ADDALY · THE AI CLASSROOM

Hugging Face, End to End

Read a model card licence-first, run a Space for free, keep a token safe, publish something of your own.

---

8 modules · 75 lessons · 28 figures · 8 case studies · 55,572 words · about 11 hours

Dr Mustafa Mun
Author and editor

ABOUT THIS BOOK

Hugging Face, End to End, published by Addaly, 2026. Author and editor: **Dr Mustafa Mun**.

The manuscript was drafted with the assistance of a large language model and was edited and published under his name. That is stated plainly here rather than left to be discovered, because a reader is entitled to know how a book they are trusting was made.

This book is the printed form of the course of the same name at addaly.com/learn/hugging-face. The website is the living version and is corrected first; where the two differ, the website is right.

Free to read, free to download, free to print, and free to teach from. There is no paid edition and there never will be. If you paid for this, somebody has sold you something that was given away.

Every diagram in it is drawn from data by code, not generated as a picture, so the numbers on the page are the numbers in the argument. Answers to every exercise, test and examination question are at the back, with a note on why each wrong option attracts people — those notes are the teaching, not the marking.

Contents

1. The Hub, read properly

Find a specific model on the Hub using filters and the Python API rather than the search box, read its repository files to say what architecture it is, what format its weights are in and what code will run when you load it, and explain what the download counter is actually counting

2. Transformers, from characters to output

Trace a piece of text from characters through a tokenizer to logits and back to text with the transformers library, and say which of the Auto class, the tokenizer, the padding side, the chat template or the generation config is at fault when a model loads without error and produces nonsense

1. One line that hides five steps
2. AutoModel gives you numbers, not answers
3. Your text is integers, and the count is not the word count
4. Padding on the wrong side ruins generation quietly
5. Logits, and the four knobs between them and words
6. The chat template is not optional
7. Throughput and latency are different problems
8. Showing tokens as they arrive, and making them stop
9. It loaded, it ran, the output is nonsense
10. Case study: The classifier that was ninety-nine per cent sure of nothing
11. Module test

3. Running it on the hardware you actually have

Compute from a model's parameter count and dtype whether it will fit the machine in front of you, pick between a full-precision, quantized, GGUF or ONNX route for a stated constraint, and measure peak memory, first-token latency and quality loss well enough to defend the choice

1. Your disk fills up before your patience does
2. The memory arithmetic you can do in your head
3. Spreading a model across whatever you have
4. Quantization: what you buy and what you pay
5. GGUF, and the laptop that has no GPU
6. ONNX, Optimum, and models that run in a browser tab
7. Where the free GPUs actually are
8. What actually runs on eight gigabytes
9. Proving the change helped
10. Case study: Eleven old machines against one new one
11. Module test

4. Spaces: putting something in front of people

Build, configure and debug a Space that other people can use — choosing the SDK, pinning versions in the README, holding secrets and state correctly, picking hardware against a sleep policy — and call an existing Space programmatically as an API instead of reimplementing it

1. A Space is someone else's computer, already configured
2. Forty lines from nothing to a public URL
3. Blocks, when Interface is no longer enough
4. Duplicate it, and the app becomes yours to change
5. The README front matter is the deployment config
6. Hardware, sleep, and the bill that runs while you are asleep
7. The filesystem disappears, and what to do about it
8. Streamlit, Docker and a Space with no server at all
9. Every Space is an API you can call
10. It built, it started, it is red
11. Case study: The draft that appeared in somebody else's browser
12. Module test

5. Datasets, and the work that happens before any model

Inspect, stream, filter and transform a dataset far larger than your RAM, query the Hub's parquet conversion with SQL before downloading anything, run duplicate, leakage and class-balance checks on your own data, and publish it with a card stating provenance and licence

1. Do not download the dataset
2. A dataset that behaves like a list and is not one
3. Why a 300 GB dataset opens instantly
4. Transforming a million rows without waiting an hour
5. Query a dataset with SQL before downloading it
6. Turning your files into a dataset
7. The card is where a dataset stops being a file
8. Media columns decode when you touch them
9. Duplicates, leakage, balance, and label noise
10. From a dataset to batches a trainer can eat
11. Case study: Ninety-four in the lab, sixty-one in the field
12. Module test

6. Embeddings, and search that understands

Build and measure a semantic search over your own documents — choosing an embedding model against your language and text length, chunking deliberately, normalising vectors for the similarity you use, adding a cross-encoder reranker, and reporting recall@k on questions you wrote yourself

1. A list of numbers where near means similar
2. The library, and the four calls you need
3. Picking the model, and reading MTEB without being fooled
4. Cosine, dot product, and the normalisation that connects them
5. Chunking decides what your search can find
6. NumPy first, FAISS second, a database rarely
7. The second pass that fixes the first
8. One space for pictures and words
9. Thirty questions, and the numbers they give you
10. Case study: The part number the search could not find
11. Module test

7. Training and adapting, with the libraries the Hub is built on

Decide from evidence whether a task needs fine-tuning at all, run a LoRA or QLoRA adaptation with Trainer or SFTTrainer on a single free GPU, read a loss curve well enough to name what is wrong, and publish an adapter a stranger can apply to the base model

1. Three cheaper things to try first
2. Trainer, and what it saves you writing
3. The six arguments worth understanding
4. LoRA: training 0.1% of the parameters
5. QLoRA, and fine-tuning a 7B model on a free GPU
6. TRL, and training on conversations
7. Watching a run you cannot sit next to
8. Loss is NaN, memory is gone, nothing is learning
9. Shipping the thing you trained
10. Case study: The fine-tune that was two different problems
11. Module test

8. Depending on it, and publishing so others can

State what a model's licence permits for your intended use and where the question is unresolved rather than answered, choose between a provider, a dedicated end-point and your own hardware on stated cost and privacy grounds, judge a model with thirty examples of your own, and publish work a stranger can audit and reproduce

1. A token in a public Space is a token you have given away
2. What the licence tag does and does not settle
3. The leaderboard is not measuring your task
4. The evaluation that decides it
5. Renting the compute instead of owning it
6. When transformers stops being the right server
7. Publish it so a stranger can use it without writing to you
8. Writing a card somebody can audit
9. Pull requests, discussions and being useful
10. Still working next year
11. Case study: Three weeks before launch, the base was not theirs to sell
12. Module test

Hugging Face, End to End — final examination

The paper that opens the next course. Answers to everything follow it.

MODULE 1

The Hub, read properly

The Hub holds over a million model repositories and the search box is the worst way into them. This block teaches the Hub as what it is — a git host with a filter sidebar, a file layout, a naming convention and a set of counters — so that you can find a specific model among millions, tell from its files alone what will load it, and know which of its numbers mean anything.

> BY THE END OF THIS MODULE YOU CAN
>
> Find a specific model on the Hub using filters and the Python API rather than the search box, read its repository files to say what architecture it is, what format its weights are in and what code will run when you load it, and explain what the download counter is actually counting

1 · 10 min

The Hub is git, and that explains everything else

THE ONE THING TO KEEP

A model on the Hub is a git repository with big files stored beside it, so it has versions, history and pull requests — and if you do not pin a revision, the model under your code can change without your code changing.

Most people use the Hub as a download button

Open a model page — `openai/whisper-large-v3`, say — and look at the tabs across the top: Model card, Files and versions, Community. Then look inside Files and versions. There is a commit history. There are branches. Under Community there are pull requests sitting alongside the discussion threads.

None of that is decoration. A Hugging Face repository **is** a git repository. The model card is a file called `README.md` inside it. The weights are files committed to it. The Hub is a git host that happens to be full of neural networks, and almost everything confusing about the platform stops being confusing once you hold that one fact.

What being git actually buys you

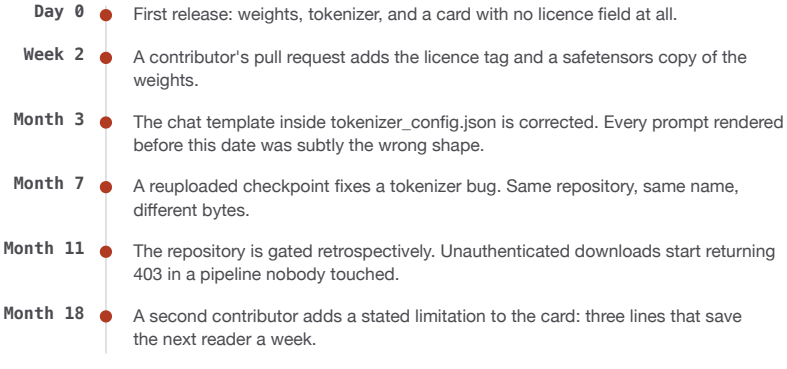
A model has versions, and `main` moves. When a lab fixes a tokenizer bug or reuploads a corrected checkpoint, the repository gets a new commit. If your code says `from_pretrained("some-org/some-model")`, it means *whatever is on main at the moment it runs*. That is fine while you are experimenting and a real hazard in something you have shipped, because your model can change without your code changing. Every loading function takes a `revision` argument, and it accepts a branch, a tag, or a commit hash:

```
from transformers import AutoModel

model = AutoModel.from_pretrained(
    "sentence-transformers/all-MiniLM-L6-v2",
    revision="PASTE_THE_COMMIT_HASH_FROM_THE_HISTORY_TAB",
)
```

Copy the hash from the History link in Files and versions. Pinning costs you nothing and removes a whole category of Monday morning.

Eighteen months on one model's main branch



None of these commits changed the model's name. Code that did not pin a revision picked up every one of them, the next time it ran, with no change of its own. The hash in the History tab is what makes a result reproducible.

You can read the diff. History shows what changed and when. If a model started behaving differently last week, that is the first place to look, and often the last.

Anyone can open a pull request. Someone adds a missing licence tag, converts weights to safetensors, translates a card into Hindi. On the Hub these live as real git refs like `refs/pr/4`, so you can download and test a proposed change before it is merged.

Nothing has to be public. A repository can be private, and every command below works the same way with a token.

The big files are not really in git

Git was built for source code. It handles a 5 GB tensor file badly. So the Hub keeps the *pointers* in git and the *bytes* in a separate large-file store — Git LFS originally, and since 2025 Hugging Face has been moving repositories onto its own chunk-level storage that deduplicates blocks, so reuploading a slightly changed checkpoint does not resend the whole thing. You do not need to know which one sits behind a given repo. You do need to know one consequence.

Run this on a machine that does not have Git LFS installed:

```
git clone https://huggingface.co/openai/whisper-small
```

You get a folder with all the right filenames and a `model.safetensors` that is about 130 bytes. It is not corrupt and the download did not fail. It is the pointer file — a few lines of text naming an object hash — because git faithfully cloned exactly what was committed, and nothing fetched the real object. People lose an hour to this and conclude the model is broken.

Three ways to actually get the files

The browser. Every file has a download link. On a phone this is the only route, and for one 200 MB file it is fine. Direct URLs follow a single pattern worth memorising: `https://huggingface.co/<owner>/<repo>/resolve/main/<filename>`.

The command line. Install the client and use it instead of raw git:

```
pip install -U huggingface_hub
hf download openai/whisper-small
```

The command was called `huggingface-cli` for years and was renamed to `hf` in 2025. If your shell says `hf: command not found`, upgrade the package, or use the old name — both are common in tutorials you will find. `hf download` resolves large files properly, resumes an interrupted transfer, and puts everything in a shared cache rather than a folder you will forget about. That last part matters more than it sounds; lesson seven is about the disk it fills.

Python, when you want one file. A repository often holds three copies of the same weights in different formats. You rarely want all of them:

```
from huggingface_hub import hf_hub_download

path = hf_hub_download("openai/whisper-small",
"model.safetensors")
```

For a subset, `snapshot_download` takes `allow_patterns` — passing `["*.safetensors", ".json"]` skips the duplicate PyTorch `.bin` files and can halve what you pull down.

Do this now

Pick any model you have used through somebody else's app. Open Files and versions, then History. Count how many times it has changed since it was announced, and read the message on the most recent commit. That commit is the version you have been using.

BEFORE YOU MOVE ON

Someone runs `git clone` on a model repository and ends up with a `model.safetensors` of about 130 bytes containing a few lines of text. What has happened?

A. The repository is gated and they were not authenticated, so the server returned a placeholder file instead of the weights.

B. The transfer was cut short by the network and needs to be resumed before the file is complete.

C. Git cloned the pointer file that stands in for the weights; the actual bytes live in a large-file store that Git LFS or `hf download` fetches separately.

D. The branch `main` had moved to a commit made before the weights were uploaded, so only the metadata existed at that revision.

2 · 9 min

The search box is the worst way in

THE ONE THING TO KEEP

Hub search matches repository names and card text rather than meaning, so the filter sidebar — task tag, library, licence, and the base-model links to fine-tunes and quantizations — is the real interface, and the sort order answers popularity rather than quality.

Typing what you want does not work

Go to the Hub and type `sentiment analysis` into the search box. You get repositories whose *names* contain those words. You do not get `cardiffnlp/twitter-roberta-base-sentiment-latest`, which is one of the most used sentiment models on the platform, unless the substring happens to match. Hub search is largely a text match over repository names and card content. It is not a semantic search over what models do.

This catches everyone once. The model you are looking for exists, has been downloaded four million times, and does not appear, because the person who uploaded it called it `distilbert-base-uncased-finetuned-sst-2-english` and never wrote the phrase you typed.

The filters are the real interface. The search box is for when you already know the name.

The filter that does most of the work

Every model repository carries a `pipeline_tag` in its card metadata — one value from a fixed list: `text-classification`, `automatic-speech-recognition`, `image-segmentation`, `text-generation`, and around fifty others. That tag is what the **Tasks** sidebar filters on, and unlike free text it is a controlled vocabulary. Pick the task, and you are looking at models that claim to do that job and nothing else.

The filters compose into the URL, which is worth knowing because it lets you keep a search as a bookmark or paste it to a colleague:

```
https://huggingface.co/models?pipeline_tag=automatic-speech-recognition&library=transformers&language=hi&sort=downloads
```

That one reads: speech recognition, loadable with transformers, tagged Hindi, most downloaded first. Four constraints, one line, and it narrows a million repositories to a screenful.

The filters worth reaching for, in the order they usually help:

- **Tasks** — the controlled vocabulary above.
- **Libraries** — `transformers`, `sentence-transformers`, `diffusers`, `gguf`, `onnx`. This is a filter on *what will load it*, which is often the real question.
- **Languages** — tagged by the uploader, so it is a claim rather than a measurement.
- **Licences** — `apache-2.0`, `mit`, `cc-by-nc-4.0`, and the bespoke ones. Filter here early if the work is commercial.
- **Other** — model size ranges, and the `base_model` relationships described below.

Sorting: trending is not quality

The sort control offers Trending, Most likes, Most downloads, Recently created and Recently updated. Each answers a different question, and none answers "which is best".

- **Most downloads** is a thirty-day window, and it favours models that libraries load by default. It tells you something is safe and widely exercised. It tells you nothing about whether it suits your language or your text length.
- **Trending** weights recent activity, so it surfaces last week's release. Useful for knowing what happened while you were on holiday, useless for picking a production dependency.

- **Recently updated** is the one people forget, and it is how you find out whether a model still has a maintainer.

Finding the variants of a model you already have

This is the trick that saves the most time. A model card declares its parent with a `base_model` field, and the Hub indexes that relationship in both directions. On any base model's page there is a line reading something like *"Fine-tunes: 3,417 · Quantizations: 891 · Adapters: 2,104"*, and each is a link.

So when you have found `meta-llama/Llama-3.1-8B-Instruct` and your laptop cannot run it at full precision, you do not search for "llama 8b small". You click **Quantizations** and get every GGUF, AWQ and GPTQ conversion anyone has published, in one filtered list. The same route finds the adapter someone already trained for your language, or the fine-tune already specialised for medical text.

The dataset and Space filters are the same idea

Datasets filter by task, size category, language, licence and format. The size filter is a band — `1K<n<10K`, `10M<n<100M` — and it is populated automatically from the dataset viewer rather than claimed by the uploader, which makes it one of the more trustworthy facets on the site.

Spaces filter by SDK (`gradio`, `streamlit`, `docker`, `static`) and by hardware. Filtering Spaces to `gradio` and sorting by likes is the fastest way to find a working reference implementation of almost any task, because a Space that runs is a configuration that works.

Full-text search, when you need it

There is a full-text option that searches inside model cards rather than only names. It is reachable from the search results page and it is genuinely useful for phrases that only ever appear in prose — a benchmark name, a paper title, a specific dataset. Searching card text for `WER` and filtering to speech recognition finds the models whose authors actually reported an error rate.

Its limitation is the same one that afflicts the whole site: a card is a claim written by the uploader. Searching for a phrase finds people who wrote the phrase, not people whose model is good at the thing.

Do this now

Pick a task you care about. Build the URL by hand with three filters — task, library, licence — and sort by downloads. Then open the third result rather than the first, and look at when it was last updated. The gap between the ranking and the maintenance is the thing most people never check.

BEFORE YOU MOVE ON

A team needs a 4-bit version of a model they have already chosen so it fits on a laptop. What is the reliable way to find one on the Hub?

A. Search for the model name plus a term like `4bit` or `quantized`, since conversions almost always put that in the repository name.
B. Open the base model's page and follow its Quantizations link, which indexes every published conversion.
C. Filter the models list to the `gguf` library and sort by downloads, then pick the highest-ranked result.
D. Use full-text card search for the model name, because conversions cite their parent in the card prose.

3 · 11 min

Read the licence before you read the demo

THE ONE THING TO KEEP

Check the licence family and the base model before anything else, because a fine-tune inherits its parent's terms no matter what its own card claims.

The order everyone reads it in is wrong

Most people open a model page, look at the sample outputs, decide they like it, and start building. The licence gets checked at the end, if at all — usually the week before launch, by someone who is now unhappy.

A model card is a `README.md` with a YAML block at the top. That block carries the machine-readable facts: `license`, `base_model`, `datasets`, `pipeline_tag`, `language`, `tags`. The prose underneath is written by whoever uploaded the model, and it is a mixture of documentation and advertising. Read the page in this order instead.

One: the licence, and which family it belongs to

The licence shows in the right-hand sidebar and in the front matter. There are roughly five families and they behave very differently.

- **Permissive** — Apache-2.0, MIT, BSD. Commercial use, modification, redistribution, keep the notice. The easy case.
- **Share-alike and copyleft** — anything CC-BY-SA, occasionally GPL. Your derivative may have to carry the same terms. That bites when you fine-tune.
- **Non-commercial** — `cc-by-nc-4.0`, `cc-by-nc-sa-4.0`, and several custom research licences. Excellent models sit here. You may not put them in a paid product, and *my app is free but carries ads* is not a question the

card answers. Image people meet this constantly: `sd-turbo` is the fastest thing you can run on a laptop CPU and its licence is non-commercial.

- **Custom community licences** — Llama, Gemma and others. Mostly permissive in practice, with named conditions: an acceptable-use policy, an obligation to display attribution such as *Built with Llama*, a naming rule for derivative models, and in Llama's case a clause that only engages above 700 million monthly active users. Read one properly once; they are short.
- **`other`, or no licence field at all.** Treat that as *unlicensed*, not as *free*. Absence of a licence is not permission.

Two traps. **Gated is not a licence**: clicking *Agree and access repository* gets you the files, and the licence still governs what you may then do with them. And **the model's licence and its training data's licence are separate questions** — a permissively licensed model trained on scraped material does not settle the second one for you.

Two: lineage, because you inherit it

The `base_model` field links a fine-tune to its parent, and the Hub draws the tree from it. This is the most useful thirty seconds on the page. A model presented as an original release is often three fine-tunes downstream of something whose terms, biases and context limit it silently inherits. If a card claims Apache-2.0 while its base model is under a non-commercial research licence, the card is wrong and the base wins. That happens more often than you would like, particularly on community merges.

Three: the numbers, and the sentence next to them

An evaluation table tells you what the author chose to measure. Ask three things of any of them.

- **Which benchmarks are missing?** Six results with three obvious ones absent is a report, not an oversight.
- **In what language and what format?** Nearly all famous scores are English multiple-choice. If your work is Hindi customer messages or

Yoruba speech, the number is close to meaningless for you.

- **Compared with what, and when?** Beating a 7B model is not the claim it sounds like if the comparison is two years old.

Lesson nine is entirely about this, and /learn/evaluating-ai takes it further.

Four: the limitations section

Scroll to *Bias, Risks and Limitations*. Most people skip it because it reads like boilerplate. It is usually where the author has written down, in plain language, the exact failure you are about to have: the languages it degrades on, the groups it misclassifies, the input length past which it falls apart, the fact that the speech model was trained on clean read audio and collapses on a phone call recorded in a market.

A card with no limitations section is not a model without limitations.

Five: the Files tab, which cannot lie

The prose is written. The files are what exists.

- **Size arithmetic.** Parameters times bytes per parameter. A 7-billion-parameter model at 16-bit is about 14 GB; at 8-bit about 7 GB; at 4-bit about 4 GB. Add the file sizes up and check them against what the card implies.
- **Prefer `safetensors` to `.bin` or `.pt`.** The older PyTorch formats serialise with Python's own object format, and loading one can execute code embedded inside it. The safetensors format exists so that loading weights is only loading weights. If a repo offers both, take the safetensors.
- `trust_remote_code=True` in the usage snippet means the repo ships custom Python that will run on your machine. Sometimes genuinely necessary. Always worth knowing you agreed to it.
- `config.json` states the real architecture, hidden size and context length, whatever the prose above claims.

Do this now

Open the last model you used and answer three questions without scrolling to the demo: what is its licence, what is its base model, and name one stated limitation. If you cannot find all three, you have already learned something about that model.

BEFORE YOU MOVE ON

A team wants to sell a product built on a fine-tuned model. Its card says `license: apache-2.0`, but the `base_model` field points at a model released under a non-commercial research licence. What is the accurate reading?

A. The fine-tune is a new work with its own weights, so the uploader's choice of Apache-2.0 governs and commercial use is fine.

B. The non-commercial terms of the base flow down to the derivative, so the Apache-2.0 label on the card does not make it safe to sell.

C. Because the repository is not gated, no acceptance step was required, which means no restrictive terms are in force.

D. The conflict makes the licensing void, so the ordinary copyright default applies and the weights can be used freely.

4 · 8 min

A model's name is a specification, if you can read it

THE ONE THING TO KEEP

A repository name usually encodes owner, family, parameter count, tuning style and weight format, which is enough to compute memory and rule a model in or out — but it never encodes context length or licence, and a mixture-of-experts name like 8x7B states resident memory rather than compute.

Nine fields in one string

Here is a repository identifier you will meet:

```
Qwen/Qwen2.5-7B-Instruct-GGUF
```

Unpacked, that is: owner `Qwen`, family `Qwen2.5`, roughly seven billion parameters, tuned to follow instructions, converted to the GGUF file format. Four of those five facts change what you can do with it, and all four are in the name. Learning to read the convention saves you opening twenty repositories to find out they were the wrong shape.

The convention is a habit, not a rule. Nobody validates it. But it is followed often enough that a name that breaks it is itself a signal — usually that the uploader is new, occasionally that they are hiding something.

Parameter counts, and the arithmetic they imply

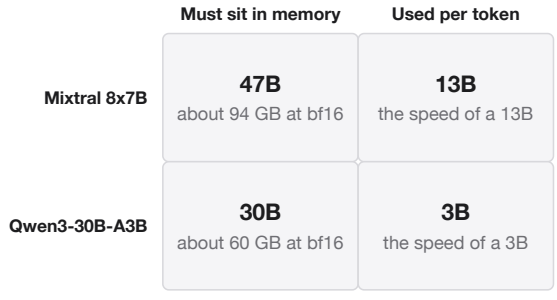
`7B` means about seven billion parameters. `0.5B`, `1.5B`, `3B`, `8B`, `14B`, `32B`, `70B` and `405B` are the common rungs. The number is the single most decision-relevant token in the name, because it converts directly into memory:

> ***bytes ≈ parameters × bytes per parameter***

At bfloat16 that is two bytes each, so a 7B model is about 14 GB of weights. At 4-bit quantization it is roughly 3.5–4.5 GB. Add a gigabyte or two for the activations and the key-value cache during generation, and you can decide in your head whether a model fits a free Colab T4 (16 GB) or your laptop (often 8 GB of usable RAM) before you click anything.

Two naming patterns hide a different arithmetic:

What a mixture-of-experts name is actually stating



People size an 8x7B as a 13B because that is the speed they experience, and then run out of memory. The first column is what you must fit; the second is what you get for it. A name like 30B-A3B is the honest version of the same arithmetic, because it prints both numbers.

- `8x7B` is a mixture of experts. It does *not* mean 56 billion parameters of compute; it means roughly 47 billion parameters must be **resident in memory** while only about 13 billion are used per token. It is fast for its size and large for its speed. People size it as a 13B and run out of RAM.
- `A3B` or similar, as in `Qwen3-30B-A3B`, states both numbers explicitly: 30 billion total, 3 billion active. This is the honest version of the same naming problem.

Base, instruct, chat, and why it matters more than size

- **No suffix, or `-Base`** — the model was trained to continue text and nothing else. Ask it a question and a good base model will often produce three more questions, because that is what a page of questions looks like. This is not a fault.
- **`-Instruct`, `-it`, `-Chat`** — the model has been tuned to treat the input as an instruction or a conversation. It expects its own chat template, and

using the wrong one degrades output in ways that look like the model being stupid rather than misused.

- `-Coder`, `-Math`, `-Vision`, `-VL` — specialised, usually at some cost to general ability.
- `-Reasoning`, `-R1`, `-Thinking` — trained to emit a long internal working-out before its answer. Budget for many more output tokens than you expect, because you pay for the thinking.

The practical rule: for anything you would phrase as a request, you want an instruct or chat variant. Choosing a base model by mistake is the single most common reason a beginner concludes that open models are bad.

Format and quantization suffixes

- `-GGUF` — converted for `llama.cpp` and the tools built on it, which run well on CPU and on Apple silicon. Files inside are further labelled `Q4_K_M`, `Q5_K_S`, `Q8_0`: the digit is bits per weight, the letters are the variant of the scheme. `Q4_K_M` is the common default — about a quarter of the memory of fp16 for a quality loss most people cannot detect in ordinary use, and can detect on code and arithmetic.
- `-AWQ`, `-GPTQ` — 4-bit schemes aimed at GPU serving rather than CPU.
- `-ONNX` — converted to a runtime that works in browsers and on ordinary CPUs.
- `-bnb-4bit` — pre-quantized with bitsandbytes, loadable directly by transformers.

Distil, mini, tiny, turbo

`distil` has a specific meaning — a smaller model trained to imitate a larger one, as in `distilbert` or `distil-whisper`. `mini`, `tiny`, `small`, `nano` and `turbo` have no agreed meaning at all and are chosen by whoever uploaded the file. Check the config, not the adjective.

The two things the name never tells you

Context length. A 7B model might accept 4,096 tokens or 128,000. That number lives in `config.json` as `max_position_embeddings`, and it is often the constraint that decides your project.

Licence. Nothing in the name is load-bearing here. `Llama` in a name does not tell you which Llama licence applies, and a fine-tune of a non-commercial model is still non-commercial no matter what its own card says.

Do this now

Take three models you have heard of and write out, from the name alone, the parameter count, whether it is instruction-tuned, and the memory it needs at bf16 and at 4-bit. Then open each config to check what you could not have known — context length — and notice how often that, not size, is the deciding number.

BEFORE YOU MOVE ON

Somebody sizes a machine for `mistralai/Mixtral-8x7B-Instruct-v0.1` by reasoning that only about 13B parameters are used per token, so a 16 GB GPU at 4-bit should be comfortable. What is wrong with the reasoning?

A. Mixture-of-experts models cannot be quantized to 4 bits, so the memory estimate should have used bfloat16 throughout.

B. The 8x7B naming means 56 billion parameters, so every figure in the estimate is too small by a factor of four.

C. All experts stay resident in memory even though only two run per token, so it needs about 47B parameters' worth of space.

D. The instruct suffix adds a second copy of the weights for the chat head, which doubles the memory an equivalent base model would need.

5 · 10 min

What the files in a model repository are for

THE ONE THING TO KEEP

Five files answer the questions a model card often does not: config.json gives the architecture and context length, the tokenizer files carry the chat template, the safetensors index gives the true download size, generation_config sets sampling defaults you did not choose, and an adapter_config means you are looking at a LoRA rather than a model.

Open Files and versions before you open anything else

A typical transformers model repository contains eight to fifteen files. Five of them decide whether the model will work for you, and you can read all five in a browser in under a minute — no download, no GPU, no install. This is the cheapest diagnostic on the Hub and almost nobody uses it.

Here is what you are looking at.

