh --cuda task1,task2

# all four done: the benchmark
./run.sh --profile
```

The code you will work with

Files (under hw1-asr/)	Responsibility	You
csrc/task1 … task4 (five files)	Your kernels: the blanks	**Edit; add helpers here**
csrc/common.cuh, asr_kernels.h	Shared helpers and the C interface	**Keep as given**
csrc/provided.cu	Finished kernels: embedding, silu_mul, im2col	**Keep as given**
glm_asr_cuda/ops.py, model.py	The model; calls your kernels via ctypes	**Keep as given**
benchmark.py, run.sh, tests/	Timing, the transcription check, kernel tests	**Keep as given**

Feel free to add any helpers, but do not change the original functions.

Keep every given function's name and signature, including the launch_* entry points that Python calls. No cuBLAS, cuDNN, CUTLASS, Thrust or PyTorch inside a task.

Subject to change. Please refer to the released codes.

The baseline: the same model on PyTorch

What it is

./run.sh --cuda none runs every operation with PyTorch instead of your kernels. Its matrix multiplies use cuBLAS, NVIDIA's tuned library.

Why you run it

First, to check your setup before you change any code. Then, as the reference your kernels are compared with.

How to read it

Time: mean ± std of 3 runs after 1 warm-up. Per token: the time divided by the 13 tokens. Accuracy: share of the expected words found; PASS above 80%.

./run.sh --cuda none (RTX 2080 Ti)

```
Time: 992.9 ms per
  transcription (+/- 4.8 ms),
  76.4 ms per token
Peak GPU memory: 8.88 GiB
Transcription: Concord
  returned to its place
  amidst the tents.
Accuracy: 100.0%
Status: PASS
```

Times differ a lot between GPU types: run the baseline on the same kind of GPU as your benchmark.

The tests and the benchmark

Command	What it runs	What it tells you
make test (or make test1 … test4)	Each kernel on the model's shapes, against a CPU version	PASS or FAIL per case; time, and GB/s or GFLOP/s
./run.sh --cuda task1,task2	The model with only these tasks on your kernels	Whether the transcription still passes
./run.sh --profile	The model on all your kernels, plus one profiled transcription	Time per transcription, GPU time per operation
./run.sh --cuda none --profile	The same on PyTorch	The baseline's GPU time per operation
sbatch run_asr.sbatch	make test and ./run.sh --profile as one batch job	Both outputs, in asr-JOB_ID.out

The profile table: one line per operation, with where it ran (cuda = your kernel, torch = PyTorch), its calls, total ms and share of the GPU time.

What you must compare, at least

Comparison	How	Report
Correctness	make test; ./run.sh	Every case PASS; transcription Status: PASS
Your kernels vs PyTorch	./run.sh --profile vs ./run.sh --cuda none --profile	ms per transcription and per token, peak memory, time per operation
Your GEMM vs naive vs cuBLAS	make test3 prints all three	GFLOP/s per case, and why cuBLAS is faster
Memory-bound kernels	make test1, make test2	GB/s as a share of the GPU's peak
One “Try this” per task	Change one thing (e.g. TILE, ATTN_WARPS), re-run the test	Numbers before and after, and why

Use the same kind of GPU for every number you compare (pin it with --gres) and report nvidia-smi -L.

What to hand in

Code	The five task files from csrc/, with any helpers you added in them.
Output	make test and ./run.sh --profile from the same GPU node, and nvidia-smi -L.
Report	Answers to the questions at the end of each guide, and at least one “Try this” experiment per task: the change, the numbers and your explanation.

Subject to change. Please refer to the released codes.