`config.json` — the model's shape

This is the architecture, in JSON. A trimmed example:

Five files, and the question each one answers

config.json	Architecture, context length, saved dtype, vocabulary size
tokenizer_config.json	The chat template, the padding side, the special tokens
model.safetensors.index.json	total_size: the honest disk bill, before duplicate formats
generation_config.json	The temperature and do_sample you did not choose
adapter_config.json	Not a model: a LoRA, and the exact base it needs

All five are readable in a browser in under a minute, with no download, no GPU and no install. Four of them answer questions the prose card usually does not, and the fifth tells you that what you are looking at is not a model.

```
{
  "architectures": ["LlamaForCausalLM"],
  "hidden_size": 4096,
  "num_hidden_layers": 32,
  "num_attention_heads": 32,
  "max_position_embeddings": 131072,
  "vocab_size": 128256,
  "torch_dtype": "bfloat16",
  "rope_theta": 500000.0
}
```

Four fields earn your attention:

- **`architectures`** names the Python class that will be instantiated. `LlamaForCausalLM` tells you this is a text generator; `BertForSequenceClassification` tells you it is a classifier with a fixed number of labels. If this names a class your installed transformers version does not have, you get `KeyError` or a message about an unrecognised model type, and the fix is upgrading transformers rather than anything to do with the model.
- **`max_position_embeddings`** is the context length. This is the number the model name never tells you and the one that most often decides a project.
- **`torch_dtype`** records what precision the weights were saved in. It is a record, not an instruction: loading does not automatically honour it in older versions of transformers, which is how people end up holding a bfloat16 model in float32 and using twice the memory they budgeted.
- **`vocab_size`** must match the tokenizer. When it does not — usually after somebody added special tokens and did not resize the embedding matrix — you get an index error deep inside the forward pass, or silent nonsense.

You can read it without a browser:

```
curl -s https://huggingface.co/meta-llama/Llama-3.2-
1B/resolve/main/config.json | head -40
```

The tokenizer files — three of them, doing different jobs

- **`tokenizer.json`** is the fast tokenizer: the full vocabulary and merge rules in one file, loaded by the Rust implementation. Its presence is why `AutoTokenizer` is fast.
- **`tokenizer_config.json`** holds the settings around it — padding side, maximum length, the special token names, and, on chat models, the **`chat_template`**. That template is a Jinja string, and it is the difference between a chat model working and a chat model producing drivel. It is worth reading.
- **`special_tokens_map.json`** maps roles like `bos_token`, `eos_token`, `pad_token` onto actual strings.

Older repositories carry `vocab.txt`, `merges.txt` or `spiece.model` instead. They work; they load through the slow Python path unless a converter runs.

The weights, and why there are several files

`model.safetensors` is the whole thing when the model is small. Above a few gigabytes it is sharded:

```
model-00001-of-00004.safetensors
model-00002-of-00004.safetensors
model-00003-of-00004.safetensors
model-00004-of-00004.safetensors
model.safetensors.index.json
```

The index is a map from every parameter name to the shard that holds it, plus a total byte count. It is small, it is readable, and the `total_size` field in it is the fastest honest answer to "how much disk will this cost me" — more reliable than adding up what the file listing shows, because the listing also contains formats you will not download.

Many repositories carry **both** `.safetensors` and `.bin` copies of the same weights. Downloading the repository naively fetches both and doubles your bill. The next lesson explains why you want the safetensors ones specifically.

`generation_config.json` — the defaults you did not choose

For generative models this file sets `temperature`, `top_p`, `do_sample`, `max_new_tokens`, `eos_token_id` and friends. When people compare two models and one seems wilder, the cause is frequently that their published generation defaults differ, not that the model does. If you are benchmarking, set these explicitly rather than inheriting them.

`.gitattributes`, and files with no purpose

`.gitattributes` tells git which patterns go to large-file storage. You will never edit it by hand and you should never delete it. Beyond that you will find `README.md` — which is the model card — plus example images, an `onnx/` folder, and sometimes a `runs/` directory full of training logs somebody forgot to exclude. That last one is occasionally the most informative thing in the repository.

A repository can also be an adapter

If you see `adapter_config.json` and an `adapter_model.safetensors` of a few dozen megabytes, this is not a model. It is a LoRA adapter — a small set of extra matrices that must be applied on top of the base model named inside `adapter_config.json`. Downloading it alone and wondering why nothing loads is a rite of passage.

Do this now

Open any chat model's `tokenizer_config.json` and find the `chat_template`. Read the Jinja. You are looking at the exact strings the model was trained to expect around a user turn — and once you have seen them, the whole category of "the model ignores my system prompt" bugs becomes legible.

BEFORE YOU MOVE ON

A repository lists ten files totalling 46 GB, but its `model.safetensors.index.json` reports a `total_size` of about 16 GB. What is the most likely explanation?

A. The index records the size after compression, and the listing shows the uncompressed files that will land on disk.

B. The repository holds the same weights in two formats, and the index counts only the safetensors shards.

C. The index is stale because it was written before the final training shards were uploaded.

D. The extra 30 GB is the optimiser state, which the index deliberately excludes because it is not needed for inference.

6 · 9 min

Loading a model can run someone else's code

THE ONE THING TO KEEP

A pickle checkpoint executes code when loaded, while safetensors is a JSON header plus raw bytes with no code path at all — but `trust_remote_code=True` reopens the hole by importing Python from the repository, so the weights format and the loading code are two separate decisions.

The thing nobody tells beginners

A PyTorch `.bin` or `.pt` checkpoint is a Python **pickle**. Pickle is not a data format in the way JSON is a data format. It is a small instruction stream for rebuilding Python objects, and among its instructions is one that calls an arbitrary function. Unpickling therefore executes code by design.

So `torch.load("pytorch_model.bin")` on a file from a stranger is, in the general case, running a program that stranger wrote, with your user's permis-

sions, on your machine. Not a theoretical risk: malicious models have been found on the Hub, and the usual payload is boring and effective — read environment variables, find the tokens, send them somewhere.

This is the reason safetensors exists, and it is why the format is worth understanding rather than merely preferring.

What safetensors actually is

A `.safetensors` file has two parts:

1. A header: a JSON object listing every tensor, its dtype, its shape, and the byte offsets where its data begins and ends.
2. The raw bytes of the tensors, one after another.

That is the whole specification. There is no instruction stream, no object graph, no code path that can be made to call a function. Loading it parses JSON and then points at memory.

Two consequences follow from that design, and both are practical rather than security-flavoured:

- **It is faster.** Because tensors sit at known offsets, the loader can memory-map the file and hand the framework a view of it rather than copying. On large models this turns a minute into a few seconds, and it is why cold starts improved across the ecosystem when repositories converted.
- **You can read one tensor without reading the file.** The header tells you offsets, so a tool can fetch a single layer over HTTP with a range request. That is how the Hub renders a tensor listing in the browser for a 70 GB model without downloading it.

```
from safetensors.torch import load_file
weights = load_file("model.safetensors")   # no code executes
```

Prefer safetensors, and say so in code

Transformers picks safetensors automatically when both formats are present. When you want the guarantee rather than the default, ask for it:

```
model = AutoModel.from_pretrained("some-org/some-model",
use_safetensors=True)
```

If that raises, the repository has no safetensors copy, and you now have a decision to make rather than a silent risk. Many older repositories have one because a community contributor opened a conversion pull request — check the Community tab before you conclude there is none.

The hole safetensors does not plug: `trust_remote_code`

Some models use architectures that are not in the transformers library. Their repositories ship their own Python — `modeling_custom.py`, `configuration_custom.py` — and loading them requires:

```
model = AutoModel.from_pretrained("some-org/new-architecture",
trust_remote_code=True)
```

That flag means exactly what it says. It downloads Python from the repository and imports it. Safetensors weights do not protect you here at all, because the danger is no longer in the weights file.

The flag is not avoidable — genuinely new architectures need it for months before library support lands — so treat it as a decision with steps:

- Read the `.py` files in Files and versions first. They are usually a few hundred lines of ordinary modelling code. Anything doing network calls, subprocess work or environment reads is a stop.
- Pin the revision, so a later commit cannot change the code under you: `revision="<commit hash>"` alongside `trust_remote_code=True`.
- Prefer well-known organisations for this specifically, not because large labs are virtuous but because their repositories are watched by thousands

of people.

- Run it in a container or a throwaway Colab session when you can, with no tokens in the environment.

The Hub's own scanning, and its limits

Uploads are scanned: a pickle scanner that inspects the opcode stream for dangerous imports, plus third-party malware and secret scanning, and the results appear on the file listing. This is real and it catches real things.

It is also a heuristic on an adversarial problem. A scanner that lists suspicious imports can be evaded by an attacker who reaches the same call through an indirection the scanner does not model. Treat a clean scan as evidence, not as a guarantee, in the same way you treat an antivirus pass on a downloaded installer.

The five-second habit

Before loading anything from an account you do not recognise:

1. Are there `.safetensors` files? Use them.
2. Does loading need `trust_remote_code`? If yes, read the Python.
3. Is there a pinned revision in your code? If no, add one.
4. Are there tokens in this environment? If yes, and you are unsure, move to a session without them.

Do this now

Find a repository that requires `trust_remote_code=True` and open its `modeling_*.py` in the browser. Read the imports at the top. That list — what the file is allowed to reach for — is the entire security review most of the time, and it takes about thirty seconds.

BEFORE YOU MOVE ON

A team standardises on `use_safetensors=True` for every model they load, and also loads a new vision-language architecture with `trust_remote_code=True`. What risk remains?

A. None in practice — the Hub's pickle scanner covers repository Python files, so a clean scan plus safetensors closes both paths.

B. Safetensors files can still carry executable payloads in their JSON header metadata, which the loader parses before reading tensors.

C. The remote code is imported and run regardless of weight format, with the user's full permissions.

D. Mixing the two flags forces transformers to fall back to the pickle loader, silently undoing the safetensors preference.

7 · 8 min

Four shelves, and picking the wrong one costs a weekend

THE ONE THING TO KEEP

Models are parts, datasets are material, Spaces are finished tools and inference is rented hardware — choose the shelf by how often the job runs and whether the data may leave your machine.

The question that decides everything

The Hub holds four kinds of thing, and beginners lose days by reaching for the wrong one. A creative person who wants forty photographs cut out downloads a 7 GB model and installs Python for two days. A developer who needs a background job run nightly opens a browser app and clicks four hundred times. Both are working hard in the wrong place.

The four shelves

Models are weights plus configuration. A model is inert. It does nothing until some code loads it, so a model page is a parts bin, not a tool.

Datasets are the data — text, images, audio, labels — with an automatic preview table and, for public datasets, an automatic conversion to a queryable format. Lesson eight is about using them without downloading them.

Spaces are running applications. Someone has written the code, listed the dependencies and pointed it at a model, and the Hub runs it on a URL. This is the shelf that matters most if you do not write code, and lessons four and five are about it.

Inference is somebody else's hardware running a model for you over an API. Two shapes:

- **Inference Providers** route your request out to partner companies — the roster has included Together, Fireworks, Replicate, Groq, Cerebras, SambaNova and others, and it changes. One API key reaches many models, and you pay per token or per image. Free accounts get a small monthly credit; a PRO subscription, around nine dollars a month as of late 2025, raises it. Check current prices rather than trusting that figure.
- **Inference Endpoints** are a dedicated container on a GPU you rent by the hour. Right when you need consistent latency or isolation, wrong for occasional use, because it bills while idle unless you configure it to scale to zero.

Choosing, in one pass

- You want a job done a handful of times, on your own files, today → **a Space**.
- You want the same job done four hundred times, on a schedule → **a provider API**, or a script running the model locally.
- The material must not leave your machine — client photographs, medical notes, a customer's confidential documents → **local**, and nothing else. This is not a preference. Uploading is a disclosure.

- You want to show a colleague, or change how the tool behaves → **duplicate a Space**.
- You want to train or fine-tune → **a dataset plus a GPU**, which does not have to be a paid one. Kaggle notebooks give roughly thirty GPU-hours a week free once you verify a phone number, and Google Colab's free tier gives a smaller, interruptible allocation. Both are real options if a rented GPU is out of reach.

The part nobody mentions about public Spaces

A public Space is arbitrary code, written by a stranger, running on a machine you do not control, and whatever you upload passes through it. Most authors are decent and most Spaces are thin wrappers around a model. But the honest position is that you cannot know, and the thing that makes the Hub unusual is that you can go and look: the Files tab of a Space shows `app.py`. Read it. It is normally under two hundred lines.

For anything sensitive, duplicate the Space into your own account so the code runs under your control, or run the model locally. The next two lessons cover the first; lesson seven covers the second.

Do this now

Take the task actually in front of you and say out loud which of the four shelves it belongs on, and why the other three are wrong. If the answer is *local because the files are confidential*, skip ahead to lesson seven before you upload anything.

BEFORE YOU MOVE ON

A freelance retoucher has sixty client photographs under a confidentiality agreement, a laptop with no GPU, and no Python installed. She needs the backgrounds removed. What is the right route?

A. Use a popular public background-removal Space, since sixty images is well within the free CPU tier and needs no installation.
B. Call an Inference Provider API, because per-image billing is cheaper than renting hardware for a batch this small.
C. Rent an Inference Endpoint on a GPU for an hour, since a dedicated container keeps the work isolated and fast.
D. Run a background-removal model locally on the CPU, because the confidentiality agreement rules out uploading and this task runs acceptably without a GPU.

8 · 9 min

Ask the Hub questions instead of browsing it

THE ONE THING TO KEEP

`huggingface_hub` turns the Hub into a queryable service, and the single highest-value call is `model_info(..., files_metadata=True)`, which tells you exactly what a repository will cost in disk and what formats it duplicates before you spend a byte.

The website is one client of an API

Everything the Hub shows you is available as JSON, and the `huggingface_hub` package wraps it. This matters for three jobs the browser is bad at: answering questions about many repositories at once, checking things before you download them, and automating the boring parts of publishing.

```
pip install -U huggingface_hub
```

Listing models with the filters you would have clicked

```
from huggingface_hub import HfApi

api = HfApi()

models = api.list_models(
    task="automatic-speech-recognition",
    library="transformers",
    language="hi",
    sort="downloads",
    direction=-1,
    limit=20,
)

for m in models:
    print(f"{m.downloads:>9,}  {m.id}")
```

This is the URL from the search lesson, in code, and it returns objects you can filter further in Python — which is where it stops being a browser replacement and starts being useful. "Speech models for Hindi, updated in the last year, under Apache 2.0" is three lines of list comprehension and is not a query the sidebar can express.

Checking size before you download

The most valuable call is the one that costs nothing:

```
info = api.model_info("openai/whisper-large-v3",
files_metadata=True)

total = sum(f.size or 0 for f in info.siblings)
print(f"{len(info.siblings)} files, {total / 1e9:.1f} GB")
for f in sorted(info.siblings, key=lambda s: -(s.size or 0))
[:5]:
    print(f"  {(f.size or 0)/1e9:6.2f} GB  {f.rfilename}")
```

Run that before any `snapshot_download` and you will stop being surprised by disk usage. It also shows the duplicate-format problem from the last lesson immediately: you will see the `.bin` and the `.safetensors` sitting next to each other, each several gigabytes.

`model_info` also carries `info.tags`, `info.cardData` (the parsed YAML front matter of the card, including licence and `base_model`), `info.gated`, and `info.lastModified`. Everything the card page shows in a badge is in there as a field.

Downloading precisely

```
from huggingface_hub import hf_hub_download, snapshot_download

# one file
cfg = hf_hub_download("openai/whisper-large-v3", "config.json")

# a subset, pinned, skipping duplicate formats
path = snapshot_download(
    "openai/whisper-large-v3",
    revision="06f233fe06e710322aca913c1bc4249a0d71fce1",
    allow_patterns=["*.safetensors", "*.json", "*.txt"],
    ignore_patterns=["*.bin", "onnx/*"],
)
```

`allow_patterns` and `ignore_patterns` are the difference between a 6 GB download and a 12 GB one on repositories that ship several formats. Set `HF_HUB_ENABLE_HF_TRANSFER=1` in the environment after `pip install hf_transfer` if your connection is fast enough to be bottlenecked by Python rather than the network; on a slow link it makes no difference.

Writing: repositories, files, and the commit you can see

```
api.create_repo("your-name/your-model", repo_type="model", pri-
vate=True)

api.upload_file(
    path_or_fileobj="README.md",
    path_in_repo="README.md",
    repo_id="your-name/your-model",
    commit_message="Add model card with licence and
limitations",
)

api.upload_folder(
    folder_path="./out",
    repo_id="your-name/your-model",
    ignore_patterns=["*.ckpt", "runs/*", "*.log"],
)
```

Every write is a commit with a message, visible in History. Use the message: six months later it is the only record of why a checkpoint changed.

The command line does the same things

```
hf auth login
hf download openai/whisper-small --include "*.safetensors"
"*.json"
hf upload your-name/your-model ./out --commit-message "first
checkpoint"
hf repo create your-name/your-dataset --repo-type dataset
hf cache scan
```

The binary was called `huggingface-cli` until 2025 and both names still appear in tutorials. `hf cache scan` is the one to remember; it prints what your cache is holding and is the quickest way to find the 40 GB you cannot account for.

What the API will not give you

- **Rate limits are real.** Anonymous calls are limited more tightly than authenticated ones, and a loop over ten thousand `model_info` calls will start getting HTTP 429. Authenticate, batch, and cache results to disk.
- **`downloads` is a thirty-day count**, not a total, and the next lesson explains what it counts.
- **The API cannot tell you whether a model is good.** Every field is either a counter or something the uploader typed. There is no quality signal in the JSON, and treating `downloads` as one is the mistake this whole module exists to prevent.

Do this now

Write ten lines that list the twenty most-downloaded models for a task you care about, and print each one's licence from `cardData`. You will find that several of the top results have no licence field at all — which, as the card lesson argued, is not the same as permissive.

BEFORE YOU MOVE ON

Why does `model_info(repo, files_metadata=True)` frequently report a total far larger than what you need to download?

A. It reports the size of the full commit history, including superseded checkpoint versions still held in large-file storage.

B. It counts every file at that revision, including duplicate weight formats you would never fetch.

C. Sizes are returned before large-file deduplication, so the true transfer is always substantially smaller than reported.

D. The call sums sizes across every branch and open pull request in the repository.

9 · 8 min

What a download count is counting

THE ONE THING TO KEEP

Downloads are a thirty-day count of file fetches dominated by CI, library defaults and ephemeral containers, so they are a floor on how well-exercised a model is and not a ranking — while fine-tune, adapter and Space counts cost real effort and carry more information.

Four million downloads, and what that means

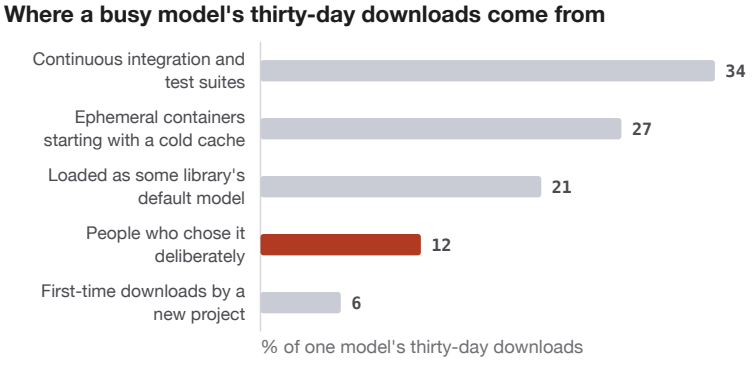
The number under a model name is a thirty-day rolling count of file resolutions — requests that actually fetched a weights file, deduplicated in the way the Hub deduplicates them. It is not four million people. It is not even four million distinct machines.

The things that inflate it, roughly in order of how much they inflate it:

- **Continuous integration.** A library with a test suite that loads a model on every commit, across a matrix of Python versions, generates thousands of downloads a week from one project.
- **Default models.** A model that a popular library loads when you do not specify one collects the downloads of everyone who did not make a choice. `sentence-transformers/all-MiniLM-L6-v2` is enormously downloaded partly because it is the sensible default in dozens of tutorials and several libraries.
- **Ephemeral compute.** Every Colab session, every container that starts with a cold cache, every Space that restarts, downloads again. A single busy Space can produce more downloads than a hundred human users.
- **Age.** A three-year-old model has had three years of all the above, though the thirty-day window blunts this more than people expect.

And the things that deflate it: a shared cache on a cluster registers one download for a lab of fifty; a popular model served behind an API shows one down-

load and a million uses.



Illustrative proportions; the Hub does not publish the breakdown. The shape is the point: most of the number is machines, not people, which is why downloads are a floor on how well exercised a model is and never a ranking.

Why this matters for a decision

The download count is a **floor on exercise**, and that is genuinely useful. A model with two million downloads has been loaded by enough different code paths that the obvious breakages — a missing tokenizer file, a config that no released transformers version can parse, weights that do not match the architecture — have been found and reported. You are unlikely to be the first person to discover it does not load.

It is not a ranking, because it does not measure quality, recency or fit. Three failure modes follow directly:

1. **The superseded champion.** A model tops its task by downloads while a successor two versions newer sits at a tenth of the count. Downloads lag adoption by months.
2. **The wrong-job winner.** The most downloaded model for `text-classification` is not the most downloaded model for *your* text, your language, or your length of input.
3. **The dependency artefact.** A tiny model used internally by another library outranks every model a human ever chose on purpose.

Likes, and what they are better at

A like is one click by one signed-in account, and unlike downloads it cannot be produced by a test suite. It is a weaker signal in volume and a cleaner one in kind: it means a person looked at the repository and thought it was worth marking.

Likes skew hard toward the announcement moment, so they measure attention rather than sustained use, and they accumulate on organisations people already trust. A model with 3,000 likes and 400 downloads is famous. A model with 40 likes and 900,000 downloads is load-bearing. Those are different things and you often want the second.

Trending

The trending sort is a recency-weighted blend of activity — downloads, likes and discussion inside a short window. Its purpose is news. Sorting a task by trending tells you what shipped this week; it is the right view once a month and the wrong view when choosing a dependency, because by construction it penalises anything stable.

The counters that are worth more than all of these

On the right-hand side of a model page, under the metadata, sit the numbers almost nobody reads:

- **Finetunes / Adapters / Quantizations.** If four hundred people have fine-tuned a model, four hundred people got far enough to train on it. This is the closest thing on the Hub to a quality signal, because it costs real effort rather than a click.
- **Spaces using this model.** A Space is a running application. If thirty Spaces load a model, it works in thirty configurations you did not have to test.
- **Last modified.** A model last touched in 2022 is not automatically bad — `bert-base-uncased` is still correct — but for a fast-moving family it tells you the maintainer has moved on.

- **Open discussions.** Two unanswered bug threads from eight months ago say more about the support you will receive than any counter.

The honest summary

Use downloads to eliminate, not to choose. A model with four downloads and no discussions is a coin flip; you may be its first real user, and that is a reasonable choice if its card is good and you have thirty test examples of your own. A model with four million downloads has cleared the loading bar and nothing else.

The decision between the two finalists never comes from the Hub's numbers. It comes from running both on your own examples, which is what the evaluation module builds.

Do this now

Open the most-downloaded model for any task and find its Spaces count and its last-modified date. Then open the fifth result and do the same. In about half of the tasks on the Hub, the fifth is newer, better documented, and the one you should use.

BEFORE YOU MOVE ON

Two speech models sit side by side: A has 2.1 million downloads, 12 likes, 3 Spaces and was last updated two years ago. B has 40,000 downloads, 900 likes, 260 Spaces and 180 fine-tunes. What does this pattern most likely indicate?

A. A is the stronger model, since sustained download volume is the only counter that cannot be inflated by a burst of attention.

B. B is newer and has not yet been exercised enough to trust, so A remains the safer production choice.

C. A is probably pulled by CI or as a library default, while B is being chosen and built on by people.

D. The two counters measure different tasks, so the comparison is meaningless and only a leaderboard score can separate them.

CASE STUDY · MODULE 1

The name that was spelled two ways

A legal aid clinic in Pune transcribing Marathi and Hindi hearing audio into case files, with four volunteers and twenty donated GPU hours a week.

Since January the clinic had been running one script. It took an audio file from a hearing, pushed it through a large open speech model, and wrote a transcript into the matter's folder. Fourteen hundred hours of audio had gone through it by August. The script was ninety lines long and nobody had touched it since the week it was written.

Nobody in the clinic thought of it as software. It was the thing that turned nine hours of a bad recording made on a phone at the back of a courtroom into text a lawyer could read on a train. The volunteers called it the transcriber, and the only thing anybody ever changed about it was the folder it wrote to.

In August a paralegal called Sushma noticed that a respondent's surname appeared as Waghmare in one transcript and Vaghmare in another, in the same matter, from two hearings six weeks apart. She assumed a typing error until she found three more, all in files produced after June.

Anup, the volunteer who had written the script, opened the model's Files and versions tab and then its History. There were two commits since January. One in April replaced the tokenizer files. One in June reuploaded a corrected checkpoint, with a message saying it fixed a bug affecting proper nouns.

The script said `from_pretrained("openai/whisper-large-v3")`. No revision. So it had never meant one model. It had meant whatever was on the main branch at the moment each transcript was made, and the archive was not one archive but three, with nothing in any filename to say which.

The decision

The clinic quotes these transcripts in written submissions. Two options were on the table and both cost something real.

Re-transcribe everything. The donated allocation was twenty hours a week on a college lab machine. Fourteen hundred hours of audio at roughly real time is about three weeks of that allocation, during which no new intake could be transcribed. Intake does not pause. Three weeks meant roughly ninety new matters waiting on a transcript before anybody could read them.

Or pin from today, accept an archive produced by three different models, and carry the risk. That risk is not a daily cost. It is a credibility cost that arrives once, unpredictably, in front of a bench, when opposing counsel asks why two documents from the same clinic spell the same person's name differently.

Anup made one more check before anybody chose. The June commit message said the fix concerned proper nouns. That narrowed the problem from everywhere to one place, and one place happened to be what a legal transcript is mostly about. He also asked a question nobody had asked yet: how much of the archive belongs to matters still before a court. The answer was a hundred and ninety hours. The rest was closed, or kept for research.

The cost on one side was ninety delayed matters. The cost on the other was a document nobody could vouch for. The information that made the choice tractable was already sitting in the repository, free, in a commit message and a list of dates.

The clinic also had to decide what it was willing to say out loud. A transcript in a submission is offered as a record of what was said. Saying that some of the clinic's transcripts were produced by a model that has since been corrected is an uncomfortable sentence, and it is a sentence somebody would eventually have to say either way — the only question was whether the clinic said it first, in a note it had written calmly, or second, in an answer to a question from a bench.

Anup pointed out one more thing that made the pinning decision easy even if the re-running decision was not. Pinning costs nothing. It is one extra argument in three function calls, it can be done in ten minutes, and it does not compete with anything for the donated GPU hours. Whatever they chose about the existing fourteen hundred hours, there was no version of the argument in which they should keep loading an unpinned model tomorrow.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They pinned. Every loading call in the script got a `revision` argument carrying the June commit hash, copied out of the History tab, including the one that loads the processor. They re-ran the hundred and ninety hours belonging to open matters, which took four days rather than three weeks, and left the closed archive as it was with a one-page note in the shared drive recording which date ranges came from which revision and what the June fix had changed.

Every transcript produced from then on carried a header line with the model identifier, the revision hash and the date it was made. Six months later a bench did ask about a spelling. The clinic answered in one sentence and produced the note. Writing the note had cost an afternoon.

Anup's script had not been edited since January, yet the transcripts it produced changed. Explain the mechanism.

A Hugging Face repository is a git repository and `from_pretrained("owner/name")` without a `revision` argument resolves to whatever is on the main branch at the moment the code runs. Two commits landed between January and August, so the same ninety lines of code loaded three different sets of weights and tokenizer files over the period. Pinning `revision` to a commit hash removes the whole category, and costs nothing.

Reading the June commit message changed the size of the problem. Why is that, and what does it say about the History tab?

The message said the fix affected proper nouns. That turned an unbounded worry — every word of fourteen hundred hours might differ — into a bounded one concentrated in names. History is a diff and not merely a list of dates, so it answers what changed as well as when. For a model that started behaving differently last week, it is the first place to look and often the last.

They re-ran a hundred and ninety hours rather than fourteen hundred. What makes that a defensible decision rather than a convenient shortcut?

They split the archive along the axis where the risk actually sits. A transcript from a closed matter is not going to be read back to a judge; one from an open matter might be. They also recorded what they had not re-run and why, which is the part that turns a scoped decision into an auditable one. A shortcut hides the gap; this documented it.

MODULE 1 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A transcription script has not been edited since January, yet the transcripts it produced in July differ from those it produced in March. What is the mechanism?

A. A cached copy expired, so the weights were fetched again in a different precision.

B. from_pretrained without a revision loads whatever is on the main branch that day.

C. The Hub serves different mirrors to different regions and they drift apart over time.

D. Downloading a model twice gives different bytes, because large files are deduplicated.

2 You clone a model repository with plain git and find that model.safetensors is about 130 bytes. What has happened?

A. The repository is gated and the Hub served a small placeholder instead of the real weights.

B. The maintainer committed an empty file and pushed the real weights to a branch.

C. Git cloned the pointer; the bytes live in large-file storage and were never fetched.

D. The model is quantized to four bits, which is why the file is so small.

3 A model shows four million downloads over thirty days. What is the strongest inference you can draw from that number?

A. Enough different code paths have loaded it that obvious breakages are known.
B. It is the best-performing model available for its task at this size, by some margin.
C. Roughly four million distinct people have used it during the last month.
D. It is actively maintained, because a stale model stops being downloaded entirely.

4 Your laptop cannot run a model at full precision. What is the fastest route to a version that fits?

A. Type the model's name and the word small into the search box.
B. Sort the task by trending and take the most recent release.
C. Open the base model's page and follow its Quantizations link.
D. Filter the whole Hub by model size band and read through every result.

5 A card tags a fine-tune apache-2.0. Its base_model field names a repository under a non-commercial research licence. Which terms govern your paid product?

A. Apache 2.0, because a fine-tune is a new work and carries its own licence.
B. Whichever is the more recent, since licences are versioned like code.
C. Neither, until the uploader answers a question in the discussions tab.
D. The non-commercial terms, because a derivative inherits its base's terms.

6 A repository ships only safetensors weights, and its usage snippet passes trust_remote_code=True. What risk remains?

A. None, because safetensors contains no instruction stream and cannot execute code.

B. Python from the repository is downloaded and imported when the model loads.

C. The safetensors header can be crafted to call a function while it is being parsed.

D. Loading will fail, because that flag only applies to pickle checkpoints.

MODULE 2

Transformers, from characters to output

The transformers library is three layers stacked: a tokenizer that turns text into integers, a model that turns integers into logits, and a decoder that turns logits back into text. Most of the bugs people report as bad models are one of those three layers being used wrongly. This block walks the whole path with real numbers, then gives you an ordered checklist for the case where everything loads and the output is still rubbish.

> BY THE END OF THIS MODULE YOU CAN
>
> Trace a piece of text from characters through a tokenizer to logits and back to text with the transformers library, and say which of the Auto class, the tokenizer, the padding side, the chat template or the generation config is at fault when a model loads without error and produces nonsense

10 · 8 min

One line that hides five steps

THE ONE THING TO KEEP

`pipeline` resolves, tokenizes, runs, post-processes and formats in one call, and when you do not name a model it silently picks a small English-only default whose confidence score is a softmax over two numbers rather than a probability of being right.

The shortest thing that works

```
from transformers import pipeline

clf = pipeline("sentiment-analysis")
print(clf("The delivery was late but the food was excellent."))
# [{'label': 'POSITIVE', 'score': 0.9993}]
```

Two lines, no GPU, no configuration. This works on a laptop and on the free tier of Colab, and it is the right place to start because it lets you check that your environment is sane before you take on anything harder.

It also prints a warning that almost everybody scrolls past:

> *No model was supplied, defaulted to distilbert-base-uncased-finetuned-sst-2-english*

That sentence is doing a lot of work. You did not choose a model; the library chose one for you. It is a 67-million-parameter English-only classifier fine-tuned on movie reviews from 2013. It is fast and it is decent on product feedback. It will be wrong in ways you cannot predict on Hinglish, on sarcasm, on medical notes, and on anything longer than 512 tokens, which it silently truncates.

The first real skill is naming the model yourself:

```
clf = pipeline(
    "sentiment-analysis",
    model="cardiffnlp/twitter-roberta-base-sentiment-latest",
)
```

What the one line is actually doing

`pipeline` assembles five steps and hides them:

1. **Resolve** — read the repository's `config.json` to find the architecture, and pick the right classes.
2. **Tokenize** — turn your string into integers, add the special tokens the model expects, build an attention mask.
3. **Forward pass** — run the tensors through the network and get raw scores out.
4. **Post-process** — softmax the scores, map the winning index through `id2label`, round.
5. **Format** — return a list of dictionaries.

Every one of those steps is a place a later lesson will take apart. For now the point is that `score: 0.9993` is not a probability of being right. It is a softmax over two numbers. A model can be catastrophically wrong at 0.9993 and there is nothing in the pipeline output to tell you so.

The task names, and why they matter

The first argument is a task from a fixed list, and it must match the model's `pipeline_tag`:

- `text-classification` (aliased `sentiment-analysis`)
- `token-classification` (aliased `ner`)
- `question-answering` — extractive, pulls a span out of a passage you supply
- `summarization`, `translation`, `text2text-generation`
- `text-generation` — the chat and completion models

- `feature-extraction` — embeddings
- `automatic-speech-recognition`, `audio-classification`
- `image-classification`, `object-detection`, `image-segmentation`
- `zero-shot-classification` — you pass candidate labels at call time

`zero-shot-classification` deserves a look early, because it solves a real problem with no training:

```
z = pipeline("zero-shot-classification", model="facebook/bart-
large-mnli")
z("My parcel has not arrived and nobody replies.",
  candidate_labels=["delivery", "refund", "product quality",
"account"])
```

It is slower than a fine-tuned classifier and usually a few points worse, and it needs no labelled data at all, which is often the trade you want in week one.

Making it run on what you have

```
import torch

pipe = pipeline(
    "text-generation",
    model="Qwen/Qwen2.5-0.5B-Instruct",
    device_map="auto",          # GPU if present, CPU if not
    dtype=torch.bfloat16,       # half the memory of float32
)
```

`device=-1` forces CPU and `device=0` forces the first GPU; `device_map="auto"` is the modern form and handles multiple devices. The `dtype` argument was called `torch_dtype` until 2025 and you will see both spellings in tutorials.

On a CPU, bfloat16 is often *slower* than float32, because many CPUs have no native half-precision arithmetic and the runtime converts back and forth. This is the opposite of the GPU advice, and it surprises people who copied a Colab notebook onto a laptop.

Where the pipeline stops being the right tool

Use it for exploration, demos, one-off scripts and Spaces. Move past it when you need:

- **Batching that you control**, because the pipeline's batching heuristics are conservative.
- **The raw scores**, because you want a threshold rather than an argmax.
- **A shared model across several calls**, since re-creating a pipeline re-loads weights.
- **Anything unusual** — a custom head, an intermediate layer, a loss.

The next lesson opens the box.

Do this now

Run the two-line sentiment example, then run it again on a sentence in a language other than English, and on a sentence with sarcasm in it. Print the score both times. The default model does not know it is out of its depth, and neither does its confidence number.

BEFORE YOU MOVE ON

A team ships `pipeline("sentiment-analysis")` with no model argument, and it performs well in testing on English product reviews but poorly on their real Hinglish support messages. What is the mechanism?

A. The default model is a small English-only classifier, and nothing in the API signals that the input is outside what it was trained on.

B. The pipeline detects the input language and routes it to a multilingual model, which is weaker on short informal text than the English default would be.

C. Mixed-script input breaks the attention mask, so the tokenizer emits padding tokens that the classifier reads as neutral content and scores near 0.5.

D. Confidence scores are uncalibrated for any input, so the model is equally unreliable on English and the testing result was chance.

11 · 9 min

AutoModel gives you numbers, not answers

THE ONE THING TO KEEP

`AutoModel` loads the network body and returns hidden states, while `AutoModelForX` attaches a task head — and when the checkpoint has no such head, transformers initialises it randomly and warns rather than failing, so a classifier that returns confident noise is the expected outcome of loading a base checkpoint.

Three Auto classes, and only one of them is safe to guess

```
from transformers import AutoConfig, AutoTokenizer, AutoModel

name = "distilbert-base-uncased"
cfg = AutoConfig.from_pretrained(name)
tok = AutoTokenizer.from_pretrained(name)
model = AutoModel.from_pretrained(name)
```

Each `Auto*` class reads `config.json`, finds `architectures`, and instantiates the matching implementation. That is all the magic is: a lookup table from a string to a Python class, so you do not have to import `DistilBertTokenizerFast` by name.

`AutoConfig` and `AutoTokenizer` do what you expect. `AutoModel` does something that catches almost everyone.

The body without the head

```
out = model(**tok("the food was excellent",
return_tensors="pt"))
print(out.last_hidden_state.shape)   # torch.Size([1, 6, 768])
```

No labels. No scores. A tensor of 6 positions × 768 numbers — one vector per token. `AutoModel` loads the **body** of the network and nothing else. It is the right class when you want representations to feed something of your own, and the wrong class when you want a prediction.

For a prediction you name the head:

```
from transformers import AutoModelForSequenceClassification

m = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-uncased-finetuned-sst-2-english"
)
```

The common heads, and what each is for:

- `AutoModelForSequenceClassification` — one label for the whole input. Sentiment, intent, spam.
- `AutoModelForTokenClassification` — one label per token. Named entities, part of speech.
- `AutoModelForQuestionAnswering` — start and end positions of a span.
- `AutoModelForCausalLM` — next-token prediction. Every chat and completion model.
- `AutoModelForSeq2SeqLM` — encoder-decoder. Translation, older summarisers, T5.
- `AutoModelForMaskedLM` — fill in a masked token. BERT-style pretraining.

The warning that means your classifier is random

Load a head that the checkpoint does not contain and transformers tells you, in a message people read as boilerplate:

> *Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-uncased and are newly initialized: ['classifier.weight', 'classifier.bias']. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.*

Those two tensors were filled with random numbers. The model will load, run, and return confident-looking scores that are pure noise. Nothing downstream will fail. You will get roughly 50% accuracy on a two-class problem and spend a day blaming the data.

The distinction is exactly this: `distilbert-base-uncased` is a **base** checkpoint with no classifier. `distilbert-base-uncased-finetuned-sst-2-english` has one. The warning is the only thing that tells them apart at load time, and it is worth treating as an error rather than a log line.

`id2label`, and why your labels say LABEL_0

When output comes back as `LABEL_0` and `LABEL_1`, the uploader did not fill in the mapping:

```
print(m.config.id2label)   # {0: 'NEGATIVE', 1: 'POSITIVE'}
```

If it prints `{0: 'LABEL_0', 1: 'LABEL_1'}`, the card is the only place the real meaning exists — and sometimes it is only in a discussion thread. Getting the polarity backwards because you assumed index 1 meant positive is a common and completely silent error. Check, do not assume.

The other Auto classes you will meet

- `AutoProcessor` — the general wrapper for models that take more than text: it bundles a tokenizer with an image processor or a feature extractor. Vision-language models and Whisper use it.
- `AutoImageProcessor` — resizing and normalisation for vision models. Normalisation constants differ per model and using the wrong ones degrades accuracy without any error.
- `AutoModelForVision2Seq`, `AutoModelForSpeechSeq2Seq` — heads for captioning and transcription.

The rule that prevents most of this

Load the tokenizer and the model from the same repository identifier, always. A tokenizer from `bert-base-uncased` with weights from a fine-tune of a different vocabulary produces integers the model was never trained on. No exception is raised, because integers are integers. The output is fluent nonsense, which is the hardest kind of bug to spot.

When you must mix them — loading an adapter, say — check that `len(tokenizer)` equals `model.config.vocab_size` and stop if it does not.

Do this now

Load `distilbert-base-uncased` with `AutoModelForSequenceClassification` and read the warning in full. Then run it on ten sentences and look at the predictions. That is what a randomly initialised head looks like from the outside: confident, varied, and meaningless.

BEFORE YOU MOVE ON

Someone loads `bert-base-uncased` with `AutoModelForSequenceClassification`, sees a warning, and gets about 50% accuracy on a balanced two-class task. What happened?

A. The base checkpoint was pretrained on masked language modelling, so its representations are unsuitable for classification without a different architecture.

B. The tokenizer and the model came from the same repository but the label mapping defaulted to LABEL_0 and LABEL_1, inverting the classes.

C. The classification head does not exist in the checkpoint, so it was created with random weights and never trained.

D. Sequence classification needs `AutoModelForMaskedLM` for BERT-family models, and the wrong head silently returns pooled embeddings instead of logits.

12 · 10 min

Your text is integers, and the count is not the word count

THE ONE THING TO KEEP

A tokenizer maps text to integers from a fixed vocabulary, and the tokens-per-word ratio depends on whose language the vocabulary was built from — so the same sentence can cost two to four times as much in an Indic script as in English on an English-centric model.

Look at the integers once

```
from transformers import AutoTokenizer

tok = AutoTokenizer.from_pretrained("gpt2")
enc = tok("Tokenisation is unforgiving")
print(enc["input_ids"])
print(tok.convert_ids_to_tokens(enc["input_ids"]))
```

You get something like `['Token', 'isation', 'Ġis', 'Ġun', 'forg', 'iving']` — six tokens for three words. The `Ġ` is a visible stand-in for a leading space, because byte-level tokenizers treat the space as part of the token that follows it. That is why `"cat"` and `" cat"` are different tokens with different ids, and why a stray trailing space in a prompt can change the output.

A tokenizer is a fixed vocabulary — typically 32,000 to 256,000 entries — plus rules for splitting anything into pieces from that vocabulary. Common words get one token. Rare words get chopped. Nothing is ever unrepresentable, because the fallback is bytes.

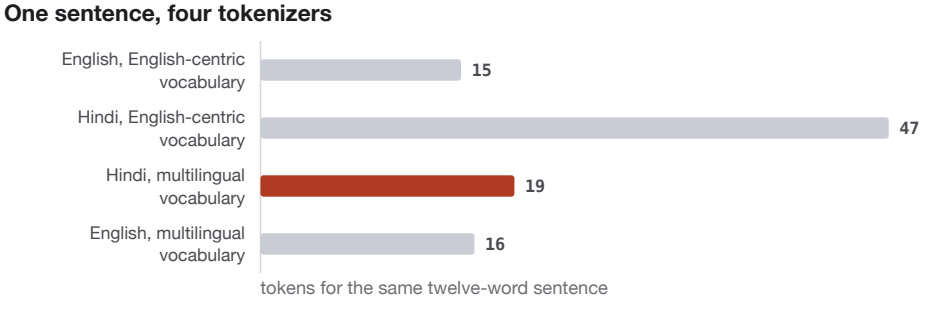
The ratio, and why it is not the same for everyone

For English prose with an English-centric vocabulary, the working figure is about **1.3 tokens per word**, or roughly four characters per token. That number is where the rule of thumb "750 words ≈ 1,000 tokens" comes from.

It does not transfer. The same text in Hindi, Bengali, Tamil, Telugu or Marathi, tokenized by a model whose vocabulary was built mostly from English text, commonly runs two to four times as many tokens for the same meaning, because Devanagari and other Indic scripts fall back to multi-byte pieces. Try it:

```
for s in ["Where is my parcel?", "मेरा पार्सल कहाँ है?"]:
    print(len(tok(s)["input_ids"]), s)
```

Three consequences, all practical:



The same meaning, measured four ways. On an English-centric vocabulary the Hindi sentence costs three times as many tokens, which is three times the price per request, three times the context window used, and three times as many steps to get right. A multilingual vocabulary closes most of the gap.

1. **Cost.** Per-token pricing means the same question costs more in Hindi than in English on the same model. This is a real, measurable inequity in how these systems are priced, and it is a property of the vocabulary rather than of the language.
2. **Context.** A 4,096-token window holds much less Hindi than English.
3. **Quality.** More tokens per word means the model has more steps to get right for the same content.

Models with multilingual vocabularies — Qwen, Gemma, Aya, IndicBERT and the muril family — narrow the gap substantially. Checking the ratio on your own text is a ten-second test that should inform model choice and almost never does.

Special tokens appear without being asked for

```
tok = AutoTokenizer.from_pretrained("bert-base-uncased")
print(tok("hello")["input_ids"])              # [101, 7592,
102]
print(tok.convert_ids_to_tokens([101, 102]))  # ['[CLS]',
'[SEP]']
```

You wrote one word and got three integers. Encoder models wrap input in markers; decoder models often prepend a beginning-of-sequence token. `add_special_tokens=False` turns this off, which you want when you are assembling a sequence yourself and do not want a second BOS in the middle of it — a genuinely common bug when people concatenate a chat template with a manual encode.

Truncation is silent

```
enc = tok(long_text, truncation=True, max_length=512)
```

Without `truncation=True`, a too-long input raises or produces a tensor the model rejects. With it, the text is cut and nothing tells you. A classifier handed a 3,000-word complaint reads the first 512 tokens — which, for a complaint, is the polite introduction rather than the accusation at the end.

If that matters, either chunk deliberately, or pick a model with a longer window, or use `truncation_side="left"` so you keep the end instead of the beginning.

Offsets: getting back to the characters

```
enc = tok("Ravi lives in Pune", return_offsets_mapping=True)
```

Each token comes with the character span it came from. This is how token classification output becomes highlighted text in a UI, and it only works with the fast (Rust) tokenizers — one more reason `tokenizer.json` in a repository is good news.

Two rules that prevent most tokenizer bugs

- **Never mix a tokenizer with a model from a different repository.** The integers mean different things in different vocabularies, and nothing checks.
- **If you add tokens, resize the embeddings.** `tok.add_tokens(["<sku>"])` followed by `model.resize_token_embeddings(len(tok))`. Skip the second line and you get an index error at best, and reads from uninitialised memory at worst.

Do this now

Take one paragraph of your own text — in whatever language you actually work in — and count its tokens under two models: one English-centric like `gpt2`, one multilingual like `Qwen/Qwen2.5-0.5B`. Divide by the word count. That ratio is a cost multiplier you will carry through the rest of the course.

BEFORE YOU MOVE ON

A support tool costs three times as much per message for Hindi users as for English users on the same model and the same message lengths. What explains it?

A. Multilingual models run a translation step internally before generating, which doubles or triples the tokens processed per request.

B. Devanagari text falls back to many short vocabulary pieces, so the same meaning becomes far more tokens and billing is per token.

C. Non-Latin scripts are stored as UTF-16 internally, so every character consumes two positions in the context window.

D. Hindi prompts are longer in characters on average, so the difference is in what users write rather than in how it is tokenized.

13 · 9 min

Padding on the wrong side ruins generation quietly

THE ONE THING TO KEEP

The attention mask makes padding invisible to the model, but a decoder-only model generates from the last position, so right padding makes it continue from a pad token and quietly degrades batched output while single requests still look fine.

Why padding exists at all

A batch is a rectangle. Three sentences of 4, 11 and 6 tokens cannot be stacked into a tensor until they are the same length, so the short ones get filled with a pad token:

```
enc = tok(["hi", "a much longer sentence here"], padding=True,
return_tensors="pt")
print(enc["input_ids"])
print(enc["attention_mask"])
```

The mask is the important half. It is 1 for real tokens and 0 for padding, and the attention mechanism uses it to make padded positions contribute nothing. Pass `input_ids` without the mask and the model attends to the padding as if it were content — a mistake that is easy to make when people build tensors by hand and then wonder why batched results differ from one-at-a-time results.

> *Rule: whatever the tokenizer returns, pass all of it.* `model(**enc)`*, never* `model(enc["input_ids"])`*.*

The side matters, and only for generation

For an encoder model doing classification, padding on the right is fine. Every position is visible to every other, the mask zeroes the pad, the pooled output is correct.

For a decoder-only model doing generation, right padding is a silent quality bug. Generation continues from the **last position** of the sequence. If the sequence ends in padding, the model is being asked to continue from a pad token, having seen your prompt some distance back. Depending on the model and the mask handling, you get output that is degraded, generic, or occasionally empty. No error, no warning, and the same prompt run alone produces a good answer — so people conclude that batching breaks the model.

The fix is one line:

```
tok.padding_side = "left"
```

Now every sequence ends with its real final token and generation starts from the right place. Do this for every `AutoModelForCausalLM` you batch. For training a causal model, right padding is correct, because labels are shifted and the loss ignores padded positions. The side depends on the operation, not on the model.

The missing pad token

Many decoder-only checkpoints ship with no pad token, because generation one at a time never needs one. Batch them and you get:

> *ValueError: Asking to pad but the tokenizer does not have a padding token.*

The standard fix:

```
if tok.pad_token is None:
    tok.pad_token = tok.eos_token
```

This is fine for inference. For training it is worth thinking about once: with `pad_token == eos_token`, a naive label setup can teach the model that end-of-sequence is followed by more end-of-sequence, which makes it fail to stop. The usual answer is to set padded label positions to `-100`, which the loss ignores, and the data collators in the library do this for you.

Truncation, and the side of that too

```
enc = tok(texts, padding=True, truncation=True,
max_length=1024, return_tensors="pt")
```

`padding=True` pads to the longest item in the batch. `padding="max_length"` pads everything to `max_length`, which wastes compute but produces uniform shapes — occasionally required by compiled or exported models.

`truncation_side` defaults to right. For a chat history, cutting from the right removes the most recent turn, which is the one that mattered. Set `tok.truncation_side = "left"` when the end of the input is the important part.

The cost of padding

Padding is wasted computation. A batch of 32 where one item is 2,000 tokens and the rest are 50 runs as if all 32 were 2,000 — roughly a fortyfold waste. Two cheap mitigations:

1. **Sort by length before batching**, so each batch is internally uniform. This alone often doubles throughput on mixed-length data and costs about four lines.
2. **Cap the batch by token count rather than by item count** — for example 8,000 tokens per batch — so a batch of long items is small and a batch of short items is large.

Serving stacks like vLLM and text-generation-inference remove the problem entirely with continuous batching, but the sorting trick works today in a plain script and needs no new dependency.

The five-line sanity check

After any change to batching, run one input alone and the same input inside a batch of eight, and compare:

```
solo = pipe("What is the capital of Karnataka?")
batched = pipe(["filler"] * 7 + ["What is the capital of
Karnataka?"])[-1]
```

They should agree. When they do not, the padding side is the first suspect, the missing mask the second.

Do this now

Take any small instruct model, batch four prompts of very different lengths with the default right padding, and look at the shortest prompt's answer. Then set `padding_side = "left"` and run the identical batch. The difference is the bug this lesson exists for, and seeing it once makes it permanent.

BEFORE YOU MOVE ON

A chat model gives good answers one request at a time and vague or empty ones when the same prompts are batched. What should be checked first?

A. Whether the attention mask is being passed, since without it the padded positions are attended to as if they were real content.

B. Whether the batch exceeds the model's context window, causing silent truncation of the earlier prompts in the batch.

C. Whether sampling parameters differ between the two call paths, since batched calls often inherit different generation defaults.

D. Whether `padding_side` is still the default right, so short prompts end in padding and generation continues from a pad token.

14 · 10 min

Logits, and the four knobs between them and words

THE ONE THING TO KEEP

A model outputs one unnormalised score per vocabulary token and the decoding strategy picks from them, so temperature, top-p and repetition penalties are choices you make after the model has finished — and a `do_sample: false` default in generation_config silently discards every sampling parameter you pass.

What comes out of the model

```
out = model(**enc)
print(out.logits.shape)   # [batch, sequence_length,
vocab_size]
```

For a causal model with a 151,936-token vocabulary and a 12-token prompt, that is a tensor of 12 × 151,936 raw numbers. Only the last row matters for generating the next token: it holds one score per possible next token, unnormalised, typically between about −20 and +20.

Softmax turns that row into a probability distribution. Then something has to **choose**, and everything people call creativity or randomness lives in that choice.

```
import torch
next_logits = out.logits[0, -1]
probs = torch.softmax(next_logits, dim=-1)
top = torch.topk(probs, 5)
for p, i in zip(top.values, top.indices):
    print(f"{p:.3f}  {tok.decode(i)!r}")
```

Run that on a half-finished sentence once. Seeing the actual candidates — and how often the top one holds 0.95 of the mass — makes every knob below intuitive rather than mystical.

Greedy, and why it repeats

Greedy decoding takes the argmax every step. It is deterministic, which is exactly what you want for classification-like tasks, extraction and anything you need to reproduce.

It also falls into loops. Once the model emits a phrase, that phrase raises the probability of itself recurring, and with no randomness nothing breaks the cycle — so you get *"the results show that the results show that the results show that"*. This is a property of maximising at every step, not a defect in a particular model.

```
out = model.generate(**enc, do_sample=False,
max_new_tokens=200)
```

Temperature

Temperature divides the logits before the softmax. Below 1 it sharpens the distribution — the likely tokens get likelier. Above 1 it flattens it. At 0 it is greedy.

- `0.0–0.3` — extraction, classification, code, anything with a right answer.
- `0.7–0.8` — general assistant use. Most chat defaults sit here.
- `1.0–1.3` — brainstorming, variety, fiction.
- Above `1.5` — usually incoherent, because low-probability tokens become reachable.

Top-k and top-p

Both truncate the candidate list before sampling.

- **top-k** keeps the k highest-probability tokens. Fixed count, so it keeps 50 candidates whether the model is certain or not.
- **top-p** (nucleus) keeps the smallest set whose probabilities sum to p. Adaptive: when the model is confident, that might be two tokens; when it is unsure, forty.

top-p is usually the better default precisely because it responds to the model's confidence. `top_p=0.9` with `temperature=0.7` is the combination behind most defaults you will meet.

Repetition controls

`repetition_penalty=1.1` divides the logits of tokens already present. `no_repeat_ngram_size=3` forbids any 3-gram from occurring twice. The second is blunt — it makes a model literally unable to repeat a person's name three times, which in a summary of a legal document is wrong. Prefer a mild repetition penalty and fix the rest with the prompt.

The interaction that wastes an afternoon

```
out = model.generate(**enc, temperature=0.9, top_p=0.95,
max_new_tokens=100)
```

If the model's `generation_config.json` has `do_sample: false`, this runs **greedily** and your temperature does nothing. Recent versions warn; older ones do not. The output is identical on every run and people conclude the model is broken or the seed is stuck.

Be explicit:

```
out = model.generate(**enc, do_sample=True, temperature=0.9,
top_p=0.95,
                     max_new_tokens=100)
```

Inspect what you inherited with `print(model.generation_config)` before benchmarking anything. Two models that appear to differ in quality frequently differ only here.

Decoding back

```
text = tok.decode(out[0], skip_special_tokens=True)
```

`generate` returns the prompt plus the continuation, so trimming matters:

```
new_tokens = out[0][enc["input_ids"].shape[-1]:]
print(tok.decode(new_tokens, skip_special_tokens=True))
```

Without `skip_special_tokens=True` you see `<|im_end|>` and friends in your UI, which is the single most common cosmetic bug in a first demo.

Reproducibility

`set_seed(0)` makes sampling repeatable on the same hardware and library versions. Across GPUs, dtypes or versions, floating-point accumulation order differs and the same seed can produce different text. Determinism is per-configuration, not absolute, and any evaluation that assumes otherwise will chase ghosts.

Do this now

Generate the same prompt four times: greedy, then `temperature=0.7, top_p=0.9`, then `temperature=1.5`, then greedy with `no_repeat_ngram_size=3`. Four runs, one prompt. That comparison teaches more about decoding than any explanation.

BEFORE YOU MOVE ON

A developer sets `temperature=1.2` and gets byte-identical output on every run. What is the most likely cause?

A. The random seed is fixed somewhere in the environment, so sampling is reproducible rather than disabled.

B. The model's generation config sets `do_sample` to false, so decoding is greedy and the temperature is ignored.

C. Temperature above 1.0 is clamped back to 1.0 by the library, which makes the distribution effectively deterministic for confident models.

D. Sampling only applies once `top_p` or `top_k` is also set, because temperature alone has no candidate set to sample from.

15 · 9 min

The chat template is not optional

THE ONE THING TO KEEP

An instruction-tuned model was trained on one exact string format that ships inside `tokenizer_config.json`, so `apply_chat_template(..., add_generation_prompt=True)` is the only reliable way to build a prompt — hand-written `User:`/`Assistant:` text puts the model outside its training distribution with no error to show for it.

The most common quality bug in open models

Somebody downloads an instruct model, writes this, and concludes it is weak:

```
prompt = "User: Summarise this in two lines.\n\nAssistant:"
```

That is not how the model was trained. During instruction tuning it saw a specific string — specific role markers, specific special tokens, specific newlines — thousands of times. Give it something that looks similar but is not identical and you are outside the distribution it learned. Output gets vaguer, it ignores system instructions, it fails to stop, it starts inventing a second user turn.

The correct format is not a convention you can guess. It is stored in the repository, in `tokenizer_config.json`, as a Jinja string named `chat_template`.

Two prompts the model sees very differently

Typed by hand

- User: and Assistant: written as plain text
- None of the special tokens the model was trained on
- Nothing marks where the assistant's turn begins
- The model often invents a second user turn
- Rewritten by hand for every model family

Rendered by apply_chat_template

- The exact Jinja string from tokenizer_config.json
- The role markers the model saw thousands of times
- add_generation_prompt opens the assistant turn
- The end marker matches the configured eos_token_id
- Unchanged when you swap the model underneath

Neither raises an error and both produce fluent text, which is why this is the commonest quality bug in open models. The left column is a string that resembles the training format; the right is the training format, read out of the repository the weights came from.