INFR11269 MACHINE LEARNING SYSTEMS

Coursework: RAG and systems investigation

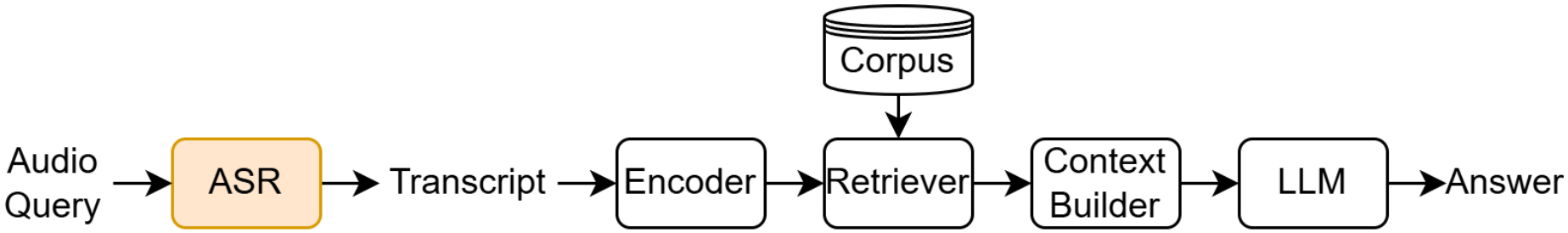


The pipeline, experiments and report
2026/27

The coursework task