Use it

```
messages = [
    {"role": "system", "content": "You reply in one
sentence."},
    {"role": "user", "content": "Why is the sky blue?"},
]

text = tok.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True,
)
print(repr(text))
```

For a ChatML-style model that prints something like:

```
<|im_start|>system
You reply in one sentence.<|im_end|>
<|im_start|>user
Why is the sky blue?<|im_end|>
<|im_start|>assistant
```

Every model family differs. Llama 3 uses `<|start_header_id|>` markers, Mistral uses `[INST]` and `[/INST]`, Gemma uses `<start_of_turn>`. Guessing wrong is invisible in the output shape and visible in the quality.

`add_generation_prompt=True` is the part people miss. It appends the opening marker for the assistant's turn, so the model knows it is its turn to speak. Without it the model may produce another user message, because that is what would plausibly come next.

Print it once per model

`tokenize=False` exists so you can look. Do it once for every new model — it takes five seconds and tells you the system-role support, the turn markers, and the stop token:

```
print(tok.chat_template)          # the Jinja source
print(tok.eos_token, tok.eos_token_id)
```

Some models genuinely have no system role, and their template silently folds the system message into the first user turn, or drops it. That explains a whole class of "it ignores my instructions" reports.

In the pipeline

The text-generation pipeline applies the template for you when you pass a list of messages rather than a string:

```
pipe = pipeline("text-generation", model="Qwen/Qwen2.5-1.5B-
Instruct")
out = pipe(messages, max_new_tokens=128)
print(out[0]["generated_text"][-1]["content"])
```

That is the recommended path, and it is why the message-list form is worth adopting even in throwaway scripts: it stops you hand-writing a format you will get wrong later.

Multi-turn, and the thing to get right

A conversation is the whole list, re-templated every turn:

```
messages.append({"role": "assistant", "content": reply})
messages.append({"role": "user", "content": follow_up})
```

Append the assistant's *clean* reply — not the raw decode with special tokens still in it. Feeding `<|im_end|>` back in as message content corrupts the next template render, and the model starts emitting stray markers.

Tool calling lives here too

Models trained to call functions define the format in the same template. `apply_chat_template(messages, tools=[...])` renders the tool schemas into whatever shape that family expects, and the model's tool calls come back as text you parse. There is no cross-model standard, which is why a tool-calling application written against one model needs work to move to another. Frameworks paper over this; the underlying variation is real.

Two models, one application

If your code has to support more than one model, keep the messages list as the only representation of a conversation anywhere in your program, and render it at the last possible moment with whichever tokenizer is loaded. The moment a rendered string with `[INST]` in it is stored in a database or passed between functions, you have hard-coded one family, and swapping the model becomes a refactor rather than a config change.

The same rule protects your evaluation set. Store the messages, not the prompt. A set of rendered prompts measures one model and cannot be reused against its successor, which is the quiet way a team ends up unable to compare anything.

When there is no template

Base models have none, and `apply_chat_template` raises or falls back to a default. That error is useful information: it usually means you picked a base checkpoint when you wanted an instruct one. Occasionally a genuine instruct model ships without its template by oversight — then the card, or a discussion

thread, is where the format is written, and it is worth opening a pull request to add it.

Do this now

Print the rendered template for two instruct models from different families with the same two messages. Put the outputs side by side. They will share no markers at all — and that is the whole argument for never writing the prompt format by hand.

BEFORE YOU MOVE ON

A model keeps writing a plausible follow-up question from the user instead of answering. The messages list and template are otherwise correct. What is the likely omission?

A. `add_generation_prompt=True` was not set, so the rendered string does not open the assistant's turn.

B. The system message is missing, and without one the model has no instruction to answer rather than converse.

C. The eos token is absent from the template, so generation runs past the end of the answer into invented turns.

D. The previous assistant reply was appended with its special tokens intact, which corrupts the role sequence in the render.

16 · 9 min

Throughput and latency are different problems

THE ONE THING TO KEEP

Batching trades latency for throughput by amortising weight reads, so it pays on GPUs and often not on CPUs — and because a generation batch runs until its slowest member finishes, sorting by length and watching the KV cache matter more than raising the batch size.

Decide which number you are optimising

Latency is how long one person waits. **Throughput** is how many items you finish per minute. Batching improves the second by making the first worse, and almost every confused performance discussion is two people optimising different ones.

- Interactive chat: latency. Time to first token is what a user perceives.
- Overnight classification of 400,000 rows: throughput. Nobody is waiting.
- A queue behind a web request: both, resolved by a deadline.

Why batching helps at all

A forward pass reads every weight from memory. On a GPU, moving 14 GB of weights costs far more time than the arithmetic does, so a batch of one is mostly the hardware waiting for memory. Push 32 items through together and the weights are read once for all of them. The arithmetic grows with the batch; the memory traffic barely does.

That is why batch size 32 is often nowhere near 32 times slower than batch size 1 on a GPU — five to ten times is typical for small models.

On a CPU the picture is different. There is no comparable memory-bandwidth headroom, and the cores are already saturated by one item. Batching on CPU

gives modest gains and sometimes none. Measure rather than assume; this is the single most common false transfer from GPU tutorials to laptop reality.

Batching with the pipeline

```
pipe = pipeline("text-classification", model=name, device=0,
batch_size=32)
results = pipe(list_of_texts)
```

Pass a list, not a loop. Better, pass a `Dataset` so items stream rather than all sitting in memory:

```
from datasets import Dataset
ds = Dataset.from_dict({"text": list_of_texts})
for r in pipe(KeyDataset(ds, "text"), batch_size=32):
    ...
```

The padding tax, measured

Batching sorted by length is the highest-return four lines in this course:

```
order = sorted(range(len(texts)), key=lambda i: len(texts[i]))
sorted_texts = [texts[i] for i in order]
# run in batches, then restore original order with `order`
```

On mixed-length support tickets — a few words to a few thousand — this routinely doubles throughput, because a batch no longer runs at the length of its longest member. Restoring the original order afterwards is the part people forget, and it produces a silent mislabelling of every row.

Generation is a different shape

Classification is one forward pass. Generation is one pass per output token, and the batch finishes when its **slowest** member finishes. Batch eight prompts where seven produce 20 tokens and one produces 500, and you paid for 500 steps on all eight.

The KV cache is what makes each step cheap: attention keys and values for tokens already processed are kept rather than recomputed. It also costs memory that grows with batch size times sequence length, and it is why a batch that fits at 512 tokens runs out of memory at 4,096. When you see out-of-memory partway through generation rather than at load, the cache is the reason.

Measuring honestly

```
import time, torch

# warm-up: the first call compiles kernels and allocates
_ = pipe(texts[:8])

torch.cuda.synchronize()          # GPU work is asynchronous
start = time.perf_counter()
_ = pipe(texts, batch_size=32)
torch.cuda.synchronize()
print(len(texts) / (time.perf_counter() - start), "items/sec")
```

Three things people get wrong here, all of which produce numbers that are too good:

1. **No warm-up.** The first batch includes one-off costs and is two to ten times slower; excluding it is right, forgetting it is wrong in the other direction.
2. **No synchronise.** CUDA calls return before the work is done. Without the sync you are timing how fast Python can queue work.
3. **Reporting the mean.** A mean latency of 300 ms with a 99th percentile of 4 seconds is a system some users experience as broken. Report the tail.

When to stop tuning and change tool

For serving many concurrent generation requests, a plain transformers loop is the wrong shape however well you tune it. vLLM, text-generation-inference and llama.cpp's server implement continuous batching — new requests join a running batch instead of waiting for it — and paged attention, which stops the

KV cache fragmenting memory. The gain is typically several times, not a few per cent.

For batch scoring of a fixed list, a sorted transformers loop is fine and needs no new dependency.

Do this now

Time 200 mixed-length inputs three ways: one at a time, batched unsorted, batched sorted by length. Write the three numbers down. The gap between the second and third is free and most people never collect it.

BEFORE YOU MOVE ON

A batch job of 50,000 short texts and a few very long ones runs far slower than the per-item timings suggest. Batch size is already 64. What is the most likely cause?

A. The batch size exceeds what the GPU can hold, so the framework is silently splitting batches and paying the overhead twice.

B. Short texts finish early but their results are held until the batch completes, so wall-clock time reflects queueing rather than compute.

C. Every batch runs at the length of its longest member, so a few long items inflate the cost of the short ones padded beside them.

D. The tokenizer is the bottleneck at this volume, since fast tokenizers do not parallelise across a batch.

17 · 8 min

Showing tokens as they arrive, and making them stop

THE ONE THING TO KEEP

Streaming changes perceived latency rather than compute, and it works by running `generate` on a second thread feeding a queue — while a model that never stops usually has a mismatch between the end token its template emits and the `eos_token_id` its generation config is watching for.

Why streaming changes the product

A 300-token answer at 25 tokens per second takes twelve seconds. Delivered at the end, that is a long silence and people reload the page. Delivered as it arrives, the first words appear in under a second and the wait is tolerable. Nothing about the computation changed; the perceived speed did.

This is why time to first token is the latency number worth tracking for anything interactive, and total time is the one worth tracking for anything batch.

The simple streamer

```
from transformers import TextStreamer

streamer = TextStreamer(tok, skip_prompt=True, skip_special_tokens=True)
model.generate(**enc, max_new_tokens=200, streamer=streamer)
```

This prints to stdout as tokens decode. It is perfect for a terminal script and useless for a web application, because it writes rather than yields.

The one you use in an app

```
from transformers import TextIteratorStreamer
from threading import Thread

streamer = TextIteratorStreamer(tok, skip_prompt=True,
skip_special_tokens=True)
kwargs = dict(**enc, max_new_tokens=512, streamer=streamer)

Thread(target=model.generate, kwargs=kwargs).start()

for chunk in streamer:
    yield chunk          # into a Gradio generator, or an SSE
response
```

`generate` blocks, so it runs on a second thread while the main thread consumes the queue. This is the shape behind every streaming chat UI you have used, including the Gradio Spaces in the next module.

Chunks are not tokens and not words. The decoder buffers until it can emit valid text, so you may get `" un"` then `"forgiving"`. Never assume a chunk boundary is a word boundary, and never split on it for parsing.

Why models do not stop

Generation ends when one of three things happens: the model emits its end-of-sequence token, `max_new_tokens` is reached, or a stopping criterion fires. When a model rambles past its answer into an invented next turn, the usual cause is that the EOS it was trained to emit is not the EOS configured at generation time — a common mismatch in fine-tunes and conversions, where the template's end marker and `eos_token_id` disagree.

Check them against each other:

```
print(tok.eos_token, tok.eos_token_id)
print(model.generation_config.eos_token_id)
```

Some models legitimately have several stop tokens, and `eos_token_id` accepts a list:

```
model.generate(**enc, eos_token_id=[tok.eos_token_id,
                                    tok.convert_tokens_to_ids("
<|eot_id|>")])
```

`max_new_tokens`, not `max_length`

`max_length` counts prompt plus output, so a 900-token prompt with `max_length=1024` leaves room for 124 tokens and the answer gets cut off mid-sentence. `max_new_tokens` counts only the output and is what you almost always want. Both exist; only one behaves the way people expect.

Always set it. A model with no limit and a broken stop token will generate until it exhausts the context window, which on a 128k model is a long and expensive silence.

Stop strings

When the model reliably emits a marker you want to cut at:

```
model.generate(**enc, max_new_tokens=256, stop_strings=
["\nUser:", "```end"],
               tokenizer=tok)
```

Stop strings are checked on decoded text, so the tokenizer must be passed. They are a patch over a prompt or template problem rather than a fix, but they are an effective patch and there is no shame in it.

What streaming costs you

Streaming is not free of trade-offs, and two of them bite in production.

You cannot validate output you have already shown. If the answer must be checked — against a schema, a moderation pass, a groundedness test — you

either buffer it and lose the benefit, or you display text you may have to retract. Most teams compromise: stream the prose, buffer anything structured.

And a streamed response is harder to cache. A complete answer is one object you can store against a prompt hash. A stream is a sequence of events, so the usual pattern is to assemble the full text as it goes and write that to the cache at the end, serving later identical requests instantly and non-streamed.

Cancellation, which people forget

A user closes the tab and your GPU keeps generating 500 tokens for nobody. In a threaded setup, a custom `StoppingCriteria` that checks a flag is the mechanism:

```
from transformers import StoppingCriteria

class Cancelled(StoppingCriteria):
    def __init__(self, flag): self.flag = flag
    def __call__(self, input_ids, scores, **kw): return
self.flag.is_set()
```

On a free Space with one worker, one abandoned long generation blocks everybody in the queue. This is a small piece of code with a large effect on how a shared demo feels.

Do this now

Stream a 300-token answer and time two things: seconds to the first visible character, and seconds to the last. Then run the same prompt without streaming and note only the second number. The gap between those two experiences is the whole argument, and it costs six lines.

BEFORE YOU MOVE ON

A fine-tuned chat model answers correctly and then continues into an invented next user turn until it hits the token limit. What is the first thing to check?

A. Whether `max_new_tokens` is set too high, since an unbounded limit encourages the model to keep producing text.
B. Whether the template's end marker matches the `eos_token_id` generation watches for.
C. Whether the temperature is high enough for the model to escape the loop that keeps it generating.
D. Whether the streamer is buffering chunks, so the extra turn appears only because decoded text was released late.

18 · 9 min

It loaded, it ran, the output is nonsense

THE ONE THING TO KEEP

Fluent-but-wrong output has a short list of causes to check in order — base instead of instruct, missing chat template, mismatched tokenizer, padding side, dtype overflow, over-aggressive quantization, inherited generation defaults and silent truncation — and the model itself is the last suspect, not the first.

The worst category of bug

No traceback. No warning you noticed. The model produced fluent, confident, wrong text, or a classifier returned labels that look plausible and are random. Everything downstream works perfectly on garbage.

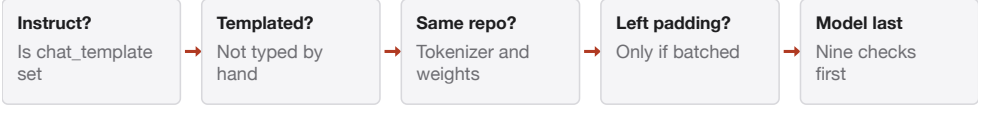
This lesson is a checklist in order of how often each cause is the cause. Work down it rather than guessing; each step is under a minute.

1. Is it an instruct model?

```
print("chat_template:", tok.chat_template is not None)
```

`False` on a model you are prompting conversationally means you have a base checkpoint. A base model continues text; asked "What is the capital of Bihar?" it may reply with three more quiz questions. Nothing is broken. Get the `-Instruct` variant.

The order to check, not the order to guess



Each step is under a minute and they are ordered by how often each is the cause. The model itself is the tenth suspect, not the first, because fluent-but-wrong output is far more often a configuration mismatch than a bad checkpoint.

2. Did you apply the template?

Covered at length earlier, and it stays at number two because it stays that common. If you built the prompt with f-strings and role labels, stop and use `apply_chat_template`.

3. Is the tokenizer from the same repository as the weights?

```
print(len(tok), model.config.vocab_size)
```

These should match, or differ only by a handful of added tokens. A serious mismatch means the integers mean different things to the two halves, which produces exactly the fluent-nonsense signature. This is the classic failure when loading an adapter against the wrong base.

4. Are the padding side and mask right?

If single calls are good and batched calls are bad, this is it. `tok.padding_side = "left"` for decoder-only generation, and pass the mask.

5. What dtype is it actually in?

```
print(next(model.parameters()).dtype)
```

Two real failures here:

- **float16 overflow.** Some models, notably the T5 family and a few older checkpoints, were trained in bfloat16 and overflow in float16 — you get `nan` in the logits and empty or garbage output. `bfloat16` has the same exponent range as float32 and does not; use it when the GPU supports it.
- **float16 on CPU.** Slow, and on some builds numerically wrong. Use float32 on CPU unless you have measured otherwise.

A quick test: if `torch.isnan(out.logits).any()` is true, you have found it.

6. Is the quantization too aggressive?

4-bit is fine for chat and noticeably worse for arithmetic, code and long structured output. If a model is good at fp16 and bad at 4-bit on your task, that is the quantization, not a bug. Step up to 8-bit or a `Q5_K_M` GGUF and re-check before blaming anything else.

7. Are the generation parameters sane?

```
print(model.generation_config)
```

Look for `temperature` above 1.2 with no top-p, a `repetition_penalty` above about 1.3 (which starts suppressing ordinary words), and `do_sample: false` quietly ignoring everything you passed.

8. Did the input get truncated?

```
print(enc["input_ids"].shape[-1],
model.config.max_position_embeddings)
```

Silent truncation removes the end of a long document — usually the part that mattered. For a classifier reading complaints, this alone can halve accuracy.

9. For classifiers: what does `id2label` say?

`{0: 'LABEL_0', 1: 'LABEL_1'}` means nobody told the model which is which, and a coin flip on polarity is a 50% chance you have it backwards across your whole dataset.

10. Only now, suspect the model

After nine checks, it is reasonable to conclude the model is unsuited. Confirm it cheaply: run the same ten inputs through a well-known model of similar size. If that one is also poor, the task is harder than you thought or your inputs are unusual — both useful findings.

The minimal reproduction

When you report this to a discussion thread or a colleague, include: model id **with revision**, transformers and torch versions, the exact rendered prompt (`tokenize=False`), the generation kwargs, the dtype, and the device. Half the questions on the Hub cannot be answered because the rendered prompt is missing, and the rendered prompt is the answer about a third of the time.

Do this now

Deliberately break one working script four ways — drop the template, right-pad a batch, force float16 on a T5, and load the base instead of the instruct checkpoint. Look at each failure. You are building a library of signatures, and recognising a signature turns an afternoon into a minute.

BEFORE YOU MOVE ON

A T5-family summariser returns empty strings on a GPU under float16 but works on CPU under float32. What is happening?

A. The GPU kernels for encoder-decoder attention require float32, so the half-precision path falls back to an untested code branch.

B. Activations exceed float16's range, producing NaN logits, so decoding immediately hits the end token and emits nothing.

C. The model's weights were saved in bfloat16, and converting them to float16 rounds small values to zero, erasing the decoder's embeddings.

D. CPU and GPU use different tokenizer implementations, so the device change altered the input ids rather than the arithmetic.

CASE STUDY · MODULE 2

The classifier that was ninety-nine per cent sure of nothing

A district education office in Nashik with thirty-eight thousand parent feedback forms, a review meeting in twelve days, and one intern who knew Python.

The forms came back from two hundred and twelve schools, most of them handwritten and then typed up by clerks. About two thirds were in Marathi and the rest in English or a mixture. The review meeting that would allocate a maintenance budget wanted a picture of what parents were complaining about, and nobody had ever put numbers on the forms before.

The clerks had done their part properly. Every form had a school code, a date and a line of free text that ran anywhere from four words to nine hundred. What nobody had was a way of saying how many of those lines were about a leaking roof and how many were about a teacher who does not come.

Rohit, on a three-month placement, had the pipeline working in an afternoon. Six lines, a well-known model name he had seen in a tutorial, and a loop over a spreadsheet. The output looked entirely believable: sixty-one per cent of forms in one category, the rest spread across three others, with confidence scores mostly above 0.95.

The labels came back as LABEL_0 and LABEL_1. He mapped LABEL_1 to positive because one usually means yes, and moved on.

The decision

Shalini Pawar, the deputy director, had a pack due in four days and this was the only quantitative thing in it. The alternative to using it was two days of people reading forms by hand, and there were no two days.

She took sixty forms home instead and labelled them herself over one evening. The model agreed with her on thirty-three. That is a coin flip with a very good opinion of itself.

Rohit went back to the output of his first run and found the warning he had scrolled past. It said that two tensors, the classifier weight and the classifier bias, had not been found in the checkpoint and had been newly initialised, and that he should probably train the model before using it for predictions. He had loaded a base checkpoint with no classification head. The library had filled the head with random numbers, warned in a log line, and then run perfectly.

Two more things surfaced the same evening. The model was English-only, and two thirds of the forms were in Marathi. And on that model's vocabulary a Marathi sentence cost three to four times as many tokens as its English equivalent, so the five-hundred-and-twelve-token truncation was cutting long complaints in half — and a complaint puts its polite introduction first and its accusation last.

So the choice in front of Shalini was not whether to trust a slightly noisy number. It was whether to present a fabricated distribution to a meeting that allocates money, or to arrive with less than she had promised.

Shalini's evening of sixty forms is worth dwelling on, because it is the cheapest thing in this case study and it is the only reason anything was caught. She did not audit the code, read the model card, or ask anybody technical. She took the output and the input and compared them by hand, on a sample small enough to finish, and the answer was unambiguous within an hour.

Rohit's instinct afterwards was to blame the model, which is the tenth item on the checklist rather than the first. The model was fine. It was a perfectly good base checkpoint, downloaded four million times, doing exactly what a base checkpoint does. What had gone wrong was a mismatch between the class he asked for and the weights that existed, and a warning that described the mismatch in plain English in the terminal, above the first result.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

She withdrew the automatic pass. Four people hand-labelled four hundred forms over two days, chosen to cover all two hundred and twelve schools rather than the first four hundred in the pile, and the pack reported those four hundred with the disagreement rate between labellers printed beside every figure. One paragraph explained that an automatic pass had been attempted and withdrawn, and why. The review accepted it, and the disagreement rate turned out to be the number the committee argued about most usefully.

Rohit rebuilt the pipeline on a multilingual model whose card named its labels in `id2label`, set the truncation side to the left so the end of a complaint survived, and added five lines at the top of the script that refuse to run if `id2label` still reads LABEL_0 or if loading raised an initialisation warning. On the four hundred hand-labelled forms the new pipeline agreed three hundred and forty-one times — a number that could be quoted, because the sample it came from could be named.

Why did the model return confident scores instead of failing, and what did the 0.95 confidence actually measure?

The checkpoint was a base model with no classification head, so transformers created one, filled it with random numbers and emitted a warning rather than an exception — deliberately, because that is the normal first step before fine-tuning. The score is a softmax over the head's raw outputs. A softmax over random numbers is still a distribution and still has a maximum, so it always looks confident. It is not a probability of being right.

Two thirds of the forms were in Marathi on an English-centric tokenizer. Name two separate costs that follows from, beyond accuracy.

First, the token count: the same meaning costs three to four times as many tokens, so a fixed truncation limit holds far less Marathi than English and the input is silently cut. Second, where the cut falls: truncation defaults to the right, and a complaint's substance is usually at the end, so the classifier read the polite opening and never reached the accusation. Setting `truncation_side` to left, or chunking, fixes the second; only a better vocabulary fixes the first.

Rohit's five new lines refuse to run under two conditions. Why are those two worth an automatic check rather than a habit?

Both failures are silent and both survive every downstream test. A randomly initialised head produces output of exactly the right shape, and LABEL_0 output looks like a label until somebody asks which class it is. Neither raises, so neither can be caught by watching for errors; they can only be caught by asserting the thing you assumed. A habit is forgotten under a deadline and an assertion is not.

MODULE 2 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A default sentiment pipeline returns a score of 0.9993 on a Hinglish sentence. What does that number actually measure?

A. The probability that the classification is correct, given the model's calibration.

B. A softmax over the head's raw outputs, which says nothing about being right.

C. How close the input was to the model's training distribution during pretraining.

D. The margin between the top two labels, expressed as a confidence percentage.

2 Loading a base checkpoint with AutoModelForSequenceClassification warns that classifier.weight was newly initialised. What will the model then do?

A. Refuse to predict until train() has been called at least once on the head.

B. Return the body's hidden states instead of labels, as AutoModel would.

C. Predict with a randomly initialised head, at roughly chance accuracy.

D. Borrow the nearest matching head from another checkpoint already in the cache.

3 The same twelve-word sentence costs 15 tokens in English and 47 in Hindi on one model. What causes the difference?

A. Hindi needs more grammatical marking, so the same meaning takes more words.
B. Devanagari is stored as two bytes per character, which doubles every count.
C. The tokenizer inserts an extra special marker token before each non-Latin character it meets.
D. The vocabulary was built mostly from English, so Devanagari falls back to small pieces.

4 A prompt that is answered well on its own returns generic text inside a batch of eight. What is the first suspect?

A. Right padding, so a decoder-only model continues from a pad token.
B. The batch exceeded the context window and every item was truncated.
C. Batching changes the sampling seed, so each item decodes differently.
D. The attention mask was applied twice and therefore cancelled itself out.

5 You pass temperature=0.9 and top_p=0.95 to generate, and every run returns an identical answer. What is the likely cause?

A. An earlier call to set_seed fixed the sampler for the remainder of this Python session.
B. Temperature applies only to encoder-decoder models, not to causal ones.
C. The model's generation_config sets do_sample to false, discarding both values.
D. A top_p above 0.9 is defined as greedy decoding, so sampling is skipped.

6 A fine-tuned chat model rambles past its answer and invents a new user turn. Which explanation fits best?

A. max_new_tokens was too low, so the answer was cut off and then restarted.

B. The end marker its template emits and the configured eos_token_id disagree.

C. Streaming was enabled, which bypasses the model's stopping criteria entirely.

D. The tokenizer has no pad token, so the model pads with generated text instead.

MODULE 3

Running it on the hardware you actually have

Most of this readership is on a phone, a shared laptop, or a free Colab session that disconnects after a few hours. This block is the arithmetic and the tooling for that reality: how to compute the memory a model needs before downloading it, what quantization buys and costs, the formats that run well without a GPU, where free compute genuinely exists and where its limits are, and how to measure whether a change helped.

> **BY THE END OF THIS MODULE YOU CAN**
>
> Compute from a model's parameter count and dtype whether it will fit the machine in front of you, pick between a full-precision, quantized, GGUF or ONNX route for a stated constraint, and measure peak memory, first-token latency and quality loss well enough to defend the choice

19 · 11 min

Your disk fills up before your patience does

THE ONE THING TO KEEP

Weights cost parameters times bytes-per-parameter and everything downloaded lands in one shared cache, where deleting the readable filenames frees nothing because they are only links to the blobs.

Two lines that quietly download two gigabytes

Running a model yourself is less dramatic than it sounds. Install one package and call one function:

```
pip install -U "transformers[torch]"
```

```
from transformers import pipeline

clf = pipeline(
    "text-classification",
    model="distilbert-base-uncased-finetuned-sst-2-english",
)
print(clf("The bus was late but the driver waited for me."))
```

That model is around 250 MB and classifies in a fraction of a second on any laptop CPU. No GPU, no configuration. The point of the example is not sentiment analysis, which is a solved and slightly boring task — it is that `pipeline` downloads the weights, the tokenizer and the config, caches them, and gives you a working function. Swap the model name and the same three lines do translation, transcription, captioning or entity extraction.

The cache, and the symlink that fools you

Everything you download goes to one shared cache, by default `~/.cache/huggingface/hub`. Point it somewhere with room by setting one variable before you start:

```
export HF_HOME=/mnt/bigdisk/huggingface
```

Inside the cache, each repository has a `blobs/` directory holding the actual bytes and a `snapshots/<commit-hash>/` directory holding symbolic links with the human-readable filenames. Two revisions of the same model share every blob that did not change.

This structure produces one specific and very common disaster. You run out of disk, you go into the cache, you find a folder full of familiar filenames under `snapshots/`, you delete them, and nothing is freed. You deleted links. The blobs they pointed at are still there. Use the tool instead:

```
hf cache scan
hf cache delete
```

On older versions these are `huggingface-cli scan-cache` and `huggingface-cli delete-cache`. The scan prints every repository, its size and its revisions, which is also how you discover you have four copies of the same 13 GB model from four different tutorials.

The memory arithmetic

This is the whole of capacity planning, and it fits in five lines.

- 32-bit weights: 4 bytes per parameter.
- 16-bit: 2 bytes.
- 8-bit: 1 byte.
- 4-bit: roughly half a byte.

So a 7-billion-parameter model needs about 28, 14, 7 or 4 GB for the weights alone. Then add the key-value cache, which grows with how much context you feed it, and a gigabyte or two of framework overhead. A rule that holds: the weights plus about a quarter again.

The failure mode matters as much as the number. If a model does not fit in RAM, your machine does not usually raise a clean error — it starts swapping to disk, and the process appears to hang. People wait forty minutes for a first token believing it is slow when it is actually stuck. Check the size before you start, not while you wait.

What a machine with no GPU actually does

A great deal, once you stop trying to run the wrong things.

Small models run fine. Classification, embeddings, named entities, translation with a compact model, Whisper at tiny or base size. These are CPU-friendly because they are small, not because of any special trick.

Image generation is possible at low step counts. Most diffusion models need twenty to fifty passes through the network to make one image, which is what makes them hopeless on CPU. Distilled models cut that to one to four:

```
pip install -U diffusers torch pillow
```

```
from diffusers import AutoPipelineForText2Image
import torch

pipe = AutoPipelineForText2Image.from_pretrained(
    "stabilityai/sd-turbo", torch_dtype=torch.float32
).to("cpu")

img = pipe(
    "a steel tumbler of chai on a wooden table, morning light",
    num_inference_steps=2,
    guidance_scale=0.0,
).images[0]
img.save("chai.png")
```

About 2.5 GB to download, tens of seconds per image on a normal CPU, and the quality is well below SDXL. Note the licence — `sd-turbo` is non-commercial, exactly as lesson two warned. If the output is going anywhere near a paying client, this is not the model. /learn/ai-images covers the craft side.

Chat models run through llama.cpp, not through `transformers`. The GGUF format packs quantised weights into a single file, and the Hub is full of them from community quantisers. `Q4_K_M` is the usual balance of size against damage. Ollama and LM Studio both pull GGUF files straight from the Hub, and LM Studio has a graphical interface on Windows, macOS and Linux, so this path needs no Python at all. Rough expectations: with 8 GB of RAM a 3B model at 4-bit reads at a comfortable speed and a 7B is slow but usable; with 16 GB, 7B to 8B is comfortable. Apple silicon does unusually well because the CPU and GPU share memory. /learn/running-models-yourself goes into this properly.

A phone does not run these. What a phone runs is a browser pointed at a Space, or very small models compiled for the browser through transformers.js. That is not a limitation to work around; it is the reason lessons four and five exist.

Do this now

Run `hf cache scan` on whatever machine you have been experimenting on. Most people find several gigabytes they did not know about, and at least one model downloaded twice.

BEFORE YOU MOVE ON

A laptop is out of disk. The user opens the Hugging Face cache, finds a `snapshots` folder full of files like `model.safetensors`, deletes several gigabytes of them, and the free space barely changes. Why?

A. The deleted files went to the operating system's trash and still occupy the disk until it is emptied.
B. The library re-downloaded the models automatically as soon as it noticed they were missing.
C. The files under `snapshots` are symbolic links to blobs stored elsewhere in the cache, so removing them deletes the names and not the bytes.
D. Those were quantised copies that share weights with the full-precision originals, so the real space is held by the originals.

20 · 9 min

The memory arithmetic you can do in your head

THE ONE THING TO KEEP

Weights cost parameters times bytes-per-dtype, but the bill that decides whether generation fits is weights plus a KV cache that grows with sequence length and batch — and bfloat16 is preferred over float16 not for size but because it keeps float32's exponent range and so does not overflow to NaN.

One multiplication decides most of it

> ***weights in bytes** = **parameters** × **bytes per parameter***

The bytes per parameter come from the dtype:

- `float32` — 4 bytes. The default if you say nothing, and almost never what you want.

- `float16` / `bfloat16` — 2 bytes. The normal choice on a GPU.
- `int8` — 1 byte.
- `int4` — about 0.5 bytes, plus a small overhead for scales.

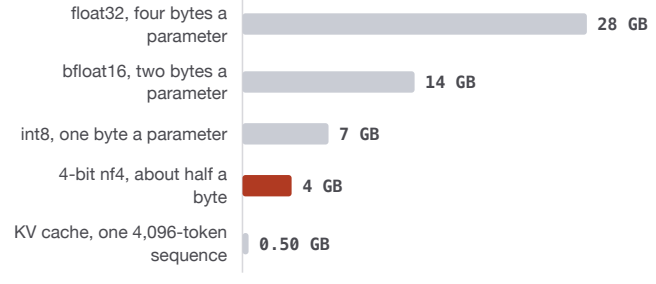
So a 7-billion-parameter model is 28 GB at float32, 14 GB at bfloat16, 7 GB at int8 and roughly 4 GB at int4. A 1.5B model is 3 GB at bfloat16 and fits comfortably in a free Colab session with room to work. A 70B model is 140 GB at bfloat16, which is two 80 GB cards, and about 40 GB at int4, which is one.

You can now look at any repository name and know, before downloading anything, whether it is worth downloading.

float16 and bfloat16 are not interchangeable

Both take two bytes. They spend those bytes differently.

A seven-billion-parameter model, by dtype



The first four bars are the weights alone. The fifth is what a single 4,096-token conversation adds on top of whichever of them you chose: eight concurrent conversations of that length cost about four gigabytes, which is the whole of the quantized model again. Most people discover the constraint is the cache, not the weights.

- **float16** gives more bits to precision and has a narrow exponent range, with a maximum around 65,504.
- **bfloat16** keeps float32's exponent range and sacrifices precision.

For neural networks the exponent range matters more, because activations occasionally spike. A value above 65,504 becomes infinity in float16, infinity times zero becomes NaN, and the NaN spreads through the network within one layer. This is the mechanism behind the T5 and older-checkpoint failures from the debugging lesson.

Practical rule: **use bfloat16 if the hardware supports it** — Ampere-generation GPUs and newer, which includes the A100 and the free-tier T4's successors but not the T4 itself. On a T4, float16 works for most modern models and fails on a few. On CPU, use float32; half precision there is usually emulated and slower.

```
model = AutoModelForCausalLM.from_pretrained(name,
dtype=torch.bfloat16)
print(next(model.parameters()).dtype)
```

Check the second line. Loading without specifying gives you float32 on some paths regardless of what `config.json` says the weights were saved as, and doubling your memory by accident is a routine mistake.

Weights are not the whole bill

Three other things occupy memory during inference:

1. **Activations** — the intermediate tensors of the forward pass. Modest for a single short sequence, and they scale with batch size and sequence length.
2. **The KV cache** — for generation, the stored keys and values for every token so far. This is the one that grows, and the formula is worth carrying:

> KV bytes ≈ 2 × layers × kv_heads × head_dim × sequence_length × batch × bytes_per_element

For an 8B model with 32 layers, 8 key-value heads and a head dimension of 128 at bfloat16, one 4,096-token sequence costs roughly 0.5 GB. Eight concurrent conversations at that length cost about 4 GB — comparable to the weights of a quantized model.

1. **Framework overhead** — CUDA context, allocator fragmentation, the library itself. Budget one to two gigabytes and stop being surprised.

So the honest working estimate for inference is:

> ***weights + KV cache + 1–2 GB***

And for training, roughly **four times the weights** before you count the data, because you also hold gradients and two optimiser states per parameter under Adam. That is why a model you can run happily is a model you cannot fine-tune fully on the same card — and why the adapter methods in the training module exist.

Grouped-query attention changed the numbers

Older models gave every attention head its own keys and values. Most models since 2023 share them across groups, which cuts the KV cache by a factor of four to eight. It is why a modern 8B model handles a long context on hardware where a 2022 7B model would not. The relevant fields are `num_attention_heads` and `num_key_value_heads` in `config.json`; when the second is much smaller than the first, the cache is cheap.

Reading your own usage

```
import torch
torch.cuda.reset_peak_memory_stats()
_ = model.generate(**enc, max_new_tokens=256)
print(f"peak: {torch.cuda.max_memory_allocated()/1e9:.2f} GB")
```

`max_memory_allocated` is the number that matters, not the current allocation — out-of-memory errors happen at the peak, which may occur for a few milliseconds inside one layer. On CPU, `psutil.Process().memory_info().rss` gives the equivalent.

Do this now

Pick three models you might use — a small one, a mid one, and one that is clearly too large. For each, write down the bf16 weight size, the int4 size, and the KV cache for 4,096 tokens. Compare against the memory of the machine you actually have. Most people discover the constraint is the cache, not the weights.

BEFORE YOU MOVE ON

A 7B model in bfloat16 loads fine on a 16 GB GPU and serves single short prompts, then runs out of memory once several users hold long conversations. What grew?

A. The KV cache, which stores keys and values for every token in every active sequence and scales with length and concurrency.
B. The weights, because transformers promotes cached layers to float32 during multi-request serving to preserve numerical stability.
C. Activation memory, which grows with the number of tokens already generated in the same way the cache does.
D. Allocator fragmentation, which is the usual cause when a model loads successfully and later fails under identical shapes.

21 · 8 min

Spreading a model across whatever you have

THE ONE THING TO KEEP

`device_map="auto"` places layers across GPU, CPU and disk and streams weights shard by shard, which makes oversized models run correctly and extremely slowly — so offloading suits one-off jobs while quantization is the right answer for anything interactive.

One line that does a surprising amount

```
model = AutoModelForCausalLM.from_pretrained(
    name, dtype=torch.bfloat16, device_map="auto"
)
```

`device_map="auto"` comes from the accelerate library and does four things: inspect available devices and their free memory, load the model skeleton with-

out weights, decide which layers go where, then stream the weights in one shard at a time so the full model never has to exist in system RAM at once.

That last part is why it also helps on a single GPU. Without it, the naive path is: read 14 GB into CPU memory, then copy to the GPU. On a machine with 16 GB of RAM that fails before the GPU is involved at all. With it, shards move through and are released.

What the map looks like

```
print(model.hf_device_map)
# {'model.embed_tokens': 0, 'model.layers.0': 0, ...,
#  'model.layers.24': 'cpu', ..., 'lm_head': 'disk'}
```

Three kinds of destination, in descending order of speed:

- **A GPU index** — fast.
- `'cpu'` — the layer lives in system RAM and its weights are copied to the GPU when that layer runs, then evicted. Correct results, considerably slower.
- `'disk'` — the layer is read from an offload folder each time. Correct results, dramatically slower.

This is the honest mechanism behind "I ran a 70B model on my laptop and it took four minutes per token". It ran because every layer was fetched from disk for every single token. Nothing was broken; you were reading gigabytes from an SSD thousands of times.

Where device_map auto puts a layer

A GPU index	The layer stays on the card. Full speed.
cpu	Copied to the GPU when that layer runs, then evicted. Correct, several times slower.
disk	Read from an offload folder for every token of every request. Correct, hundreds of times slower.

Print hf_device_map and read it. Every entry a zero means the model fits and the speed is the hardware's. A cpu or a disk entry is the whole explanation for four minutes a token, and the fix is quantization rather than patience.

```
model = AutoModelForCausalLM.from_pretrained(
    name, device_map="auto",
    max_memory={0: "14GiB", "cpu": "24GiB"},
    offload_folder="./offload",
)
```

`max_memory` is worth setting explicitly. Left to itself, accelerate fills the GPU and leaves no headroom for the KV cache, so the model loads and then runs out of memory on the first long generation — which reads as a mysterious intermittent failure.

When offloading is the right answer

- **Yes:** a one-off batch job overnight; checking whether a large model is even worth pursuing; running something once to produce a dataset.
- **No:** anything interactive; anything you will run more than a few dozen times.

For the second case, quantization almost always beats offloading. A 70B model at 4-bit entirely in 40 GB of GPU memory is faster by orders of magnitude than the same model at bfloat16 with half its layers on disk — and usually better than dropping to a 13B model, because a heavily quantized large model tends to outperform a small model at full precision on the same memory budget. That comparison is worth running yourself on your own task rather than taking on trust.

Apple silicon

On a Mac, PyTorch's Metal backend gives you `mps`:

```
model = AutoModelForCausalLM.from_pretrained(name,
dtype=torch.float16).to("mps")
```

Unified memory means the GPU can address system RAM, so a 32 GB MacBook runs models that would need a 32 GB discrete card. The limits are real: some operations still fall back to CPU, bfloat16 support has been uneven

across versions, and throughput is well below a comparable NVIDIA card. For most Mac users, `llama.cpp` through GGUF is the faster and more reliable path, and the next lesson covers it.

Multiple GPUs, briefly

`device_map="auto"` across two cards gives you **pipeline parallelism**: layers 0–15 on card 0, 16–31 on card 1. Card 1 waits while card 0 works, so you get the memory of both and roughly the speed of one. It is a memory solution, not a speed solution, and people are regularly disappointed by that. Tensor parallelism — splitting each layer across cards so both compute at once — is what serving frameworks do, and it is a reason to reach for vLLM rather than tuning transformers further.

Do this now

Load any model with `device_map="auto"` and print `hf_device_map`. If every entry is `0`, the model fits and you are fine. If you see `'cpu'` or `'disk'`, you now know exactly why generation is slow, and the fix is quantization rather than patience.

BEFORE YOU MOVE ON

A large model loaded with `device_map="auto"` produces correct output at roughly one token every few seconds on a machine with one GPU and a fast SSD. What is the explanation?

A. Pipeline parallelism is serialising the layers, so only one part of the model computes at a time.

B. Automatic placement defaults to float32 when it cannot determine the dtype, quadrupling the arithmetic per token.

C. Some layers were placed on CPU or disk, so their weights are fetched and evicted for every token generated.

D. The KV cache has been offloaded to disk, so each new token requires re-reading the whole conversation history from storage.

22 · 10 min

Quantization: what you buy and what you pay

THE ONE THING TO KEEP

Quantization buys memory at a quality cost that concentrates in arithmetic, structured output and rare languages rather than spreading evenly — and at a fixed memory budget a heavily quantized large model usually beats a small model at full precision.

The idea in one paragraph

Weights are stored as 16-bit numbers because training needed that precision. Inference mostly does not. Quantization stores each weight in fewer bits — 8, 4, sometimes 3 — by keeping a scale factor per small group of weights and storing each weight as a small integer offset from it. Memory falls roughly in proportion to the bit width. Accuracy falls too, by an amount that depends on the scheme, the bit width, and heavily on the task.

The four routes you will meet

bitsandbytes — quantizes on the fly as the model loads, so it works with any transformers model with no prepared repository:

```
from transformers import BitsAndBytesConfig

bnb = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True,
)
model = AutoModelForCausalLM.from_pretrained(name, quantiza-
tion_config=bnb,
                                             device_map="auto")
```

The easiest route and the standard one for QLoRA training. Its weakness is speed: 4-bit bitsandbytes inference is often *slower* than bfloat16 on a card with enough memory, because weights are dequantized on the fly. You are buying memory, not throughput.

AWQ and **GPTQ** — pre-quantized repositories, produced once by running a calibration set through the model to decide which weights matter. Loading is a normal `from_pretrained`. These are faster at inference than bitsandbytes because they use kernels written for the packed format, and they are the usual choice for GPU serving. The cost is that somebody had to produce the repository, and its quality depends on a calibration set you did not see.

GGUF — the llama.cpp format, covered in the next lesson. CPU-first, excellent on Apple silicon, the practical answer for a laptop.

Illustrative of a pattern reported repeatedly: the loss concentrates in exact structured output rather than spreading evenly across chatting, and a small model suffers far more than a large one because it has less redundancy to spare. Chat answers at four bits look equivalent; the JSON is where the cost shows.

Native low-precision formats — newer hardware supports 8-bit floating point directly, and models are increasingly published in it. Where the hardware supports it, this is quantization with much less quality loss, and it is the direction the ecosystem is moving.

What it actually costs in quality

Published perplexity comparisons and community measurements land in roughly the same place, with wide variation by model:

- **8-bit** — the difference is usually below what you can detect on ordinary tasks. Treat it as close to free.
- **4-bit (nf4, AWQ, Q4_K_M)** — small degradation on chat and summarisation, larger on code, arithmetic, long structured output and instruction-following at the edges. Small models suffer more than large ones, because they have less redundancy to spare.
- **3-bit and below** — visible damage. Sometimes still worth it if the alternative is not running at all.

The pattern worth holding: **a 4-bit large model usually beats a 16-bit small model at the same memory budget.** A 4-bit 13B model and an fp16 7B model both take about 8 GB, and the 13B is generally better. This is counter-intuitive and it is one of the most useful facts in the whole area.

The failure mode nobody warns about

Quantization damage is not uniform. It concentrates in exactly the capabilities that are hardest to spot-check: multi-step arithmetic, precise JSON, rarely used languages, and long-range consistency. A model that chats indistinguishably at 4-bit may have quietly got worse at emitting valid JSON, which your parser turns into an error rate three weeks later.

If your application depends on structure or arithmetic, measure it. Thirty examples with exact-match scoring at both precisions takes twenty minutes and is the difference between a decision and a guess.

Calibration, and what it means for you

AWQ and GPTQ choose their scales by watching activations on a calibration set — typically a few hundred samples of general web text. If your domain is far from that text, the quantization was tuned for a distribution you are not in.

This is rarely catastrophic and it is a real reason a quantized model can underperform its published numbers on specialised text.

You can quantize with your own calibration data using the same libraries. For most people this is not worth it; for a narrow domain with a model you will run a million times, it is.

Choosing, in one pass

- **Does it fit at bfloat16?** Use bfloat16. Quantization is a constraint, not an improvement.
- **GPU, needs to fit, needs speed?** AWQ or GPTQ from a prepared repository.
- **GPU, needs to fit, will be fine-tuned?** bitsandbytes 4-bit plus LoRA.
- **CPU, laptop or Mac?** GGUF at `Q4_K_M`, or `Q5_K_M` if the memory is there.
- **Structured output or maths matters?** 8-bit, and measure.

Do this now

Take one model you can run at both bfloat16 and 4-bit. Run the same twenty prompts through each, including three that require exact output — a date reformatted, a small sum, a JSON object with four fields. Diff the results. The chat answers will look equivalent; the exact ones are where you will find the cost.

BEFORE YOU MOVE ON

A team moves a 13B model from bfloat16 to 4-bit on the same GPU. Chat quality seems unchanged, but their downstream JSON parser starts failing on about 4% of requests. What is the most likely explanation?

A. 4-bit weights change the tokenizer's handling of punctuation, so braces and quotes are emitted as different tokens.
B. The quantized repository was built with a different chat template, so the model no longer receives the formatting instruction.
C. The parser is stricter than before because the shorter 4-bit outputs omit optional fields the parser requires.
D. Quantization damage concentrates in precise output like exact syntax, which chat testing never exercises.

23 · 9 min

GGUF, and the laptop that has no GPU

THE ONE THING TO KEEP

GGUF packs weights, tokenizer and template into one self-contained file that llama.cpp runs on CPU, Apple silicon or a phone with no CUDA and no Python, making `Q4_K_M` the default free path for anyone without a graphics card.

The format built for the machine you have

`llama.cpp` is a C++ implementation of transformer inference with no Python and no CUDA requirement. It runs on an ordinary CPU, on Apple silicon through Metal, on an Android phone, and on a Raspberry Pi. GGUF is its file format, and between them they are the most important free path in this entire course, because they are what makes open models usable by people without a graphics card.

A GGUF file is self-contained: weights, tokenizer, architecture metadata and the chat template in one file. There is no `config.json` to match, no tokenizer to mismatch, no virtual environment to break. You download one file and run it. For anyone who has lost a day to a CUDA version conflict, this is a substantial thing.

Reading the quantization names

Inside a GGUF repository you will find a dozen files of the same model at different quantizations:

- `Q8_0` — 8-bit, about half the bf16 size, quality effectively identical.
- `Q6_K` — very close to Q8 at less memory.
- `Q5_K_M` — the quality-conscious default.
- `Q4_K_M` — **the usual recommendation**; roughly a quarter of the bf16 size, small quality loss.
- `Q3_K_M`, `Q2_K` — for when nothing else fits. Visible damage, especially on small models.

The `_K` means k-quants, which allocate more bits to the layers that need them rather than treating all weights alike. `_S`, `_M`, `_L` are small, medium and large variants within a level. `IQ` prefixes are a newer scheme that squeezes more quality out of very low bit widths at some speed cost.

For a first choice: take `Q4_K_M` if your RAM is tight, `Q5_K_M` or `Q6_K` if you have room. Files are named with their size, so the arithmetic is done for you.

Running it

The llama.cpp server, which gives you an OpenAI-compatible HTTP endpoint on your own machine:

```
llama-server -hf unsloth/Qwen2.5-7B-Instruct-GGUF:Q4_K_M --port
8080
```

That flag downloads directly from the Hub. Then anything that speaks the OpenAI API — including most chat front-ends and the `openai` Python package pointed at `http://localhost:8080/v1` — works against a model running entirely on your own hardware, with no key and no data leaving the machine.

From Python, without the server:

```
from llama_cpp import Llama

llm = Llama.from_pretrained(
    repo_id="unsloth/Qwen2.5-7B-Instruct-GGUF",
    filename="*Q4_K_M.gguf",
    n_ctx=4096,
)
print(llm.create_chat_completion(
    messages=[{"role": "user", "content": "Explain GGUF in two
sentences."}]
)["choices"][0]["message"]["content"])
```

Ollama and LM Studio wrap the same engine with a friendlier surface and pull GGUF files from the Hub too. Ollama is free and open source; LM Studio is free to use with a graphical interface. All three are running the same inference underneath.

Speed, honestly

On a recent laptop CPU with eight cores, a 7B model at `Q4_K_M` generates roughly 5–12 tokens per second. On Apple silicon with Metal, a 7B is often 20–40 tokens per second on an M-series chip with enough memory. A 3B model roughly doubles those figures; a 1B model is comfortably interactive on almost anything, including a mid-range phone.

For comparison, reading speed is about 4 tokens per second. So 7 tokens per second is slower than a fast API and faster than you can read, which for a private assistant is often enough.

Prompt processing is separate and usually faster per token than generation, but a 4,000-token document still takes real seconds before the first output token appears. This is the number that makes long-context work on CPU feel

bad, and `--n-gpu-layers` on a machine with any GPU at all — even a modest integrated one — offloads part of the model and helps disproportionately.

What GGUF does not do

- **Training.** GGUF is an inference format. Fine-tune in PyTorch, convert afterwards.
- **Every architecture.** Support is added per model family, so a brand-new architecture may have no GGUF for days or weeks.
- **Custom heads.** A classifier with a bespoke output layer is not what this path is for.

And one caution: because anyone can convert and upload, quality varies. A conversion done before a bug fix in llama.cpp can be subtly wrong. Prefer conversions from accounts that maintain them — the model's own organisation where it publishes one, or well-known conversion accounts — and check the repository's discussion tab before trusting an old file.

Do this now

Download a 1B or 3B instruct model at `Q4_K_M` — under 2.5 GB — and run it on whatever machine you have, including a phone if that is what you have. Time ten generations. That number is your floor, and everything else in this course is a choice about whether to exceed it.

BEFORE YOU MOVE ON

Why does a GGUF file avoid the tokenizer-and-weights mismatch failures that plague transformers loading?

A. llama.cpp validates the vocabulary size against the embedding matrix at load and refuses to start when they disagree.

B. GGUF repositories are restricted to official model organisations, so conversions cannot introduce mismatched components.

C. Tokenizer, template, weights and metadata all live in the one file, so there is nothing to mismatch.

D. The format stores text as raw bytes rather than token ids, so no vocabulary needs to be shared between the two halves.

24 · 9 min

ONNX, Optimum, and models that run in a browser tab

THE ONE THING TO KEEP

ONNX freezes a model into a portable graph that runs without PyTorch, and through transformers.js that graph runs inside a browser tab — which makes a static file a free, private, infinitely scalable deployment for any model small enough to download.

A different kind of portability

PyTorch is a research framework that happens to run in production. ONNX is an interchange format: a frozen computation graph plus weights, executed by runtimes written for speed on specific hardware. Exporting a model to ONNX gives you something that runs without PyTorch installed — on a CPU-only server, inside a mobile app, or in a web page.

For small models on CPU, ONNX Runtime is commonly two to four times faster than PyTorch eager execution, because the graph is known ahead of time and can be fused and optimised. For large generative models the gap narrows and the tooling gets harder, so the sweet spot is embeddings, classifiers, token taggers and small speech models — which is most of what an ordinary application actually needs.

Optimum is the bridge

```
pip install "optimum[onnxruntime]"
optimum-cli export onnx --model distilbert-base-uncased-fine-
tuned-sst-2-english out/
```

Then the API is deliberately the same shape as transformers:

```
from optimum.onnxruntime import
ORTModelForSequenceClassification
from transformers import AutoTokenizer, pipeline

model =
ORTModelForSequenceClassification.from_pretrained("out/")
tok = AutoTokenizer.from_pretrained("out/")
pipe = pipeline("text-classification", model=model,
tokenizer=tok)
```

Swapping `AutoModelForSequenceClassification` for `ORTModelForSequenceClassification` is the entire migration. Many Hub repositories already carry an `onnx/` subfolder, in which case `from_pretrained(name, subfolder="onnx")` skips the export step.

Optimum also wraps quantization for CPU — dynamic int8 quantization typically halves model size and speeds up CPU inference by 2–3× on encoder models, with a small accuracy cost you should measure on your own set.

Transformers.js: the model in the page

```
<script type="module">
import { pipeline } from "https://cdn.jsdelivr.net/npm/@hug-
gingface/transformers";

const pipe = await pipeline("sentiment-analysis");
console.log(await pipe("The train was delayed but the staff
were kind."));
</script>
```

This downloads an ONNX model into the browser and runs it in WebAssembly, or on the GPU through WebGPU where the browser supports it. No server, no API key, no request leaving the device.

The consequences are worth stating plainly, because they change what is possible for people without budget:

- **Hosting is a static file.** A GitHub Pages site or a static Hugging Face Space can host a working AI demo for nothing, forever, with no server to pay for or maintain.
- **The data never leaves.** For a tool handling medical notes, diaries or anything a user would not paste into a website, this is a genuine privacy property rather than a promise.
- **It scales for free**, because the user's device does the work.

The limits are equally plain. The model downloads on first visit — 30 MB for a small embedding model is fine, 2 GB for a chat model is not — though the browser caches it afterwards. WebAssembly is slower than native, roughly by a factor of two to five. WebGPU is much faster and is not available everywhere. And a phone on a weak connection will simply not finish a large download.

The practical ceiling today is models in the tens to low hundreds of megabytes: embeddings, classifiers, speech recognition for short clips, background removal, translation of short text. Within that ceiling it works genuinely well.

Other runtimes worth knowing exist

- **OpenVINO**, for Intel CPUs and integrated GPUs, also wrapped by Optimum. On an ordinary office laptop it is often the fastest CPU option.
- **Core ML**, for Apple devices, exported through `coremltools` or Optimum's exporters.
- **ExecuTorch** and **TensorFlow Lite**, for on-device mobile deployment.

All four solve the same problem — run this graph fast on this hardware — and all four trade flexibility for it. Once exported, you cannot easily change the model's shape, and dynamic control flow may not survive the conversion at all.

The honest caveat about export

Exports fail. A model with unusual control flow, custom operators or a `trust_remote_code` implementation may export with warnings and then produce subtly different numbers. Always verify:

```
import numpy as np
np.testing.assert_allclose(torch_logits, onnx_logits, rtol=1e-
3, atol=1e-4)
```

If that fails, the export is not equivalent, and the difference will show up as an accuracy drop nobody can explain later.

Do this now

Put the eight-line transformers.js example in an HTML file and open it locally. You are running a neural network with no install, no key and no server — and you can host that file for free. For a lot of projects, that is the whole deployment story.

BEFORE YOU MOVE ON

A team wants a free, private tool that classifies short notes typed by users, with no server costs. Which constraint decides whether transformers.js is viable?

A. Whether the browser supports WebGPU, since WebAssembly cannot execute transformer attention operations.
B. Whether the model is small enough that users will tolerate downloading it once on first visit.
C. Whether the model has an ONNX export available, since transformers.js can only load repositories published by Hugging Face itself.
D. Whether the notes can be batched, because per-request browser inference has fixed overhead that only batching amortises.

25 · 9 min

Where the free GPUs actually are

THE ONE THING TO KEEP

Free GPU compute genuinely exists — Colab, Kaggle, Spaces CPU and ZeroGPU — and every one of them is temporary, so the discipline that makes it usable is pushing artefacts to the Hub, checkpointing to resume, and debugging on a tiny model before switching the name.

Four real options, and what each costs you

This lesson is deliberately specific, because "use a GPU" is useless advice to somebody who does not have one, and the free tiers are genuinely usable if you know their edges.

Google Colab, free tier. A T4 with 16 GB of GPU memory, about 12 GB of system RAM, and a session that ends after roughly a few hours — sooner if the tab is idle, and availability is not guaranteed when demand is high. Good for: fine-tuning a small model with LoRA, running anything up to about a 7B

model in 4-bit, exploring a dataset. The T4 predates bfloat16 support, so use float16 and watch for the overflow failure.

Kaggle Notebooks. Around 30 hours of GPU per week, with a choice of a P100 or two T4s, and sessions up to nine or twelve hours. The weekly quota and the longer sessions make it the better free option for anything that needs to run overnight. Datasets can be attached from Kaggle or pulled from the Hub.

Hugging Face Spaces, free CPU. Two vCPUs and 16 GB of RAM, running continuously, sleeping after inactivity. No GPU. Right for embeddings, small classifiers, ONNX models and anything the previous lesson covered.

ZeroGPU on Spaces. Shared A100-class GPU time granted per function call rather than per Space, available to Spaces built with Gradio and allocated through a decorator. Free for the Space owner within quota, with a larger quota for paying accounts. It is the only way to publish a free public demo that needs a real GPU, and the constraint is per-call duration rather than total uptime — which shapes what you can build, favouring short inferences over long batch jobs.

Working with a session that will end

Everything about free compute follows from one fact: **the machine is temporary.** Write code that assumes it.

1. **Save to the Hub, not to the session.** A private model or dataset repository is durable storage that costs nothing.

```
python trainer.push_to_hub() # or
api.upload_folder(folder_path="./out", repo_id="you/run-1")
```

1. **Checkpoint often and resume.** `save_steps` set to something you can afford to lose, and `resume_from_checkpoint=True` on restart.
2. **Mount Drive on Colab** for intermediate files if you prefer, but the Hub survives the notebook and is easier to share.
3. **Do not download datasets you will stream.** The disk is small and the download is repeated on every restart.