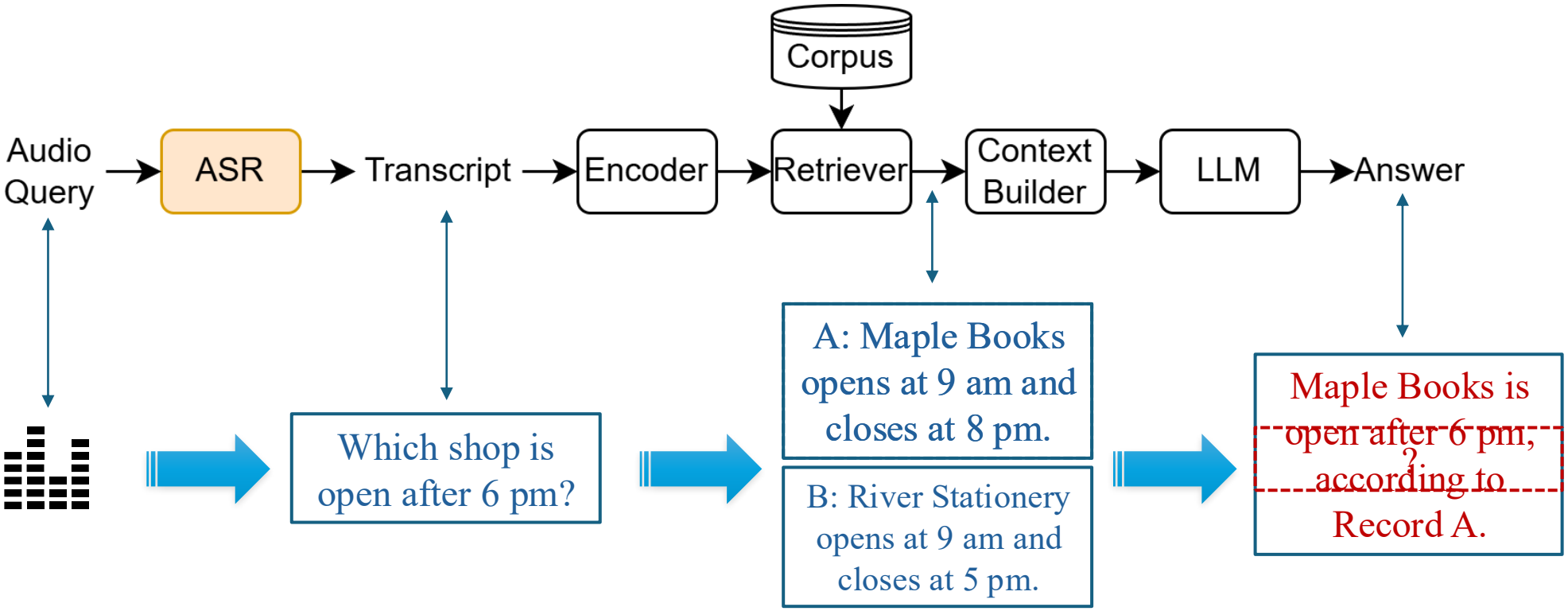
1. Common baseline	Understand the complete audio-to-answer workflow.
2. Technical investigation	Change a component to answer a precise systems question.
3. Experimental evidence	Compare quality with a relevant cost, using controlled conditions.
4. Report	Explain the method, results, trade-offs and limitations.

The common pipeline



Stage	Main operation	Output
1 ASR interface	Transcribe the audio query	Transcript
2 Local encoder	Encode the transcript with enc_a	Query vector
3 Exact retrieval	Rank corpus vectors by similarity	Top-k record IDs
4 Context builder	Fetch text and construct the prompt	Evidence + question
5 ELM generator	Generate an answer from the prompt	Answer + timings

A small RAG example



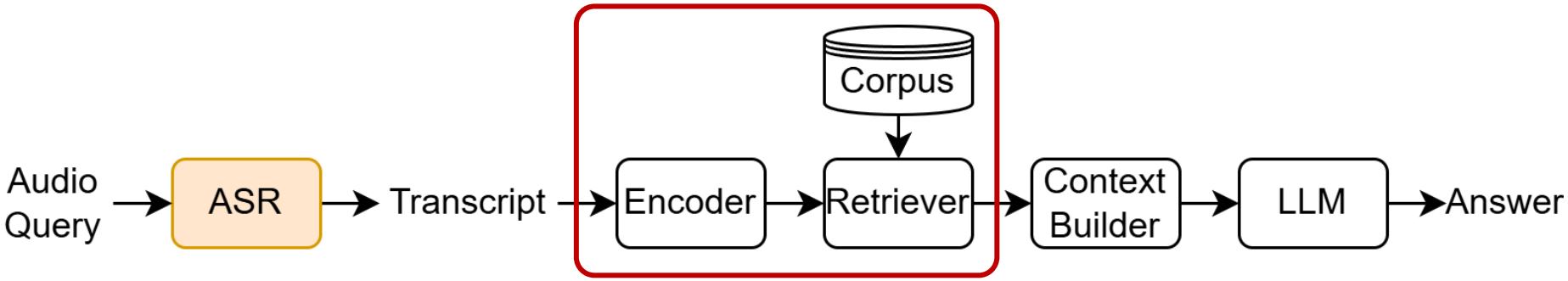
Fictional example.

The code you will work with

Files (under rag/)	Responsibility
src/data.py + src/encoder.py	Corpus access and local query encoding
src/retriever.py	Exact dense retrieval
src/context_builder.py + src/generator.py	Evidence assembly and ELM generation
src/rag.py	Pipeline orchestration and timings
scripts/	Run queries and evaluate experiments
extensions/	Optional place to put your additional codes

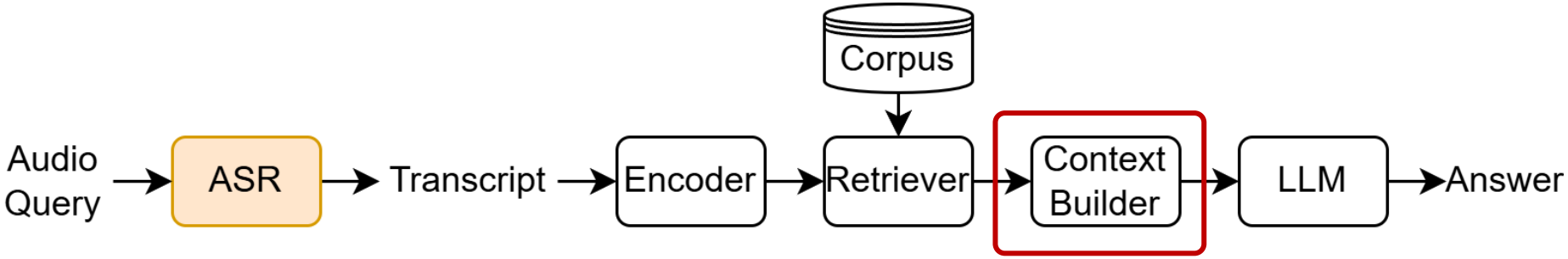
Subject to change. Please refer to the released codes.

Embeddings and exact retrieval



Corpus matrix	24,000 records × 384 dimensions
Similarity	Normalised query and corpus vectors use a dot product.
ID mapping	Each vector row maps to a corpus record ID.
enc_a and enc_b	Matching dimensions do not imply matching vector spaces.