The thing that wastes most free GPU time

Installing packages. A fresh session spends several minutes on `pip install` before any work happens, and if you are iterating on a bug you pay it repeatedly. Two habits help: pin versions so resolution is fast and reproducible, and get the code correct on CPU with a tiny model first, then switch the model name and the device. A script debugged on a 0.5B model on CPU almost always runs unchanged on a 7B model on GPU.

Costs, when free runs out

Rental prices move constantly, so treat these as orders of magnitude rather than quotes. Community GPU marketplaces rent a 24 GB card for something in the region of a few tens of US cents per hour, and an 80 GB card for a small number of dollars per hour. Hugging Face's own hosted options and the large clouds sit above that, buying reliability and integration.

The arithmetic that matters more than the rate: a LoRA fine-tune of a 7B model on a few thousand examples is a matter of hours, not days. At those rates it is a few dollars. People assume fine-tuning is expensive because full training of a foundation model is expensive; adapting one is not, and the training module returns to this.

The rules you must not break

Free tiers are offered on terms. Mining cryptocurrency, running a public service off a notebook, chaining accounts to evade quotas, and leaving jobs running with nobody watching are all against the terms of one or more of these providers and get accounts suspended. The quotas exist because the compute is genuinely scarce.

Do this now

Open a free Colab or Kaggle notebook and run `!nvidia-smi`. Write down the GPU name and the memory. Then compute, from the previous lessons, the largest model you could serve on it at bfloat16, at 8-bit and at 4-bit. That is

your working envelope, and knowing it stops you starting projects that were never going to fit.

BEFORE YOU MOVE ON

A learner fine-tunes on free Colab and loses six hours of work when the session ends. What change most directly prevents a repeat?

A. Reducing the model size so training finishes inside a single guaranteed session window.

B. Saving checkpoints to a Hub repository and resuming from the last one.

C. Mounting Google Drive so the notebook's filesystem persists across disconnections.

D. Switching to Kaggle, where the weekly quota and longer sessions remove the interruption risk.

---

26 · 9 min

What actually runs on eight gigabytes

THE ONE THING TO KEEP

On 8 GB and no GPU, embeddings, encoder classifiers, small Whisper models and 1–3B generative models at 4-bit all run usefully — and for a fixed narrow task a small specialised model usually beats a large general one on accuracy, speed and cost at the same time.

The constraint most courses ignore

A large share of the people reading this are on a laptop with 8 GB of RAM and integrated graphics, or on a phone. The honest answer is not "you cannot do this". It is that a specific and useful set of things runs well, and knowing which is more valuable than aspiring to the rest.

What fits, with numbers

On 8 GB of system RAM with no usable GPU, budget about 5 GB for models after the operating system and a browser:

- **Embedding models** — `all-MiniLM-L6-v2` is 90 MB and encodes hundreds of sentences a second on a CPU. `bge-small-en-v1.5` and the multilingual `paraphrase-multilingual-MiniLM` are similar. Semantic search over a few hundred thousand documents is entirely practical here.
- **Encoder classifiers** — DistilBERT-sized models, 250 MB, tens of items per second. Sentiment, intent, spam, topic, named entities.
- **Speech recognition** — `whisper-tiny` (75 MB) and `whisper-base` (145 MB) transcribe faster than real time on a CPU. `whisper-small` (480 MB) is slower than real time but usable for files.
- **Small generative models at 4-bit** — a 1B model is about 0.8 GB and runs at a readable speed; a 3B model is about 2 GB and is usable; a 7B model at `Q4_K_M` is about 4.4 GB and will run, slowly, if little else is open.
- **Image models** — background removal, OCR and small vision classifiers run fine. Image *generation* does not: even a distilled diffusion model wants several gigabytes and produces one image in minutes on a CPU.

What a phone does

A mid-range Android phone from the last few years runs, through llama.cpp-based apps or MLC, a 1B model at 4-bit at several tokens per second, and a 3B on a better handset. Whisper-tiny transcribes short clips. Browser-based transformers.js works on mobile browsers for small models, subject to the download.

The real constraints on a phone are thermal and battery: sustained generation heats the device and the system throttles it, so the tenth minute is much slower than the first. Design for bursts.

Choose a smaller model on purpose

The instinct is to reach for the largest model that fits. Often the better move is a **task-specific small model**:

- Sentiment: a 67M fine-tuned classifier beats an 8B general model on accuracy, runs 500 times faster, and costs nothing to serve.
- Entity extraction: a token classifier, same story.
- Search: an embedding model, not a chat model.

The general model is the right tool when the task is open-ended or changes weekly. When the task is fixed and narrow, a small specialised model is usually both better and cheaper, and the reflex to reach for a chat model for everything is one of the more expensive habits in the field.

Distillation, as a consumer of it

You do not need to train anything to benefit from distillation. `distil-whisper/distil-large-v3` is roughly six times faster than `whisper-large-v3` with a word error rate within about one percentage point on the data it was distilled for. `distilbert` is 40% smaller and 60% faster than BERT at about 97% of its accuracy on the standard benchmark suite. These are published, measured trades, and picking the distilled variant is often the single largest speed win available for no effort.

The caveat is in "for the data it was distilled for": a distilled model can be relatively worse outside the distribution it was trained to imitate. Check on your own examples, as always.

Practical habits for a small machine

- **Close the browser** before running a large model. Modern browsers routinely hold 2–4 GB.
- **Delete the cache you are not using.** `hf cache scan` then `hf cache delete`. The blob store fills silently.
- **Load once, reuse.** Never create a pipeline inside a loop.
- **Prefer ONNX or GGUF on CPU.** Both are meaningfully faster than PyTorch eager on the same hardware.
- **Use a Space for the heavy step.** Encode your documents once on a free Space or a Colab session, download the vectors, and search them lo-

cally forever afterwards. Heavy work is often a one-off, and one-off work does not need local hardware.

The last resort that is not a defeat

When the model genuinely cannot run locally, calling a hosted API for that one step is a reasonable engineering decision, not a failure of principle. What matters is knowing which step needs it, so you rent for the ten per cent that requires it rather than the whole pipeline.

Do this now

Build a search over your own files with `all-MiniLM-L6-v2`: embed a few hundred documents, store the vectors in a NumPy array, and search by cosine similarity. It runs on any machine in this lesson, it needs no GPU, and it is a genuinely useful tool. Module six builds it properly.

BEFORE YOU MOVE ON

A team needs sentiment labels on 200,000 short reviews using an ordinary laptop. Why is a 67M fine-tuned classifier usually the better choice than a 7B chat model at 4-bit?

A. Chat models cannot produce consistent labels because sampling introduces variation the classifier does not have.

B. Quantized models are unsuitable for classification because 4-bit damage concentrates in short outputs.

C. The classifier was trained for this one task, so it is typically more accurate as well as hundreds of times faster.

D. The 7B model's context window is wasted on short reviews, so most of its compute goes to padding rather than content.

27 · 8 min

Proving the change helped

THE ONE THING TO KEEP

A hardware change has to be reported as peak memory, time to first token, throughput and quality on your own thirty examples together, because each configuration trades along all four axes at once and any one number alone can be improved while the system gets worse.

Four numbers, not one

Every decision in this module — dtype, quantization, offloading, runtime, model size — trades along four axes at once. Report all four or you have not measured anything:

1. **Peak memory.** The constraint that decides whether it runs at all.
2. **Time to first token.** What an interactive user perceives.
3. **Throughput**, in tokens or items per second. What a batch job costs.
4. **Quality**, on your own examples. The one people skip, and the one that invalidates the rest.

A change that halves memory and costs three points of accuracy is a good trade or a disaster depending on the application, and you cannot tell which without the fourth number.

A harness small enough to actually use

```
import time, torch

def measure(pipe, prompts, max_new_tokens=128):
    _ = pipe(prompts[0], max_new_tokens=8)           # warm-up
    if torch.cuda.is_available():
        torch.cuda.synchronize(); torch.cuda.reset_peak_memo-
ry_stats()

    t0 = time.perf_counter()
    outs = [pipe(p, max_new_tokens=max_new_tokens) for p in
prompts]
    if torch.cuda.is_available():
        torch.cuda.synchronize()
    elapsed = time.perf_counter() - t0

    peak = torch.cuda.max_memory_allocated()/1e9 if torch.cud-
a.is_available() else None
    return {"seconds": round(elapsed, 2),
            "per_prompt": round(elapsed/len(prompts), 3),
            "peak_gb": round(peak, 2) if peak else None}
```

Three things this gets right that ad-hoc timing gets wrong:

- **Warm-up excluded.** The first call allocates, compiles kernels and pages weights in. It is two to ten times slower and it is not representative.
- **Synchronised.** CUDA work is queued asynchronously; without `synchronize()` you are timing Python.
- **Peak, not current.** Out-of-memory happens at the peak, which may last milliseconds.

Measure time to first token separately, because it is a different quantity:

```
t0 = time.perf_counter()
for chunk in streamer:
    ttft = time.perf_counter() - t0
    break
```

Report the tail

A mean of 300 ms with a 99th percentile of 4 seconds describes a system that one user in a hundred experiences as broken. Collect at least fifty runs and report the median and the 95th percentile. On shared free hardware the spread is wide and the mean is close to meaningless.

Measuring the quality side

This is twenty minutes of work and it is the part that gets skipped.

1. Fix **thirty examples** from your real inputs, with expected outputs where an expected output exists.
2. Run them through configuration A and configuration B.
3. Score whatever is objective: exact match, valid JSON, correct number, correct label.
4. Read the disagreements yourself. Thirty is few enough to read.

Thirty examples will not detect a two-point difference — it is too small a sample for that, and pretending otherwise is how people convince themselves of improvements that are noise. It will reliably detect the failure that matters here: a configuration that has quietly stopped producing valid structured output, or lost a language, or started truncating. That is a large, visible effect and thirty items find it.

Keep the examples in a file in the repository, not in a notebook cell. They will outlive the notebook.

Change one thing

The universal discipline, and the one people abandon when they are in a hurry. If you switch from bfloat16 on a T4 to 4-bit AWQ on an A10 with a new batch size, and it is faster, you have learned nothing transferable.

Write the configurations down before you run them:

```
A: bf16, batch 1, T4        → 8.1 tok/s, 14.2 GB, 30/30 valid
JSON
B: 4-bit nf4, batch 1, T4   → 6.4 tok/s,  5.1 GB, 27/30 valid
JSON
C: Q4_K_M GGUF, CPU laptop  → 7.9 tok/s,  4.4 GB, 29/30 valid
JSON
```

That table — three rows, four columns, an afternoon — is a decision. Note the result in row B that surprises people: 4-bit bitsandbytes is *slower* than bfloat16, because it buys memory rather than speed.

Do this now

Run your current setup through the harness and write down the four numbers. Even with nothing to compare against, you now have a baseline, and every later claim that something got faster becomes checkable rather than remembered.

BEFORE YOU MOVE ON

Why does timing generation on a GPU without calling `torch.cuda.synchronize()` typically produce a figure that is far too fast?

A. CUDA calls return once the work is queued, so the timer stops before the GPU has finished executing it.

B. The first call includes kernel compilation, which distorts the average unless it is excluded by a warm-up.

C. Without synchronisation the framework batches several generate calls together, so per-call time is divided across them.

D. Peak memory statistics are only flushed on synchronisation, so timings taken before it read from a stale counter.

CASE STUDY · MODULE 3

Eleven old machines against one new one

A coaching centre in Kota with eleven lab computers, eight gigabytes of RAM each, no graphics cards, and three hundred and forty students asking doubts after nine at night.

The doubt register was a paper book on the lab desk. A student wrote a question, a teacher answered it the next morning, and by then the student had moved on. Meena Joshi, who ran the centre, wanted something that answered at the moment the doubt existed.

The eleven machines were bought in 2019 for a computer-literacy course that no longer ran. Eight gigabytes each, integrated graphics, and a network that reached a printer. They were the centre's only fixed asset that was not a whiteboard, and they sat idle from four in the afternoon.

One constraint was settled before any arithmetic. After a previous incident with a homework app, parents had objected to student work leaving the building. A hosted interface was therefore not a cheaper option to be weighed. Uploading is a disclosure, and the disclosure was the thing being refused.

That left two shapes. Buy one machine with a sixteen-gigabyte graphics card, roughly what the centre spends on electricity in two months, and put a good model on it. Or run a small model on each of the eleven machines already in the lab.

Arvind, who taught physics, did not argue about it. He took thirty real doubts out of the register, covering definitions, unit conversions, two-step numericals and three questions where the right answer is that the question is missing information. Then he measured three configurations on all four numbers.

On the proposed new machine, a seven-billion-parameter instruct model at a four-bit quantization: peak memory 5.9 gigabytes, first token in 0.8 seconds, thirty-one tokens a second, and twenty-seven of thirty doubts answered acceptably by two teachers marking without knowing which machine produced what.

On a lab machine, a three-billion-parameter model at the same quantization: peak 2.4 gigabytes, first token in 2.6 seconds, nine tokens a second, twenty-two of thirty.

On a lab machine, the seven-billion model: peak 4.6 gigabytes, first token in eleven seconds, 2.4 tokens a second, twenty-six of thirty — and the machine could do nothing else while it ran.

The decision

The quality column said buy the machine. The third configuration said quality was not the whole story, because it answered nearly as well and was unusable.

And the number that decided it was not in the table at all. At nine at night there are not eleven students in the lab. There are sixty. One good machine is one queue, and a queue at nine is a student who has gone to bed. Eleven adequate machines answer everybody at once, worse.

The cost on that side was real: a model that gets twenty-two of thirty is wrong eight times, in front of children preparing for an examination, and wrong confidently.

There was a third option that never reached the table, and it is worth naming because most centres take it. Buy nothing, put a chat model's website on the lab machines and tell the students not to type anything personal. It is free, it is better than everything measured here, and it fails the one constraint that was not negotiable. Meena's position was that a rule students are asked to keep, about information they do not think of as personal, is not a control.

What made the measurement possible at all was that the thirty doubts already existed. Nobody had to invent test cases or imagine what a student might ask. They were in a paper register on the lab desk, written by the people the system was for, in the words those people use — including the three that were missing information, which turned out to be the ones that separated the configurations most sharply, because a small model at four bits will confidently answer a question that cannot be answered.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They bought nothing for six weeks. The three-billion model went on all eleven machines behind a local server started at boot, with a browser page on each and a line at the top of it saying the answers are produced by a program and should be checked with a teacher. Every doubt the students marked as unhelpful was written to a shared file.

After six weeks the file said what the thirty doubts had hinted at: the small model was reliable on definitions, formulas and unit conversions, and poor on multi-step numericals — which is exactly where four-bit quantization damage concentrates. So they bought one machine, put the seven-billion model on it, called it the hard doubts station and sat a teacher beside it for an hour each evening.

One machine instead of eleven, and a decision made from twelve measured numbers rather than an argument about what feels fast. The thirty doubts stayed in a file in the repository. When a stronger four-billion model appeared four months later, re-running them took an afternoon.

The hosted option was ruled out before any measurement. Was that a technical decision?

No, and it is important that it was not treated as one. Sending student work to somebody else's machine is a disclosure, and the parents had refused the disclosure rather than the latency. A constraint of that kind sits above the measurement: it removes options from the table rather than scoring them. Local was then the only shelf, and the whole question became which local configuration.

The third configuration scored twenty-six of thirty, almost matching the new machine. Which of the four numbers ruled it out, and what does that show about reporting one number?

Time to first token, at eleven seconds, and throughput at 2.4 tokens a second — slower than a person reads. Quality alone would have made it look like the cheap winner. Every configuration in this module trades along peak memory, first-token latency, throughput and quality at once, so any one of them can be improved while the system gets worse. Reporting all four is what made the comparison honest.

What did keeping the thirty doubts in a file, rather than in a notebook cell, buy the centre later?

A re-runnable decision. When a new model appeared, the question was not whether it felt better but whether it scored better on the same thirty items that the current choice had been made from, and that took an afternoon. The set also detects the failures that matter at that size — a model that has lost a capability, or a configuration that stopped producing usable answers — which is exactly what thirty examples are good for.

MODULE 3 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 You delete the familiar filenames under snapshots/ in the Hugging Face cache and no disk space is freed. Why not?

A. Those names are links; the bytes live under blobs/ and are still referenced.

B. The files are already compressed, so removing them frees very little space.

C. The operating system reclaims cache space only after the machine is rebooted.

D. The cache is memory-mapped, so deletions take effect at the next download.

2 Why is bfloat16 usually preferred to float16 on hardware that supports both?

A. It halves the memory again, at one byte per parameter instead of two.

B. It is more precise, so its outputs are closer to the float32 result.

C. It keeps float32's exponent range, so a spiking activation does not overflow.

D. It is the only half-precision format that ordinary processors compute natively.

3 hf_device_map shows several layers on 'disk' and generation takes minutes per token. What is happening, and what fixes it?

A. The weights downloaded badly; fetch them again and the speed returns.
B. The GPU is thermally throttled under sustained load, so lowering the batch size will help.
C. Disk offload is a known accelerate bug, so pin an earlier version.
D. Those layers are read from disk for every token; quantize so the model fits.

4 At a fixed budget of about eight gigabytes, which usually performs better on general tasks?

A. A 13B model at four bits, because a larger model has more redundancy to spare.
B. A 7B model at float16, because full precision preserves everything training learned.
C. Either one; quantization damage is proportional to the parameter count.
D. A 3B model at eight bits, because smaller models survive quantization better.

5 A model that chats indistinguishably at four bits starts failing a JSON parser three weeks later. What is the mechanism?

A. Quantized weights drift while the file is memory-mapped over many runs.
B. Quantization damage concentrates in exact structured output rather than spreading.
C. Four-bit models cannot emit braces, because punctuation tokens are rare.
D. The parser must have been upgraded, because quantization does not affect formatting.

6 You time a GPU generation loop with a plain Python timer and get a suspiciously high throughput figure. What did you actually measure?

A. How fast Python can queue asynchronous work, because CUDA calls return early.

B. The kernel time, minus the time spent moving your tensors onto the device.

C. The throughput of one warmed-up batch, repeated across the whole of the timed run.

D. Wall-clock time, which on a GPU is always lower than the true compute time.

MODULE 4

Spaces: putting something in front of people

A Space is a git repository that the Hub builds and runs, with a public URL and a free tier that never expires. This block takes you from a forty-line Gradio script to an application with state, secrets, storage and a build you can debug — and then shows how to call somebody else's Space as an API, which turns the whole platform into infrastructure rather than a gallery.

> BY THE END OF THIS MODULE YOU CAN
>
> Build, configure and debug a Space that other people can use — choosing the SDK, pinning versions in the README, holding secrets and state correctly, picking hardware against a sleep policy — and call an existing Space programmatically as an API instead of reimplementing it

28 · 9 min

A Space is someone else's computer, already configured

THE ONE THING TO KEEP

Free CPU hardware runs one-file-at-a-time work well and cannot run image generation or fast chat, and a slow Space is usually a cold start or a shared queue rather than a fault.

The thing you were about to install already runs in a browser

The hardest part of using an open model is not the model. It is CUDA versions, a Python environment that fights you, six gigabytes of dependencies and a laptop that does not have the memory. A Space skips all of it. Someone has already written the code, pinned the dependencies and pointed it at a model, and the Hub runs the whole thing behind a web page.

Search the Spaces tab for what you want in plain words — *remove background*, *transcribe audio*, *upscale*, *caption image*, *extract table from pdf*. Sort by likes or trending to find the ones people actually use. A phone browser works perfectly well; a Space is just a web page, and the only real constraint on a phone is uploading a large file over a mobile connection.

Why it is asleep, and why there is a queue

Two things confuse new users, and both have simple mechanisms.

Sleeping. A Space on free hardware pauses itself after a stretch of inactivity — 48 hours by default — to stop the Hub running thousands of idle containers. Your visit wakes it. The container has to start, install nothing (it is already built) but load the model into memory, which for a mid-sized model is tens of seconds. A blank page or a *Space is starting* message is not a broken Space. Wait a minute before concluding anything.

What free CPU hardware does, and what it cannot

Runs well, one file at a time	Does not run, and the arithmetic says why
• Background removal and matting	• Modern text-to-image: minutes to tens of minutes a picture
• Whisper at tiny, base or small on short clips	• A 7B chat model at conversational speed
• Captioning, OCR, document layout extraction	• Anything real-time, including video
• Classification, entity extraction, small translation	• Large models at full precision, which do not fit 16 GB
• Embedding text for search	

Two virtual CPUs, sixteen gigabytes of RAM, no GPU, free permanently. The right-hand column is stopped by arithmetic rather than by software, so no amount of configuration moves an item across. If you find a free Space doing one of them, it is on ZeroGPU or its owner is paying.

The queue. Gradio, the framework most Spaces use, processes requests one at a time by default. If nine other people clicked *Submit* just before you, you are ninth. The interface usually tells you your position. On a popular Space at a busy hour this is the whole of the experience, and it is the strongest argument for duplicating it, which is the next lesson.

The hardware shelf

Every Space runs on a tier chosen by its owner.

- **CPU basic**, the free default: roughly 2 virtual CPUs, 16 GB of RAM, around 50 GB of disk, no GPU. Free with no time limit.
- **ZeroGPU**: a shared pool of high-end GPU capacity, allocated for the duration of a single function call. Free to use with a per-person quota, larger for PRO subscribers. Because the GPU is acquired per call, there is start-up overhead on each request and long-running or stateful GPU work does not fit the model well.
- **Rented GPUs**: T4, L4, A10G and up, billed by the hour. Roughly forty cents to a few dollars an hour depending on the card, as of late 2025. Billed for wall-clock time while the Space is running, not for requests served.

What free CPU genuinely does

More than people expect, as long as the work is one file at a time:

- Background removal and matting.

- Speech to text with Whisper at the tiny, base or small sizes, on clips of a few minutes.
- Image captioning, OCR, document layout extraction.
- Text classification, named entity extraction, translation with small models.
- Embedding text for search.
- Modest image upscaling, audio trimming, format conversion, subtitle generation.

And what it does not do, with the reason:

- **Text-to-image at modern quality.** SDXL or Flux on two CPU cores is minutes to tens of minutes per image, and the larger models will not fit in 16 GB at full precision at all. The arithmetic, not the software, is what stops you.
- **Chat with a 7B model at conversational speed.** It will run and it will produce a word every second or two. Fine for a demo, unusable for a conversation.
- **Video generation, or anything real-time.**

If a Space you find is doing one of those things for free, it is on ZeroGPU or its owner is paying, and both mean queues.

Before you upload anything

A Space is a stranger's program. Open its Files tab and read `app.py` — most are under two hundred lines and you can see exactly what happens to your file. Check the *Community* tab too: the discussions there are where people report that a Space has been broken for three weeks.

And apply the rule from the last lesson without negotiating with yourself: if the file is a client's, is under an agreement, or contains someone's personal data, do not upload it to a Space you do not control. Duplicate it instead, which takes one click and is the next lesson.

Do this now

Find a Space that does something you currently pay for or currently do by hand. Run it once on a real file. Then open its `app.py` and find the line where the model is loaded — that name is a model on the Hub you could run yourself.

BEFORE YOU MOVE ON

A Space that ran fine last month now shows a starting message for about a minute, then works normally. Later that day it responds only after a long delay, with a position number on screen. What is going on?

A. It slept after a period of inactivity and had to reload the model, and the later delay is the shared request queue with other users ahead of you.

B. The owner moved it from a rented GPU down to free CPU hardware, so everything now runs several times slower.

C. Your account has hit a usage quota, so requests are being throttled until it resets.

D. The underlying model repository was updated, so the Space is redownloading weights on every request.

29 · 9 min

Forty lines from nothing to a public URL

THE ONE THING TO KEEP

`gr.Interface` turns a Python function plus input and output components into a public web application, and the two things that decide whether strangers can use it are loading the model once at import rather than per request, and shipping examples plus an honest statement of the limits.

The whole thing

Create a Space on the Hub (New → Space, SDK: Gradio), and put two files in it.

`requirements.txt`:

```
transformers==4.46.2
torch==2.5.1
```

`app.py`:

```
import gradio as gr
from transformers import pipeline

clf = pipeline(
    "text-classification",
    model="cardiffnlp/twitter-roberta-base-sentiment-latest",
)

def classify(text):
    if not text.strip():
        return {}
    scores = clf(text, top_k=None)[0] if isinstance(clf(text,
top_k=None)[0], dict) \
             else clf(text, top_k=None)
    return {d["label"]: float(d["score"]) for d in scores}

demo = gr.Interface(
    fn=classify,
    inputs=gr.Textbox(lines=4, label="Text", placeholder="Paste
a review"),
    outputs=gr.Label(num_top_classes=3, label="Sentiment"),
    title="Sentiment, three ways",
    description="Roberta trained on tweets. English only. Not
calibrated.",
    examples=[
        ["The delivery was late but the food was excellent."],
        ["Fine, I suppose, if you enjoy waiting."],
    ],
    flagging_mode="never",
)

if __name__ == "__main__":
    demo.launch()
```

Commit both. The Space builds, installs, starts, and gives you a public URL that works on a phone. On free CPU hardware the first build takes a few minutes, most of it downloading torch.

What `gr.Interface` is doing

It inspects the function and the component types, builds a form and a results panel, wires the submit handler, handles errors, and exposes a REST endpoint

and a queue behind the scenes. You describe inputs and outputs; it builds the application.

The components you will reach for most:

- `gr.Textbox`, `gr.Number`, `gr.Slider`, `gr.Dropdown`, `gr.Checkbox` — inputs.
- `gr.Image`, `gr.Audio`, `gr.Video`, `gr.File` — media, with upload and record widgets built in.
- `gr.Label` — a dictionary of label to score, rendered as bars.
- `gr.Dataframe`, `gr.JSON`, `gr.Markdown`, `gr.Gallery`, `gr.Plot` — outputs.

Passing a bare string like `"text"` instead of `gr.Textbox()` works and is common in tutorials. Use the component object once you want a label, a placeholder or a line count — which is immediately.

Load the model once, outside the function

In the script above, `pipeline(...)` runs at import, once, when the Space starts. Put it inside `classify` instead and every single request reloads the weights — turning a 200 ms response into a 30-second one and, on a busy Space, exhausting memory. This is the most common performance bug in beginner Spaces and it is invisible when you test alone.

The details that make it usable by strangers

Examples. `examples=[...]` renders clickable rows. A visitor who does not know what to type will click one; a visitor faced with an empty box leaves. This single argument changes how many people actually try your Space.

An honest description. "English only. Not calibrated." in the description costs nothing and prevents somebody deploying your demo against Hindi support tickets. Every limitation you know belongs on the page, not in your head.

Empty-input handling. The `if not text.strip()` guard stops the first visitor seeing a red traceback box. Gradio surfaces exceptions in the UI, which

is good for you and alarming for them.

`flagging_mode="never"`. Flagging writes user submissions to a file on a container that gets destroyed. Unless you have set up storage and said so in a privacy note, turn it off rather than silently collecting text people typed.

Running it locally first

Everything above runs on your own machine:

```
pip install gradio
python app.py          # http://127.0.0.1:7860
```

Debug locally where the loop is seconds, and push when it works. `demo.launch(share=True)` gives a temporary public link that tunnels to your laptop — useful for showing somebody something for an hour, and not a deployment, since it dies with the process.

The free tier, precisely

Two vCPUs, 16 GB RAM, public, and it sleeps after a period of inactivity, waking on the next visit with a cold start of anything from seconds to a couple of minutes depending on what has to be downloaded. There is no time limit and no card required. For an embedding model, a classifier, an ONNX pipeline or a small Whisper model, this is a genuinely free permanent deployment.

Do this now

Ship a Space today, however small. The gap between a notebook and a URL you can send to somebody is the largest gap in this course, and it is forty lines wide. Everything after this lesson is refinement.

BEFORE YOU MOVE ON

A Space responds in 200 ms when the developer tests it and takes 30 seconds per request once a few people use it at once. What is the most likely cause?

A. Free CPU hardware throttles under concurrency, so per-request latency rises sharply with the number of active users.
B. The model is being loaded inside the handler function, so every request re-reads the weights from the cache.
C. The Space went to sleep between tests, and the reported time includes the cold start on each wake.
D. Gradio's queue serialises requests by default, so users wait for every earlier request to complete.

30 · 9 min

Blocks, when Interface is no longer enough

THE ONE THING TO KEEP

`gr.Blocks` separates layout from event wiring, and the rule that prevents the worst class of Space bug is that module-level variables are shared by every visitor while `gr.State` is per-session — so the model is shared and everything about a user is not.