Context construction and generation

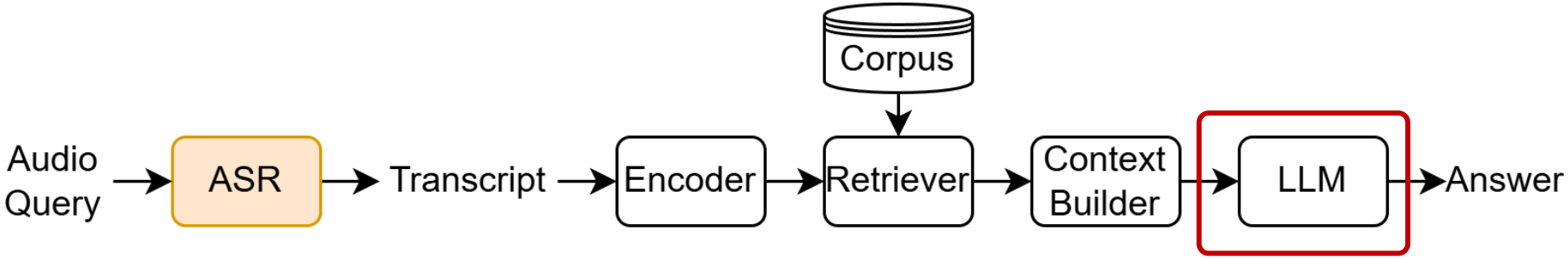


Question: <ASR transcript>

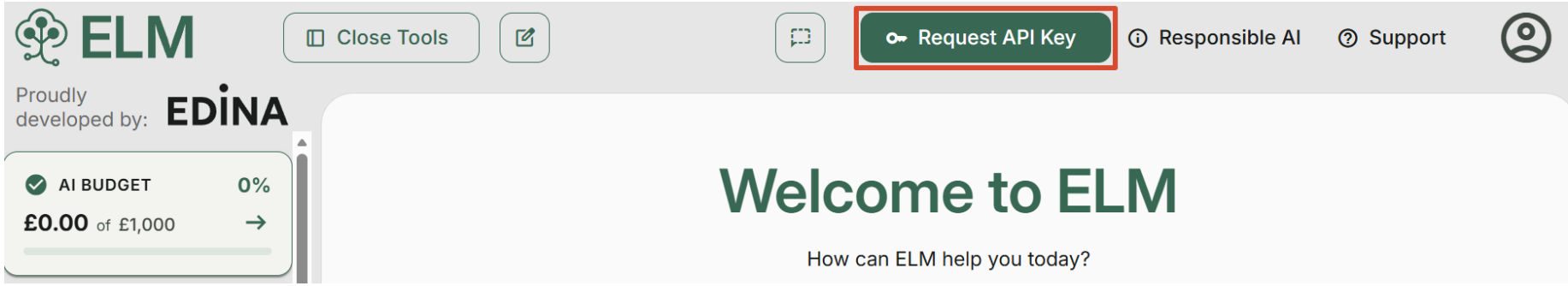
Evidence:
[Document 1] <retrieved text>
[Document 2] <retrieved text>

The baseline asks the model to use the evidence and acknowledge insufficient support.

Get your ELM API key



We will need one ELM API key to run the codes.
Go to https://elm.edina.ac.uk/elm using your university account and request an API key (if you don't have one already).



Get your ELM API key

You'll need to write about your intended use when requesting a key. Here's an example:

> I am taking INFR11269 Machine Learning Systems at the University of Edinburgh. I will use my ELM API key for the course's retrieval-augmented generation (RAG) coursework and related systems experiments.
>
> I intend to use the OpenAI-compatible Chat Completions endpoint (/chat/completions) through https://elm.edina.ac.uk/api/v1. Requests will support two activities: generating answers from retrieved coursework documents, and evaluating candidate answers for correctness, grounding and completeness.
>
> The workflow starts with course-provided benchmark audio. Speech recognition runs locally; ELM requests contain the resulting text transcript, selected documents from the supplied course corpus and, for evaluation, candidate answers. I do not intend to upload raw audio, confidential university information, credentials or personal student records. Public source text may still contain personal references or sensitive topics, so I will review the data I submit and follow the course's data-use guidance.
>
> I will keep my key in a local environment variable or an excluded .env file, and will not include it in shared code, repositories, reports or prompts. I will limit requests to coursework experiments, use caching where appropriate, monitor usage and avoid unnecessary repeated calls. I will treat generated answers and automated evaluation scores as fallible, inspect representative outputs and acknowledge my use of AI in accordance with the course's academic-integrity requirements.

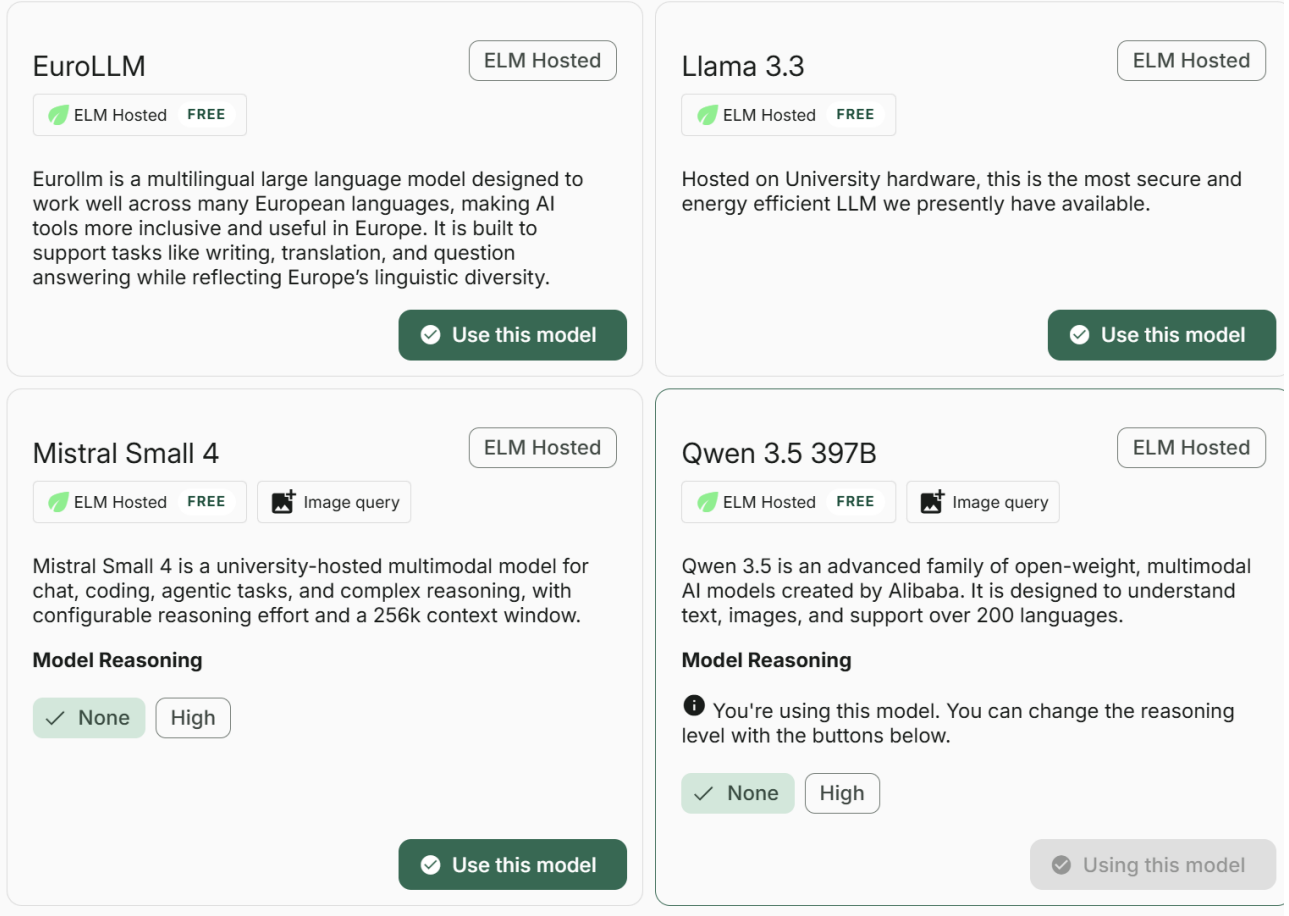
Please pay attention: you must follow the ELM rules! We won't take responsibility if there is any misconduct.