When to move

`gr.Interface` is one function, one form. Move to `gr.Blocks` when you need layout, several buttons doing different things, outputs that feed back into inputs, or anything conversational.

```
import gradio as gr

with gr.Blocks(title="Transcribe and summarise") as demo:
    gr.Markdown("## Upload audio, get a summary")

    with gr.Row():
        with gr.Column(scale=1):
            audio = gr.Audio(type="filepath", label="Audio")
            lang = gr.Dropdown(["auto", "en", "hi"],
value="auto", label="Language")
            go = gr.Button("Transcribe", variant="primary")
        with gr.Column(scale=2):
            text = gr.Textbox(label="Transcript", lines=10)
            summary = gr.Textbox(label="Summary", lines=4)
            sum_btn = gr.Button("Summarise this")

    go.click(fn=transcribe, inputs=[audio, lang], outputs=text)
    sum_btn.click(fn=summarise, inputs=text, outputs=summary)

demo.queue().launch()
```

The pattern is always the same: declare components inside the `with` block, then wire events with `component.click(fn=..., inputs=..., outputs=...)`. `inputs` and `outputs` are lists of components, and Gradio passes their current values in and writes returned values back.

Chat, which is a component

```
def respond(message, history):
    messages = history + [{"role": "user", "content": message}]
    out = pipe(messages, max_new_tokens=256)
    return out[0]["generated_text"][-1]["content"]

gr.ChatInterface(respond, type="messages").launch()
```

`gr.ChatInterface` gives you the message list, the history, retry, undo and clear. `history` arrives in exactly the `{"role", "content"}` shape that `apply_chat_template` wants, which is not a coincidence and saves a translation layer.

For streaming, make the function a generator and yield the accumulating string:

```
def respond(message, history):
    partial = ""
    for chunk in stream_from_model(message, history):
        partial += chunk
        yield partial
```

Yield the whole string so far, not the chunk. Yielding chunks replaces the display each time and the user sees one word flickering.

State, and the mistake that leaks data between users

This is the most important paragraph in the lesson.

A Space is one process serving everybody. A module-level variable is **shared by every visitor**:

```
history = []              # WRONG: one list for the whole in-
ternet

def chat(msg):
    history.append(msg)   # every user appends to the same list
```

One person's conversation appears in another's. On anything handling personal text this is a data breach with a two-line cause, and it is easy to write because it works perfectly when you are the only user.

The fix is `gr.State`, which is per-session:

```
with gr.Blocks() as demo:
    session = gr.State([])          # separate list per browser
session
    box = gr.Textbox()
    out = gr.Textbox()

    def chat(msg, hist):
        hist = hist + [msg]
        return "\n".join(hist), hist

    box.submit(chat, [box, session], [out, session])
```

The state goes in as an input and comes back as an output. It lives as long as the browser session and dies with it. `gr.BrowserState` persists in local storage across reloads for preferences; anything that must outlive the session belongs in the storage lesson later in this module.

Shared module-level objects are correct for exactly one thing: the model, which is read-only and should be shared.

Making the interface respond to itself

```
mode = gr.Radio(["summarise", "translate"], value="summarise")
target = gr.Dropdown(["hi", "bn", "ta"], visible=False)

mode.change(
    fn=lambda m: gr.update(visible=(m == "translate")),
    inputs=mode, outputs=target,
)
```

`gr.update()` changes a component's properties rather than its value — visibility, label, choices, interactivity. This is how a form stops showing fields that do not apply, which matters more for comprehension than it looks.

Progress, so the wait is legible

A ten-second handler with no feedback reads as broken. Gradio will show a spinner on its own, and you can do better in four lines:

```
def transcribe(path, progress=gr.Progress()):
    chunks = split_audio(path)
    out = []
    for c in progress.tqdm(chunks, desc="Transcribing"):
        out.append(asr(c)["text"])
    return " ".join(out)
```

Add the `progress` parameter and Gradio wires a bar into the interface. On free hardware, where everything is slower than you would like, this is the difference between a visitor waiting and a visitor reloading — and reloading queues a second job while the first is still running, which makes the problem worse for everybody.

The queue, briefly

`demo.queue()` puts requests in a managed queue with position and estimated wait shown to the user. It is on by default in current versions and worth knowing about because `concurrency_limit` is how you stop eight simultaneous visitors from exhausting the memory of a free container. A queue that makes people wait visibly is better than a Space that crashes.

Do this now

Take a Space you have built with `Interface` and rewrite it in `Blocks` with two buttons, where the second acts on the output of the first. Then add a `gr.State` and check that opening the Space in two browsers keeps the two sessions apart.

BEFORE YOU MOVE ON

A chat Space keeps conversation history in a module-level list. It works perfectly in testing and, once public, users report seeing fragments of other people's conversations. Why?

A. Gradio's queue reuses request objects across sessions, so history written by one request is visible to the next.
B. The Space caches responses by input hash, so identical opening messages return another user's continuation.
C. Browser sessions share cookies on the huggingface.co domain, so session identifiers collide between visitors.
D. One process serves every visitor, so a module-level list is a single object shared across all of them.

31 · 9 min

Duplicate it, and the app becomes yours to change

THE ONE THING TO KEEP

Duplicating a Space gives you a private copy with no queue, but paid hardware bills for wall-clock time and does not sleep the way free hardware does — set the sleep timer before you upgrade.

One click is the whole difference

On any Space, the menu at the top right has *Duplicate this Space*. You choose an owner, a name, public or private, and hardware — free CPU is the default and stays the default unless you pick otherwise. A few seconds later there is a copy in your account, running your build, with no queue but yours.

This is the single most valuable thing on the Hub for someone who does not write much code, because it converts *using a tool* into *owning a tool*.

Duplicating is also how you handle sensitive files: the code now runs in a container you control, and you can make it private so nobody else can reach it.

What you actually get

Usually three files.

- **`app.py`** — the whole program. For a Gradio Space this is typically a model load, a function, and a description of the interface. Under two hundred lines is normal.
- **`requirements.txt`** — the Python packages to install at build time.
- **`README.md`** — which is also the deployment configuration, because the YAML block at the top is read by the Hub:

```
---
title: Background Remover
sdk: gradio
sdk_version: 4.44.0
app_file: app.py
suggested_hardware: cpu-basic
---
```

Edit any file in the browser. Committing triggers a rebuild, and the Logs tab shows the build output and the running application's output separately. When something goes wrong, the traceback is one click away in there — this is the habit that separates people who can fix a Space from people who delete it and try another.

The failure you will meet first

You duplicate a Space written two years ago. It builds, then dies with a `TypeError` about an unexpected keyword argument. Nothing you did is wrong. The `sdk_version` line was absent or loose, the build picked a newer Gradio, and an argument was renamed between major versions.

The fix is to pin: set `sdk_version` in the README to the version the original was written against, and pin the packages in `requirements.txt` that matter.

It is the same lesson as pinning a model revision in lesson one, and for the same reason — anything unpinned changes under you eventually.

Hardware is a rental, so treat it like one

Settings lets you change hardware on a running Space. This is where people get a surprising bill, and the mechanism is worth stating plainly.

Free hardware sleeps. Paid hardware does not, by default. The 48-hour sleep that pauses idle free Spaces is a cost-control measure for the Hub, and it does not apply to a GPU you are renting. A rented GPU bills for wall-clock time from the moment it starts until you pause it or downgrade it, whether or not anyone uses the Space. Leaving one on over a weekend costs the same as using it hard for the weekend.

So: rent the GPU, do the work, and then either pause the Space or set it back to CPU basic. In Settings there is also a sleep timer for paid hardware — set it to something small, like fifteen minutes, the moment you upgrade. Do it before you start the job, not after, because *after* is when you forget.

Secrets and variables

Settings has a *Variables and secrets* section. Variables are visible to anyone; secrets are stored encrypted, handed to your code as environment variables, never shown on the page, and not carried over when somebody duplicates your Space — they are prompted for their own. Read them normally:

```
import os

token = os.environ["HF_TOKEN"]
```

What you must never do is put the key in `app.py`. A public Space's files are public, permanently and to everyone, and the next lesson is about exactly what that costs.

When the free GPU you need is somewhere else

If what you want is a one-off render rather than a hosted app, renting is often the wrong shape. Kaggle notebooks give roughly thirty GPU-hours a week free with a verified phone number, which is a genuinely generous allocation and enough to fine-tune a small model. Colab's free tier is smaller and can be interrupted. Both let you pull models and datasets straight from the Hub, so the skills transfer completely.

Do this now

Duplicate one Space you use. Change one visible string in `app.py` — a title, a label, a default value — commit, and watch the rebuild in the Logs tab. That loop is the whole of deploying software on this platform.

BEFORE YOU MOVE ON

Someone duplicates a Space, switches it to a rented A10G on Friday afternoon to render a batch, finishes in twenty minutes, closes the browser tab and goes home. On Monday the bill is far larger than expected. What is the mechanism?

A. Rebuilds triggered by upstream package updates over the weekend each consumed GPU time.

B. The Space stayed publicly reachable, so other people used it and their requests were billed to the owner.

C. Closing the browser tab did not stop the container, and it kept running until the next visit woke it properly.

D. Rented hardware bills for wall-clock time and does not auto-sleep the way free hardware does, so the GPU was held all weekend doing nothing.

32 · 8 min

The README front matter is the deployment config

THE ONE THING TO KEEP

A Space's deployment configuration is YAML front matter in `README.md`, so it is versioned like code — and the two fields that decide whether it still builds in six months are `sdk_version` and pinned dependency versions in `requirements.txt`.

Where the settings live

A Space's configuration is YAML at the top of `README.md`. Not a dashboard, not a hidden file — the readme:

```
---
title: Sentiment Three Ways
colorFrom: indigo
colorTo: blue
sdk: gradio
sdk_version: 5.9.1
app_file: app.py
pinned: false
license: apache-2.0
short_description: Three-way sentiment on English text, not
calibrated.
---

Everything below the fence is the readable page.
```

There is also an `emoji` field, which takes one character and appears on the gallery card; it is omitted above because nothing depends on it. This is the same pattern as model and dataset cards, and it means the deployment configuration is versioned, diffable and reviewable in a pull request like anything else.

The fields that actually change behaviour

`sdk` — `gradio`, `streamlit`, `docker` or `static`. It decides the whole build.

`sdk_version` — the one people leave off, and the one that causes the mystery. Omit it and the Space builds against whatever the platform currently considers default. Gradio 4 to 5 changed component APIs; a Space that worked in March can fail to start in October with no commit in between. Pin it. If a Space suddenly breaks, this is the first thing to check.

`app_file` — defaults to `app.py`. Set it if your entry point is elsewhere.

`python_version` — pin this too when a dependency is version-sensitive.

`models` and **`datasets`** — lists of repository ids your Space uses. They create links on the Hub in both directions, which is how the "Spaces using this model" counter from module one gets populated. It is metadata, and it is how other people find your work.

`hardware` and **`suggested_hardware`** — the second is a hint to anyone duplicating your Space, so they do not try to run a diffusion model on free CPU and conclude it is broken.

`pinned` — pins it to your profile.

`requirements.txt`, and pinning properly

```
gradio==5.9.1
transformers==4.46.2
torch==2.5.1
sentencepiece==0.2.0
```

Unpinned dependencies are the second-largest cause of a Space that stops building. Install resolution happens at build time, so a new release of any transitive dependency can break you overnight while your code is untouched.

Three practical notes:

- **`torch` is large.** On free CPU hardware, installing the CPU-only build (`torch --index-url https://download.pytorch.org/whl/cpu`) cuts the

image size and the build time substantially, because the default wheel carries CUDA libraries you will never use.

- **System packages** go in a `packages.txt`, one per line — `ffmpeg`, `poppler-utils`, `tesseract-ocr`. Audio and PDF Spaces fail without this and the error appears at runtime, not at build.
- `pre-requirements.txt` exists for the rare case where something must be installed before the main resolution, such as a specific `pip` version.

Files that do not belong in the repository

A Space is a git repository, so everything in it is public unless the Space itself is private. Weights, datasets and credentials should live in their own repositories and be fetched at startup, not committed here. Use `.gitignore`, and use the Hub:

```
from huggingface_hub import snapshot_download
local = snapshot_download("your-name/your-private-model",
token=os.environ["HF_TOKEN"])
```

That keeps the Space small, the build fast, and the weights versioned where they belong.

The readme below the fence matters too

It renders under the app on the Space page. What to put there, in order of how much it helps a visitor:

1. What the thing does, in one sentence, without marketing.
2. What it will do badly — language coverage, length limits, known failure cases.
3. Which model it uses, linked, with the revision if you pinned one.
4. Whether anything a user types is stored. Say "nothing is stored" if nothing is, because visitors cannot tell from the outside and will assume the worst or the best, both wrongly.

5. The licence of your code, which is not automatically the licence of the model inside it.

Do this now

Open the readme of any Space you have built or duplicated and add two things: a pinned `sdk_version`, and a sentence saying whether user input is stored. Both take a minute. The first prevents a failure; the second is the difference between a demo and something a stranger can trust.

BEFORE YOU MOVE ON

A Space that has not been committed to in eight months suddenly fails to start, with an error inside a Gradio component constructor. What is the most likely cause?

A. No `sdk_version` is pinned, so the Space rebuilt against a newer Gradio whose component API changed.

B. The free-tier container was migrated to newer hardware, and the compiled wheels in the cache no longer match the architecture.

C. The model repository it loads changed on `main`, and the new revision is incompatible with the component types.

D. Spaces expire their build cache after six months, forcing a rebuild that exceeded the free tier's disk limit.

33 · 9 min

Hardware, sleep, and the bill that runs while you are asleep

THE ONE THING TO KEEP

Paid Space hardware bills for wall-clock uptime rather than requests and does not sleep unless you set the timer first, while ZeroGPU attaches a GPU per function call — which is the only route to a free public demo that needs real acceleration.

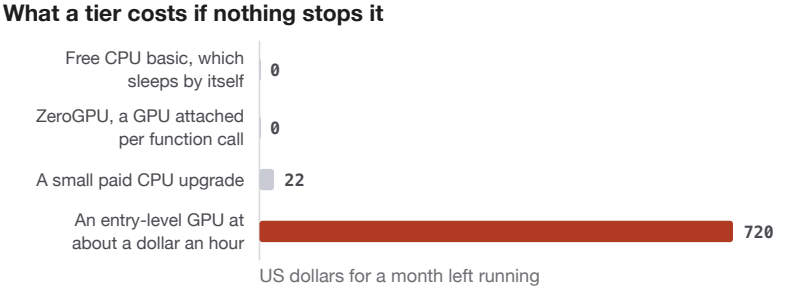
The shape of the pricing

Free CPU hardware is free permanently and sleeps when idle. Every upgrade past it bills **per hour of wall-clock time the Space is running**, not per request and not per unit of compute. A Space serving three visitors a day on a paid GPU costs exactly what one serving three thousand costs, because the meter is time.

This is the single most expensive misunderstanding on the platform. People upgrade to test something, close the tab, and find a bill weeks later. Approximate rates change, so check the current figures before committing, but the order of magnitude is worth carrying: a small paid CPU upgrade is a few US cents an hour, an entry-level GPU in the region of a dollar an hour, and a large-memory GPU several dollars an hour. A dollar an hour is about 720 dollars a month if nothing stops it.

Set the sleep timer before you upgrade

In Settings there is a sleep time — after this many minutes of no traffic, the Space stops and billing stops. The order matters:



Rates move and these are orders of magnitude rather than quotes. The shape does not move: paid hardware bills wall-clock time and does not sleep unless you set the timer first, so a Space serving three visitors a day costs exactly what one serving three thousand costs.

1. Set the sleep timer.
2. Then change the hardware.

Doing it the other way leaves a window, and windows become weekends. Free CPU Spaces sleep on their own schedule; **paid hardware does not sleep unless you tell it to.** That asymmetry is where the surprise bills come from.

Waking from sleep costs a cold start: container start, dependency layer restore, model download if the cache was cleared, model load. Ten seconds for a small ONNX model, two or three minutes for a large one. If the wait is unacceptable, the answer is either persistent storage so the weights are already there, or paying for uptime — and those are the only two answers.

ZeroGPU: the interesting option

ZeroGPU allocates GPU time **per function call** rather than reserving a machine. You mark the functions that need a GPU:

```
import spaces

@spaces.GPU(duration=60)
def generate(prompt):
    return pipe(prompt).images[0]
```

Between calls, no GPU is held. The Space itself runs on CPU; when `generate` is invoked, a slice of an A100-class GPU is attached for up to `duration` seconds, then released.

What this buys: a free public demo that genuinely uses a GPU, which is otherwise impossible. What it costs:

- **Gradio only**, and the decorator has to be applied to the right functions.
- **A per-call ceiling.** Set `duration` honestly — asking for 120 seconds when you need 20 wastes quota, and asking for 20 when you need 60 gets your call killed mid-generation.
- **Cold attachment.** There is overhead on each allocation, so many tiny GPU calls are worse than one batched call.
- **A quota**, larger for paying accounts, and shared across the platform at busy times.

It is the right choice for image generation, speech and mid-sized chat demos that are used in bursts. It is the wrong choice for a Space serving steady production traffic, where a dedicated endpoint is both faster and more predictable.

Choosing, in one pass

- **Embeddings, classifiers, ONNX, small Whisper** → free CPU. Permanently.
- **Image generation, larger chat, a public demo with a GPU** → ZeroGPU.
- **Steady traffic with a latency promise** → paid hardware with a sleep timer, or a dedicated endpoint.
- **A private tool for yourself** → free CPU and tolerate the cold start, or run it on your own machine.

Three habits that keep the bill honest

1. **Check your Spaces list monthly.** A forgotten upgraded Space is the classic way to spend money on nothing.
2. **Set a spending limit** in billing settings if your account supports one, rather than relying on memory.

3. **Develop locally, deploy once.** Every debugging cycle on paid hardware is a cycle billed for a bug.

The honest position on free

Hugging Face gives away a lot: unlimited public repositories, permanent free CPU Spaces, ZeroGPU quota, and a large amount of bandwidth. That generosity is real and it is also a commercial strategy, funded by enterprise customers and paid compute. It could change. Nothing in this lesson is a promise about 2028, and any project whose viability depends on a free tier staying free should be able to move — which, since everything here is git repositories and open formats, it can.

Do this now

Open the settings of any Space you own and look at the sleep time. If it says never and the hardware is not free, change one of the two right now, before finishing this lesson.

BEFORE YOU MOVE ON

A Space upgraded to a GPU serves about twenty requests a day. The owner is surprised by the size of the monthly bill. What is the mechanism?

A. GPU Spaces bill a minimum per-request allocation, so low-volume traffic is charged at a much worse effective rate.

B. Billing counts wall-clock uptime, and paid hardware keeps running between requests unless a sleep timer stops it.

C. Each cold start re-downloads the model, and bandwidth on paid hardware is charged separately from compute.

D. The Space kept a GPU reserved for the queue, which scales its reservation with the number of concurrent sessions.

34 · 8 min

The filesystem disappears, and what to do about it

THE ONE THING TO KEEP

A Space's local filesystem does not survive restarts, so state belongs in `gr.State` for a session, a paid `/data` volume for the model cache, or a private Hub dataset repository for anything collected — and collecting user text at all is a commitment that has to be stated on the page.

Assume the disk is temporary

A Space runs in a container. Rebuild it, restart it, let it sleep and wake, and anything written to the local filesystem may be gone. People discover this after a week of collected feedback vanishes on a routine restart.

There are four places to put state, and choosing correctly is most of this lesson.

1. Nowhere — the default, and often right

If your Space transforms input into output and keeps nothing, say so in the readme and move on. "Nothing you type is stored" is a feature, it is free, and it removes a whole category of obligation. Most demos should be here.

2. `gr.State` — per session, in memory

Conversation history, a wizard's current step, an uploaded file being worked on. Dies when the browser session ends, never shared between users. Covered in the Blocks lesson, and it is the right home for most of what people wrongly put on disk.

Four places to put state, and what survives a restart

Nowhere	Nothing is kept, and the readme says so. Right for most demos.
gr.State	One browser session, in memory. Never shared between visitors.
A paid /data volume	Survives restarts. Best spent on the model cache, not used as a database.
A private dataset repository	Free, versioned, durable. CommitScheduler batches the writes.

The container's own filesystem is not on the list, because it does not survive a restart and people discover that after a week of collected feedback disappears. Most demos belong on the first line, and saying so on the page is free.

3. Persistent storage — a real disk, paid

Spaces can be given a persistent volume mounted at `/data`, in tiers starting around 20 GB for a few dollars a month. It survives restarts and rebuilds.

Use it for the model cache, which is the highest-value thing you can put there:

```
---
# in README.md front matter
sdk: gradio
---
```

```
import os
os.environ["HF_HOME"] = "/data/.huggingface"   # before import-
ing transformers
```

Now a cold start after sleep loads weights from the volume instead of downloading them again, which can turn a three-minute wake into fifteen seconds. On a Space that sleeps often, this is the best use of a few dollars a month available anywhere on the platform.

It is still not a database: one volume, one Space, no backups by default, and no concurrency control beyond what you write yourself.

4. A Hub dataset repository — free, durable, versioned

For collecting anything — feedback, submitted examples, evaluation results — a dataset repository is better than a disk. It is free, it is versioned, it is backed up by being a git repository, and it can be private.

```
from huggingface_hub import CommitScheduler
import json, uuid, pathlib

folder = pathlib.Path("feedback"); folder.mkdir(exist_ok=True)
file = folder / f"{uuid.uuid4()}.jsonl"

scheduler = CommitScheduler(
    repo_id="your-name/space-feedback",
    repo_type="dataset",
    folder_path=folder,
    every=5,                # minutes
    private=True,
)

def record(text, verdict):
    with scheduler.lock:
        with file.open("a") as f:
            f.write(json.dumps({"text": text, "verdict": ver-
dict}) + "\n")
```

`CommitScheduler` batches writes and pushes them on a timer, so you are not making a git commit per request. The lock matters: without it, concurrent requests interleave partial lines and you get corrupt JSON.

The part that is not technical

If you store what users type, you have taken on responsibility for it. Before writing that code:

- **Say so on the page**, in plain words, above the fold. Not in a link.
- **Make the repository private** unless you have a specific reason and consent for it to be public. A public dataset of things strangers typed into a demo is a privacy incident waiting for a search engine.

- **Collect the minimum.** The text and a verdict, not IP addresses, not timestamps precise enough to re-identify, not the user agent.
- **Say how long you keep it**, and then actually delete it.
- **Do not collect at all** if the Space handles anything sensitive — health, finance, legal, anything about a named person.

Rules on personal data differ by country and are genuinely unsettled for AI training in several of them. This lesson is not legal advice and cannot be; the defensible engineering position is to collect as little as possible and to tell people exactly what you collect.

Secrets

Space settings have a Secrets section. Values set there arrive as environment variables and are not in the repository:

```
token = os.environ["HF_TOKEN"]
```

Never commit a token, never put one in a Gradio textbox default, and never print one into a log. Duplicating a Space does **not** copy its secrets — the duplicate prompts for its own, which is correct behaviour and occasionally confusing.

Do this now

Pick one Space you have made and decide, explicitly, which of the four storage answers it uses. Then write one sentence in the readme saying so. Most Spaces should say "nothing is stored", and most do not say anything at all.

BEFORE YOU MOVE ON

A Space writes user feedback to `feedback.jsonl` in its working directory and loses everything after a rebuild. Which fix addresses the cause rather than the symptom?

A. Write the file to `/tmp`, which is outside the application directory and therefore untouched by rebuilds.
B. Commit the file to the Space repository after each write, so it is versioned with the code.
C. Push the collected records to a private dataset repository on a scheduled commit.
D. Hold the records in a module-level list and write them out when the Space shuts down.

35 · 8 min

Streamlit, Docker and a Space with no server at all

THE ONE THING TO KEEP

Gradio, Streamlit, Docker and static are four different deployment shapes, and the static Space running transformers.js is the one people overlook — no server, no sleep, no cost, and the user's data never leaves their device.

Four SDKs, four different jobs

Gradio is the default and the best-supported. Its components map onto machine-learning inputs and outputs, it has ZeroGPU support, it exposes an API automatically, and most examples you find are written in it.

Streamlit re-runs your whole script top to bottom on every interaction, with `st.session_state` holding what must survive. That model is excellent for dashboards and data exploration and awkward for a chat interface, because a

long-running generation restarts the script's reasoning about state. Choose it when the Space is mostly charts and tables over a dataframe.

Docker is the escape hatch. You supply a `Dockerfile`, the Hub builds and runs it, and anything that listens on port 7860 works — FastAPI, Flask, Node, a Rust binary, a multi-process stack.

Static serves files. No server process, no sleep, no cold start.

The static Space is underrated

With transformers.js from module three, a static Space is a complete AI application that costs nothing and never sleeps:

```
---
title: Local Sentiment
sdk: static
app_file: index.html
---
```

`index.html` loads the library from a CDN, downloads an ONNX model into the browser, and runs everything on the visitor's device. No queue, no container, no per-request compute, and the user's text never leaves their machine. For classifiers, embeddings, background removal, OCR and short-clip speech recognition, this is frequently the best engineering answer as well as the cheapest.

Its limits are exactly the browser's: the model must be small enough to download once, and heavy generation is impractical.

Docker Spaces, and the one gotcha

```
FROM python:3.11-slim

RUN useradd -m -u 1000 user
USER user
ENV PATH="/home/user/.local/bin:$PATH" \
    HF_HOME=/home/user/.cache/huggingface

WORKDIR /app
COPY --chown=user requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY --chown=user . .

EXPOSE 7860
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port",
"7860"]
```

Three things this gets right, and each of them is a build that fails otherwise:

1. **Port 7860.** The platform routes to it. Listening on 8000 produces a Space that builds, starts and shows a blank page forever.
2. **`--host 0.0.0.0`.** Binding to localhost means nothing outside the container can reach it. Same blank page.
3. **A non-root user with a writable home.** Containers run unprivileged, and libraries that try to write a cache into a root-owned directory raise permission errors at import — usually the first confusing failure people meet here.

Reach for Docker when you need a database alongside the app, a system library that pip cannot install, a non-Python runtime, or a custom web front end. Do not reach for it to avoid learning Gradio; you will rebuild a queue, a file upload widget and a streaming response by hand.

Spaces are also a way to read code

Every public Space is a public repository. When you find one that does what you want, click Files and read it. That is a working configuration — the exact

dependency versions, the model revision, the hardware, the prompt — which is worth more than documentation because it is running.

This is the fastest way to learn a new technique on the Hub, and it is also why the licence of a Space's code matters: reading it is always fine, copying it is governed by whatever licence its readme declares, and many declare none. Ask, or write your own from what you understood.

The build is a cache, and caches go stale

All four SDKs share one build system, and it caches layers. Change `requirements.txt` and the dependency layer rebuilds; change only `app.py` and it does not. That is why a Space usually restarts in seconds and occasionally takes ten minutes.

It is also why a Space can fail in a way the current code cannot explain: a layer built against a package version that has since been yanked, or a half-downloaded model in the image cache. The factory reboot in the debugging lesson discards all of it and starts clean.

Choosing in one line

- Model demo with inputs and outputs → **Gradio**.
- Dashboard over a dataset → **Streamlit**.
- Small model, privacy, zero cost, no sleep → **static** plus transformers.js.
- Anything else → **Docker**.

Do this now

Find a public Space that does something close to what you want and read its `app.py` and `requirements.txt` end to end. Note the pinned versions and the hardware in the readme. You have just been handed a working configuration for free, which is the single most useful thing the Spaces ecosystem provides.

BEFORE YOU MOVE ON

A Docker Space builds successfully, reports itself as running, and shows a blank page. What should be checked first?

A. Whether the container binds to 0.0.0.0 on port 7860, since the platform routes only to that address and port.
B. Whether the SDK field in the readme is set to docker, because a mismatch makes the platform serve static files instead.
C. Whether the image runs as a non-root user, since privileged containers are rejected by the runtime after start.
D. Whether the build exceeded the free tier's image size limit, which truncates the final layer and serves an empty response.

36 · 8 min

Every Space is an API you can call

THE ONE THING TO KEEP

Gradio exposes every event handler as an API endpoint automatically, so any public Space is a callable service — but it can sleep, change or vanish and you cannot see its logs, so anything you depend on should be a private duplicate you control.

The feature almost nobody uses

Scroll to the bottom of any Gradio Space and there is a link reading **Use via API**. Click it. You get the endpoint names, the parameter types, and working code in Python and JavaScript.

This is not an extra somebody configured. Gradio exposes every event handler as an endpoint automatically. Which means: any public Space is a hosted service you can call from your own code, without deploying anything, without a key beyond a Hub token, and without the model being on your machine.

```
pip install gradio_client
```

```
from gradio_client import Client

client = Client("abidlabs/whisper")
result = client.predict("audio.wav", api_name="/predict")
print(result)
```

Files work too, in both directions:

```
from gradio_client import Client, handle_file

client = Client("your-name/your-space")
out = client.predict(
    image=handle_file("https://example.com/photo.jpg"),
    threshold=0.5,
    api_name="/segment",
)
```

`handle_file` takes a local path or a URL and does the upload. The result is a path to a downloaded file in a temporary directory.