Get your ELM API key

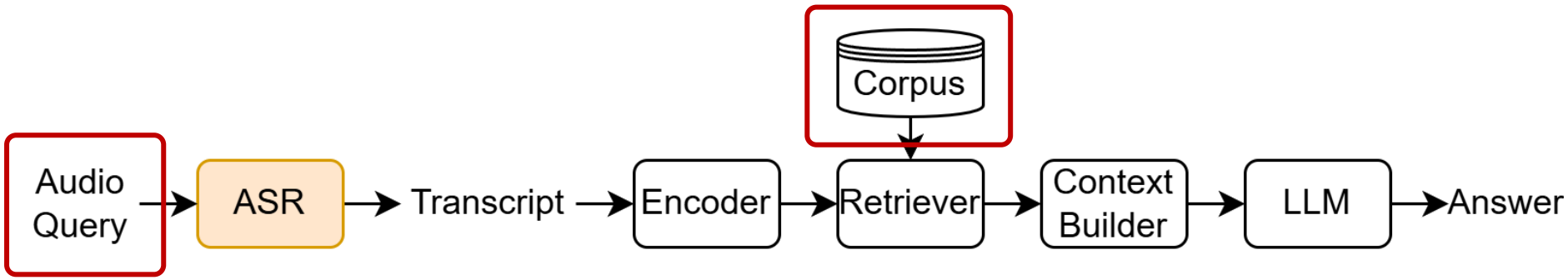
ELM provides numerous models. There are four locally-hosted models which are free to use. We recommend you use one of them. (The codes use one of them as default; no need to change it.)

Simply switching models does NOT count as a technical point.

Do not share your key with others or AI!



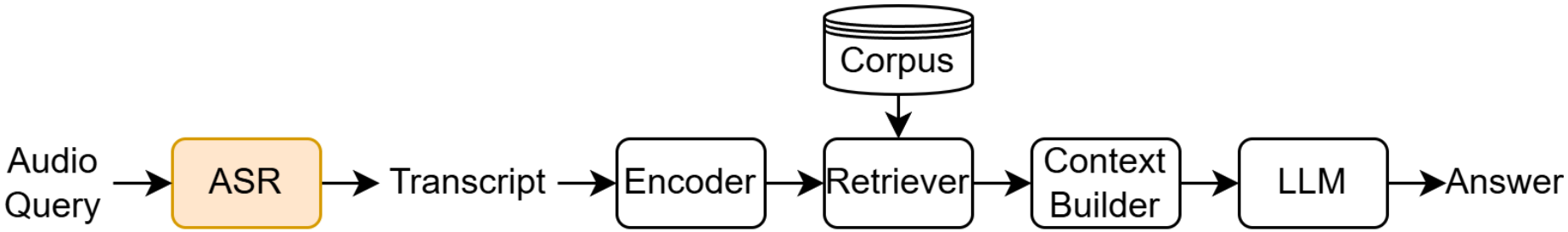
The supplied data



Asset	Purpose
corpus.jsonl	24,000 records with record IDs and text
audio/practice + audio/dev	350 practice and 125 development queries
student_audio_manifest.jsonl	Query IDs, split and audio paths
qrels_practice.tsv + qrels_dev.tsv	Public relevance judgements
embeddings/ + models/enc_a/	Corpus representations and baseline encoder

Subject to change. Please refer to the released codes.

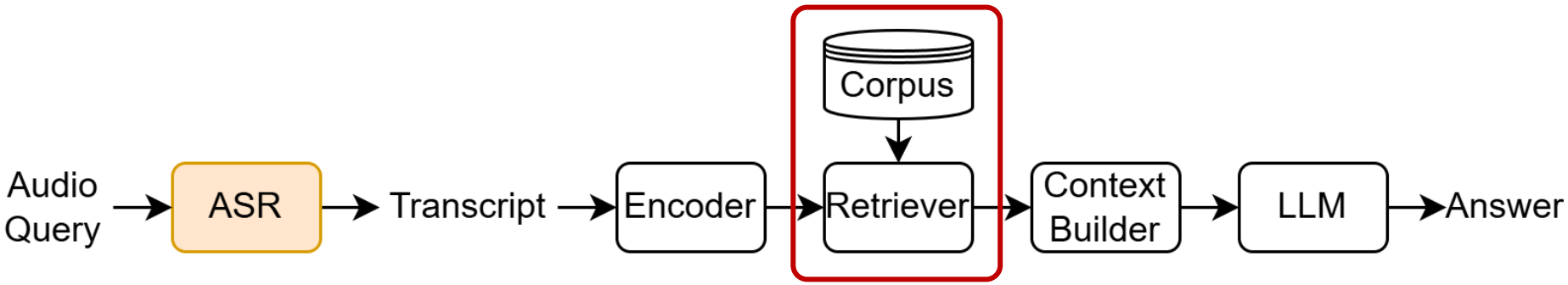
The run and evaluation workflow



Step	Command-line entry point
Check setup	python scripts/check_setup.py
Inspect one query	python scripts/run_audio_query.py --audio <WAV path>
Measure retrieval	python scripts/evaluate_rag.py --split practice
Generate candidates	python scripts/generate_answers.py --split practice --output <JSONL>
Validate / judge answers	python scripts/evaluate_answers.py --split practice --candidates <JSONL> --dry-run

Subject to change. Please refer to the released codes.
ASR integration and ELM configuration are prerequisites for the complete workflow.

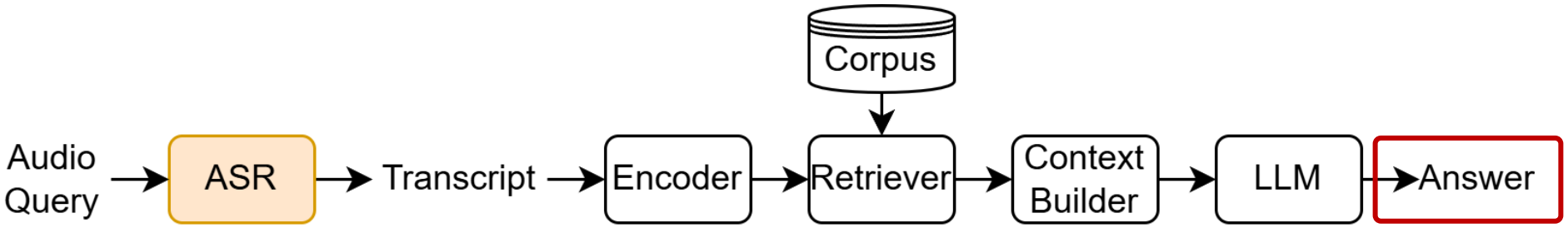
Retrieval quality measures



Measure	What it captures
Recall@k	Fraction of relevant records found in the first k results
MRR@10	Rank of the first relevant result, within the cutoff
NDCG@10	Ranking quality with relevance gains and rank discount
Toy example:	Two relevant records, one in top five: Recall@5 = 0.5

Retrieval quality and final answer quality measure different stages.

Answer evaluation – an evaluation aid



Dimension	Scale	Focus
Correctness	0–4	Factual and logical accuracy
Grounding	0–4	Support from relevant evidence
Completeness	0–4	Coverage of the answerable question
Combined score	0–1	Sum of the three scores divided by 12

Public judging uses the ASR transcript. It cannot verify that ASR preserved the intended question.
Use this as an evaluation aid only, since this is purely generated by LLM.

A fair systems experiment

How to find a good extension problem?

Comparison	State the baseline, intervention and expected trade-off.
Controls	Keep inputs and unrelated model settings consistent.
Measurements	Measure quality and the cost relevant to your question.
Reproducibility	Describe timing boundaries, hardware, repeats and caches.

Example of extension: retrieval depth

Design choice	Experiment plan
Variable	k = 2, 5, 10, 20 (baseline k = 5)
Controls	Same transcripts, encoder, corpus and generator settings
Quality	Recall@k and answer correctness / grounding / completeness
Cost	Context size and generation latency
Analysis	When does extra evidence help, and when does it add noise?

Other directions for investigation

Direction	Possible question
Batching	How does batch size affect throughput and query latency?
Context selection	Can an evidence budget preserve quality with smaller prompts?
Reranking	Does better evidence justify the additional stage cost?
Dynamic updates	What is the cost of keeping the retrieval index consistent?
Representation migration	How can a system transition between compatible representations?

You can also choose ASR optimisations as extensions.

The report’s chain of reasoning

Part of the argument	What the reader needs
Question	A precise problem and a reason it matters
Method	Baseline, technical change and comparison controls
Evidence	Quality and cost results that address the question
Interpretation	Why the behaviour occurs and which trade-off it reveals
Limitations	Conditions where the conclusion may not hold

Report format

Current guidance	Status
Template and style	Use the supplied LaTeX template and preserve its format
Section structure	Recommended headings are flexible
References and Appendix	Excluded from the configured main-body page limit
Main-body page limit	6 pages
Appendices and declaration pages	Final course policy requires confirmation

Subject to change. Please refer to the released codes.