Long jobs, without blocking

```
job = client.submit("a long prompt", api_name="/generate")
while not job.done():
    print(job.status())
print(job.result())
```

`submit` returns immediately and the job reports queue position and progress. This is what you want inside a web application, where blocking for ninety seconds is not an option.

The honest limits

Calling somebody else's Space is convenient and it is not a service level agreement:

- **It can be deleted or changed** at any moment, by someone who owes you nothing.
- **It can be asleep**, so your first call waits out a cold start.
- **It is queued**, behind everyone else on the internet using it.
- **Your input goes to their container**, and you do not know what they log. Never send personal or confidential data to a Space you do not own. This is the same judgement you would apply to any third-party API, and the informality of the Hub makes people forget it.

The responsible pattern is to **duplicate the Space first**, privately, and call your copy. You then control the hardware, the sleep policy and the data, at the cost of the hardware bill. For a Space you rely on, this is not optional.

Naming your own endpoints

If you want your Space to be callable, name the endpoints:

```
btn.click(fn=transcribe, inputs=[audio], outputs=[text], api_-
name="transcribe")
other.click(fn=internal_helper, inputs=[x], outputs=[y],
api_name=False)
```

`api_name="transcribe"` gives a stable `/transcribe` route that survives layout changes. `api_name=False` hides a handler from the API entirely, which you want for anything that is a UI detail rather than a capability.

A Space with named endpoints and an honest readme is a small, free, public service. That is a genuinely useful thing to publish, and it costs one keyword argument.

Private Spaces need a token

A private Space, or one on gated hardware, needs authentication on every call:

```
client = Client("your-name/private-space",
hf_token=os.environ["HF_TOKEN"])
```

Read the token from the environment rather than writing it in the script, for the reasons the tokens lesson sets out. A fine-grained token scoped to that one repository is better than an account-wide one, because a token that leaks from a client is a token that can be revoked without breaking everything else you own.

The MCP option

Gradio can expose a Space's functions as an MCP server, so an AI assistant that speaks the protocol can call your Space as a tool:

```
demo.launch(mcp_server=True)
```

The function signature and docstring become the tool description, which means type hints and a clear docstring stop being style and start being interface. Whether you use this or not, it is worth knowing that the same Space can be a web page, a REST endpoint and a tool for an agent with no additional code.

When not to do any of this

If you are calling a Space in a loop over ten thousand items, stop. You are using somebody's free demo hardware as a batch processor, which is rude, slow, and will be rate limited. Run the model yourself, duplicate the Space onto hardware you pay for, or use an inference provider. Spaces are for demonstration and light use; the platform's terms say as much.

Do this now

Pick any public Space that does something you need, open Use via API, and call it from a Python script. Then decide honestly whether you should be calling theirs or duplicating it — and if the answer is duplicate, do that instead.

BEFORE YOU MOVE ON

A startup builds a product feature on `gradio_client` calls to a stranger's public Space. What is the most serious problem with this, beyond reliability?

A. Gradio endpoints are not versioned, so a layout change in the Space renames the route and breaks the integration.

B. Free Space hardware rate limits by IP, so production traffic will be throttled after a few hundred calls.

C. Customer data is being sent to a container owned by someone with no obligations and unknown logging.

D. Calls through gradio_client bypass the queue, so the Space's owner sees degraded service for their own users.

---

37 · 8 min

It built, it started, it is red

THE ONE THING TO KEEP

Build logs and container logs describe different phases and most confusion comes from reading the wrong one, while a container killed with no traceback is the out-of-memory signature — and a factory reboot rather than a restart is the fix when the code cannot explain the failure.

Three logs, and they say different things

On a Space page, the top right has status and logs. There are two streams and people read the wrong one.

Build logs cover cloning, installing dependencies and constructing the image. Failures here are pip resolution conflicts, missing system packages, a Python version mismatch, or an image that got too large. The Space never starts.

Container logs are your application's stdout and stderr after it starts. Failures here are import errors, missing environment variables, out-of-memory kills and exceptions in your handlers. The Space builds and then dies or misbehaves.

A third source is the browser console, for when the app runs fine and the page misbehaves — usually a component version mismatch after an unpinned SDK upgrade.

Two log streams, and what each can tell you

	Which phase	Typical failure
Build logs	**Build** Cloning and install	**Pip conflict** Version resolution
Container logs	**Runtime** Your app's own output	**OOM kill** No Python traceback

Most confusion here is reading the wrong stream. A build failure means the Space never started; a container failure means it started and then died. The bottom right cell is the one worth memorising: no Python traceback at all is the out-of-memory signature, because Python never got to handle anything.

The five failures that cover most of it

1. Build fails on dependency resolution. Two packages want incompatible versions of a third. Pin explicitly, and pin `torch` first because everything else resolves around it. Reproduce locally in a clean virtual environment before pushing; the build loop on the Hub is minutes long and your local loop is seconds.

2. `ModuleNotFoundError` at runtime. The package is imported and not in `requirements.txt`. It worked locally because your environment has it from something else. A clean local virtual environment catches this every time.

3. Killed with no traceback. This is the out-of-memory signature. The container exceeded its RAM and the kernel terminated it. There is no Python exception because Python did not get to handle anything. On free CPU hardware with 16 GB, loading a 7B model at float32 does this reliably. Fix by using a smaller model, a lower dtype, quantization, or bigger hardware — in that order of preference.

4. Runs, but every request is slow. Model loading inside the handler. Covered earlier, still the most common performance bug.

5. Worked yesterday, broken today, no commits. Something unpinned moved: `sdk_version`, a dependency, or the model repository's `main` branch. This is why the configuration lesson insists on pinning all three.

Factory reboot versus restart

Restart stops and starts the container with the existing build cache. Fast, and it fixes a hung process.

Factory reboot discards the cache and rebuilds from scratch. Slow, and it is what you need when a stale layer is the problem — a dependency that installed badly, a model file that downloaded corrupt. If a Space is failing in a way that makes no sense against the code you can read, factory reboot before you spend an hour on it.

Making failures legible to yourself

Print at startup. It costs nothing and it answers the first three questions you will have:

```
import sys, torch, transformers, os
print("python", sys.version)
print("torch", torch.__version__, "cuda", torch.cuda.is_avail-
able())
print("transformers", transformers.__version__)
print("HF_TOKEN set:", bool(os.environ.get("HF_TOKEN")))
print("model revision:", REVISION)
```

And wrap handlers so users see a sentence rather than a stack trace:

```
def safe(fn):
    def inner(*a, **k):
        try:
            return fn(*a, **k)
        except Exception as e:
            print("handler error:", repr(e), flush=True)   # to
container logs
            raise gr.Error("Something went wrong. Try a shorter
input.")
    return inner
```

`gr.Error` shows a clean message in the interface. `flush=True` matters: buffered stdout in a container that then dies means the log line you needed never arrived.

Before you ask for help

Spaces have a Community tab, and so do the models they use. A good report contains: what you expected, what happened, the last thirty lines of the relevant log, your `requirements.txt`, and the `sdk_version`. Most unanswered questions on the Hub are missing the log or the versions, and both are two clicks away.

Shipping: the last ten minutes

Before you tell anybody about it:

1. Open it in a private window — you will find the thing that only worked because you were logged in.
2. Open it on a phone. Gradio is responsive; your custom layout may not be.
3. Click every example.
4. Submit empty input, absurdly long input, and input in a language the model does not handle.
5. Read your own readme as a stranger. Does it say what this does, what it does badly, and whether anything is stored?

Do this now

Open the container logs of a Space you have built and read them from the top. You will find at least one warning you have never seen — almost always about a default you did not know you had accepted.

BEFORE YOU MOVE ON

A Space dies a few seconds after starting, with no Python traceback in the container logs. What does that pattern indicate?

A. A dependency failed to install, so the application never reached its first import.

B. The process was killed for exceeding memory, so Python never got the chance to raise anything.

C. An exception was raised inside a handler, and Gradio swallowed it to keep the interface responsive.

D. The sleep timer triggered immediately because the Space received no traffic after starting.

CASE STUDY · MODULE 4

The draft that appeared in somebody else's browser

A transparency NGO in Ranchi running a Gradio Space that helps people draft Right to Information applications, in a week when four hundred people a day were using it.

The Space took a person's complaint, typed in Hindi, and produced a formatted application naming the right public authority and the right section. Farhan, a volunteer, had written it in about a hundred and fifty lines. It had run for four months on free hardware at roughly ten visitors a day and had never once misbehaved.

The tool existed because filing a Right to Information application is not hard and is easy to get wrong. Name the wrong public authority and the application is transferred, or lost, and thirty days become ninety. The Space did one useful thing: it read a complaint in ordinary words and produced a correctly addressed application with the right section quoted.

Then a radio programme mentioned it, and traffic went to four hundred a day.

On the third day an email arrived from a user in Gumla. He had clicked the draft button and received an application about somebody else's ration card, with a name and a village in it.

The cause was one line near the top of the file. A list, declared at module level, holding the conversation so far. A Space is one process serving everybody, so a module-level variable is shared by every visitor on the internet. At ten visitors a day, spread across a working day, two people were almost never in the handler at the same time. At four hundred a day they were.

The decision

It was nine in the evening. Farhan was confident the fix took twenty minutes: move the per-person state into a session object and leave only the model shared.

Anjali Toppo, who ran the NGO, asked two questions before agreeing to patch and stay up. How many people had seen somebody else's text? And would they know if they had?

Nobody could answer either. The Space kept no logs and no analytics, because the readme said nothing you type is stored and that had been a deliberate choice.

That absence was the finding, and it cut both ways. It made the incident impossible to investigate. It was also the reason the exposure stopped at whoever happened to be on screen at that moment, rather than sitting in a file on a container that anybody who could reach the machine could read.

The second decision was harder than the first. Saying nothing meant the radio push kept working. Saying something meant announcing, in the same week the NGO was asking people to trust it with a grievance about their own ration card, that one person's grievance had been shown to a stranger.

Anjali's decision about disclosure turned on what the NGO was asking people for. It asks people to write down a grievance about their own household — a pension that has not arrived, a ration card that was cancelled, a road that was paid for and not built. The whole product is a request to trust it with something that is nobody else's business. An organisation making that request cannot decide, privately, that one breach of it was small enough not to mention.

There was also a version of the fix that Farhan wanted and did not do. He could have kept the Space up and pushed the patch straight to the running container, which would have taken the same twenty minutes and lost no traffic. Anjali refused it on the grounds that a patch applied to a live service, at nine at night, by a tired volunteer, with no way to tell afterwards whether it had worked, is how a small incident becomes a larger one. Taking it down made the test possible: two browsers, two sessions, and a check that they stayed apart before anybody else could reach it.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They took it down that night and replaced it with a page saying the tool was paused because a fault could show one person's draft to another, that nothing had been stored, and that it would be back the next day.

Farhan moved every per-visitor value into `gr.State`, which is per session, and left the model where it was, because a model is read-only and shared is the correct thing for it to be. He opened the Space in two browsers and checked the two sessions stayed apart before pushing. He turned flagging off, so nothing anybody typed could be written to disk by accident, and added one line above the fold in the readme rather than behind a link.

It was back in nineteen hours. Use in the following week was higher than the week before, and two people wrote to say the notice was the reason they tried it. The rule Farhan left as a comment at the top of the file is four words long: the model is shared, the person is not.

The bug survived four months of use. Why did neither testing nor the first four months find it?

Because a module-level variable behaves perfectly when there is one visitor at a time, which is the condition of every test the author runs and, at ten visitors a day, of nearly every real request too. The defect only appears when two people are inside the handler at once. That is why the rule is structural rather than empirical: `gr.State` is per session, a module-level object is per process, and correctness under one user tells you nothing.

The lack of logging prevented the investigation and also limited the harm. How should a Space like this one resolve that tension?

By separating what is needed to operate from what is a record of people. A counter of requests, or a log line with no user text in it, would have answered how many sessions overlapped without storing a single grievance. The commitment in the

readme was about user text, and it could have been kept exactly while still recording enough to investigate. Collect the minimum, say what you collect, and make sure the minimum includes what you would need after an incident.

The model stayed a module-level object while the history moved. Why is that not inconsistent?

Sharing is correct for anything read-only and expensive, and wrong for anything about a person. The model is loaded once at import and never mutated, which is exactly what makes a Space fast — loading it inside the handler would reload the weights on every request. The conversation is mutated by whoever is speaking, so it has to be per session. The distinction is not module level versus not; it is read-only and shared versus written and personal.

MODULE 4 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A Space answers in 200 milliseconds when you test it and takes 30 seconds for visitors. Which defect fits?

A. Free CPU hardware throttles a container after its first few requests.
B. The model is loaded inside the handler, so the weights reload per request.
C. The queue is disabled, so requests run in parallel and compete for the same memory.
D. Gradio rebuilds the whole interface each time somebody presses submit.

2 Two visitors to one Gradio Space start seeing each other's conversation. What is the cause, and the fix?

A. The queue is off; enabling demo.queue() serialises requests and separates them.
B. Browser caching; sending a no-store header on the response prevents it.
C. A module-level variable is shared by the whole process; use gr.State instead.
D. The model object is shared, so loading it inside the handler isolates users.

3 You upgrade a Space to a rented GPU to test something, then close the tab. What happens to the bill?

A. It stops, because an idle Space always sleeps automatically after about 48 hours.
B. Nothing accrues, because paid hardware is billed per request served.
C. It pauses whenever no browser is connected to the Space.
D. It continues by wall-clock time until you set a sleep timer or downgrade.

4 A duplicated Space that worked in March fails to start in October with a TypeError about an unexpected keyword argument, and nothing was committed. What do you check first?

A. Whether the model repository it loads has been deleted or gated.

B. Whether sdk_version is pinned in the README front matter.

C. Whether the free hardware tier has reduced its memory limit.

D. Whether a factory reboot is needed to discard a stale build layer.

5 A Space builds, starts, and then the container dies with no Python traceback anywhere in the logs. What does that pattern mean?

A. A syntax error, which is reported before the interpreter starts running.

B. A missing dependency, which fails quietly at import time.

C. The process was killed for exceeding memory, so Python handled nothing.

D. A stale build layer, which must be discarded with a factory reboot first.

6 You need a public Space's function inside a nightly job over ten thousand rows. What is the responsible route?

A. Call the public Space in a loop, since Gradio queues every request anyway.

B. Ask the author to raise the rate limit on their Space for your account.

C. Duplicate it privately onto hardware you control, and call your own copy.

D. Call it with a Hub token, which removes queueing for authenticated callers.

MODULE 5

Datasets, and the work that happens before any model

Most of the real job is data, and the datasets library exists so that a 500 GB corpus behaves like a Python list on a laptop. This block covers loading and streaming, the Arrow format that makes it possible, transforming at scale, querying the Hub's parquet copies with SQL before downloading a byte, and then the unglamorous checks — duplicates, leakage, class balance, label noise — that decide whether anything trained on it means anything.

> BY THE END OF THIS MODULE YOU CAN
>
> Inspect, stream, filter and transform a dataset far larger than your RAM, query the Hub's parquet conversion with SQL before downloading anything, run duplicate, leakage and class-balance checks on your own data, and publish it with a card stating provenance and licence

38 · 9 min

Do not download the dataset

THE ONE THING TO KEEP

Inspect a dataset in the viewer and stream it with `streaming=True` rather than downloading, and treat licence obligations as surviving fine-tuning until a lawyer in your country tells you otherwise.

The reflex that costs you an afternoon

Someone finds a dataset that looks right, calls `load_dataset`, and watches a progress bar for two hours to discover that the column they needed is empty for the language they care about. Everything in this lesson exists to stop that.

Look before you fetch

Every public dataset on the Hub gets an automatic viewer: a paged table with the real rows, the column names and the inferred types, plus per-column statistics. Public datasets are also converted automatically to Parquet, a columnar format that can be read a column and a chunk at a time over the network.

Those two facts together mean you can answer most of your questions in a browser, on a phone, before downloading a byte. How many rows. What the label distribution looks like. Whether the text is actually the transcript or a URL pointing at one. Whether your language is present in any quantity or is three hundred rows out of eleven million.

When the viewer is not enough, query the Parquet directly. DuckDB reads Parquet over HTTP and fetches only the ranges it needs:

```
INSTALL httpfs;
LOAD httpfs;

SELECT language, count(*)
FROM 'hf://datasets/<owner>/<name>/**/*.parquet'
GROUP BY language
ORDER BY 2 DESC;
```

Support for the `hf://` prefix depends on your DuckDB version; if it complains, the plain `https://huggingface.co/...` URL of a single Parquet file works too. A grouped count over a large corpus pulls a few megabytes of one column, not the corpus. If SQL is unfamiliar, /learn/data-and-sql is the shorter road than learning it from a dataset this size.

Loading, and streaming

When you do want rows in Python:

```
pip install -U datasets
```

```
from datasets import load_dataset

ds = load_dataset("stanfordnlp/imdb", split="train")
print(ds[0]["text"][:200])
```

That downloads and caches everything, which is right for 80 MB and wrong for 800 GB. For anything large, add one argument:

```
ds = load_dataset("HuggingFaceFW/fineweb", split="train",
streaming=True)

for row in ds.take(5):
    print(row["text"][:200])
```

`streaming=True` returns an iterable dataset instead of a materialised one. It fetches Parquet row groups over HTTP as you iterate and never writes a com-

plete copy to disk. You lose random indexing and `len()`. You keep `.take()`, `.skip()`, `.filter()`, `.map()` and `.shuffle()` with a buffer. This is the difference between exploring a web-scale corpus on a laptop with 30 GB free and not exploring it.

One compatibility note: recent versions of `datasets` removed the old mechanism where a dataset repository shipped a Python loading script. Tutorials that pass `trust_remote_code=True` to `load_dataset` are describing the old behaviour and will fail. Almost all significant datasets now ship as plain data files, which is both safer and faster.

The licences that survive fine-tuning

Here is where people get badly caught, and where the honest answer is uncomfortable.

A dataset carries a licence just as a model does, and the obligations do not obviously evaporate when the data passes through gradient descent. Three that recur:

- **Non-commercial** data does not become commercial because you trained on it. Whatever your view of the underlying law, no dataset card has ever said *unless you fine-tune, in which case never mind*.
- **Share-alike** terms are designed to propagate to derivatives, and whether a set of weights is a derivative of its training data is exactly the question nobody has settled.
- **Attribution** obligations under CC-BY do not disappear because you no longer ship the rows.

This area is genuinely unsettled and it differs by country. Courts in different jurisdictions are actively disagreeing about whether training is reproduction, whether it falls under text-and-data-mining exceptions, and what a model's weights are in legal terms. That is the shape of it; a lawyer in your country is the person who answers it for your case. What you can do without any lawyer is keep a record of which datasets went into which training run, because the question you cannot answer later is *what was in it*. /learn/fine-tuning covers the training side.

The other half of the card

Read the dataset card's sections on collection and on personal information. Was this scraped or contributed. Did the people in it know. Is there a take-down or opt-out route. Face datasets and voice datasets in particular have been assembled from material whose subjects never agreed, and several well-known ones have been withdrawn for that reason after other people had already built on them.

If you are assembling your own dataset from a community, the consent question is not a formality you satisfy afterwards. It is the design.

Do this now

Open the viewer of a dataset in your own field and check one thing you assumed: the number of rows in your language, or whether the label you need is populated. It takes a minute and saves the two hours.

BEFORE YOU MOVE ON

A researcher wants to count how many rows of a 400 GB text corpus are in Marathi, on a laptop with 40 GB free. What is the sound approach?

A. Download a random shard of the dataset and extrapolate the proportion from it, since a sample is representative of the whole.

B. Query the automatically converted Parquet files over HTTP with DuckDB, or iterate with `streaming=True`, so only the language column is transferred.

C. Call `load_dataset` with a filter on the language column, because filtering happens before anything is written to disk.

D. Ask for the count in the dataset's Community tab, since only the maintainers can compute statistics over a corpus that size.

39 · 9 min

A dataset that behaves like a list and is not one

THE ONE THING TO KEEP

A `Dataset` is a typed table where indexing a row returns a dict and slicing returns a dict of lists, and operations like shuffle and select build an indices map rather than copying data — which is why reordering a dataset far larger than RAM is instantaneous.

Loading, and what comes back

```
from datasets import load_dataset

ds = load_dataset("imdb")
print(ds)
```

```
DatasetDict({
    train: Dataset({features: ['text', 'label'], num_rows:
25000}),
    test:  Dataset({features: ['text', 'label'], num_rows:
25000}),
    unsupervised: Dataset({features: ['text', 'label'],
num_rows: 50000}),
})
```

A `DatasetDict` is a dictionary of splits. Each `Dataset` has features — named, typed columns — and a row count. Ask for one split directly and you get the `Dataset` alone:

```
train = load_dataset("imdb", split="train")
print(train[0])                    # one row, as a dict
print(train[:3]["text"][0][:80])   # a slice returns a dict of
lists
print(train.features)
```

That last line is worth internalising. **Indexing a row gives a dict; slicing gives a dict of lists.** `train[0]` is `{'text': ..., 'label': 1}`; `train[:3]` is `{'text': [...], 'label': [...]}`. Half the confusing errors in a first hour with this library come from expecting a list of dicts.

Features are typed, and the types do work

```
print(train.features["label"])
# ClassLabel(names=['neg', 'pos'], id=None)

print(train.features["label"].int2str(1))   # 'pos'
```

`ClassLabel` carries the names, so you never have to guess whether 1 means positive — the same problem `id2label` solves on the model side. Other types you will meet are `Value("string")`, `Value("int32")`, `Sequence`, `Image`, and `Audio`, and the last two decode lazily, which the media lesson returns to.

Split slicing without writing Python

```
small = load_dataset("imdb", split="train[:1%]")
middle = load_dataset("imdb", split="train[100:200]")
both  = load_dataset("imdb", split="train+test")
```

`train[:1%]` is the habit worth forming. Develop on one per cent, then remove the suffix. A pipeline debugged on 250 rows runs unchanged on 25,000, and you will have saved yourself ten minutes per mistake.

Splitting when the dataset has no split

```
parts = ds["train"].train_test_split(test_size=0.1, seed=42,
stratify_by_column="label")
```

Three things here are deliberate. `seed` makes it reproducible, so the numbers you quote next week come from the same split. `stratify_by_column` keeps class proportions equal across the two halves, which matters enormously when a class is rare — an unstratified 10% test split of a dataset with 2% positives can easily land with almost none, and your recall figure then measures nothing.

And a validation set is a third split, not a second use of the test set:

```
tmp = ds["train"].train_test_split(test_size=0.2, seed=42)
val_test = tmp["test"].train_test_split(test_size=0.5, seed=42)
# tmp['train'] = 80% train, val_test['train'] = 10% val,
val_test['test'] = 10% test
```

You tune on validation. You touch test once, at the end. Every time you look at the test score and change something, it becomes a little more like a training set, and the number it reports becomes a little more optimistic than reality.

Loading your own files

No upload required — `load_dataset` takes formats directly:

```
ds = load_dataset("csv", data_files="reviews.csv")
ds = load_dataset("json", data_files={"train": "tr.jsonl",
"test": "te.jsonl"})
ds = load_dataset("parquet", data_files="data/*.parquet")
ds = load_dataset("imagefolder", data_dir="photos/")   # labels
from folder names
```

So the whole library is available for local work. You do not need to publish anything to get streaming, mapping, caching and typed columns.

Configurations, when one name holds many datasets

Some repositories hold several variants, and the second positional argument chooses between them:

```
from datasets import get_dataset_config_names
print(get_dataset_config_names("mozilla-foundation/com-
mon_voice_17_0")[:5])

hi = load_dataset("mozilla-foundation/common_voice_17_0", "hi",
split="train")
```

For multilingual corpora the configuration is usually the language, and loading without it either fails with a list of options or silently gives you the default — which, more often than is comfortable, is English. Print the configuration names before you assume.

Sorting, shuffling, selecting

```
ds = ds.shuffle(seed=42)
small = ds.select(range(1000))
pos = ds.filter(lambda r: r["label"] == 1)
ds = ds.remove_columns(["unused"]).rename_column("text",
"sentence")
```

None of these copy the data. `select` and `shuffle` build an **indices map** — a list of row positions — and the underlying table stays where it is. This is why shuffling a 40 GB dataset is instant and takes no memory, and it is a genuine surprise the first time.

The exception is `filter`, which has to read every row to decide. On a large dataset, pass `num_proc` to parallelise it.

Do this now

Load any dataset with `split="train[:1%]"`, print `features`, print row zero, and print a slice of three. Then run `train_test_split` with a seed and check

that running it twice gives the same first row. That reproducibility is what makes every later comparison meaningful.

BEFORE YOU MOVE ON

A team splits a dataset with 2% positive examples into 90/10 train and test without stratifying, and their reported recall swings wildly between runs with different seeds. Why?

A. The test split holds very few positives, so recall is computed over a handful of rows.

B. Unstratified splitting leaves some positives in both splits, so the model has memorised part of the test set.

C. Recall is undefined for imbalanced data and should be replaced with accuracy, which is stable across seeds.

D. The shuffle indices map is rebuilt per seed, so row order changes the order of training updates and therefore the model.

40 · 8 min

Why a 300 GB dataset opens instantly

THE ONE THING TO KEEP

Datasets are memory-mapped columnar Arrow files, so size is limited by disk rather than RAM and row access faults in only the pages it needs — while `streaming=True` drops random access and true shuffling in exchange for needing no disk at all.

The thing that should not work

```
ds = load_dataset("wikipedia", "20220301.en", split="train")
print(len(ds))      # 6,458,670
print(ds[3_000_000]["title"])
```

That is roughly 20 GB of text, opened on a laptop with 8 GB of RAM, and the row access returns immediately. Nothing was loaded into memory. Understanding why changes how you work with data for good.

Arrow, and memory mapping

The library stores datasets as **Apache Arrow** files on disk: a columnar binary layout where every value's position is computable rather than searched for. It then **memory-maps** the file — asks the operating system to make the file's bytes appear at an address range, without reading them.

When you touch row three million, the OS faults in the few pages that hold it. Pages you never touch are never read. Pages you stop using are evicted by the OS under memory pressure, with no work from Python.

Three consequences, all practical:

1. **RAM is not the limit.** Disk is. A dataset larger than memory works normally.

2. **Several processes share it.** Four dataloader workers on one machine map the same file and the pages are shared, rather than each holding a copy.
3. **Startup is instant** regardless of size, because nothing is parsed at load time.

This is also why `Dataset` objects cannot simply be pickled and sent across a network — they are a handle on a file, not the data.

Memory-mapped or streamed: what you trade

Neither is limited by RAM. The rule for choosing is how many passes you will make: stream when you will read the data roughly once, download when you will read it several times, because re-streaming every epoch means re-downloading every epoch.

Columnar, which is why one column is cheap

Arrow stores all of `text` together and all of `label` together, rather than interleaving them row by row. Reading one column of a twelve-column dataset reads about a twelfth of the bytes. `ds.remove_columns([...])` before a heavy operation is therefore not tidiness, it is a real speed-up, and on wide datasets it is a large one.

Parquet, which the Hub converts every dataset into, is the same idea with compression added, and it is the format the SQL lesson queries directly.

The cache, and the file that reappears

Every transformation writes a new Arrow file to `~/.cache/huggingface/datasets`, keyed by a hash of the operation. This is why `map` is slow the first time and instant the second, and why your disk fills quietly.

```
print(ds.cache_files)
print(ds.dataset_size / 1e9, "GB")
```

Manage it deliberately:

```
ds = ds.map(fn, load_from_cache_file=False)     # force recom-
pute
ds.cleanup_cache_files()                        # delete this
dataset's caches
```

The hash covers the function's bytecode, so editing your function invalidates the cache correctly. It does **not** cover a global variable your function reads. Change a threshold that lives outside the function and you will get stale cached results with no warning — a genuinely nasty bug, and the reason to pass parameters as arguments rather than closing over globals.

Streaming: when even disk is too much

```
ds = load_dataset("HuggingFaceFW/fineweb", split="train",
streaming=True)
for row in ds.take(5):
    print(row["text"][:100])
```

With `streaming=True` you get an `IterableDataset`: rows are fetched over HTTP as you consume them and nothing lands on disk. This is how you work with a multi-terabyte corpus from a laptop or a free Colab session.

What you give up is random access. `ds[42]` does not exist, `len(ds)` usually does not, and shuffling becomes a buffer rather than a true permutation:

```
ds = ds.shuffle(seed=42, buffer_size=10_000)
```

That fills a 10,000-row buffer and samples from it. Good enough for training; not a real shuffle, which matters if the file order correlates with anything.

`map` and `filter` work on streams and are applied lazily as rows pass. The rule for choosing: **stream when you will read the data roughly once**, download when you will make several passes, because re-streaming for every epoch means re-downloading.

Interleaving sources

```
from datasets import interleave_datasets
mixed = interleave_datasets([hindi, english], probabilities=
[0.3, 0.7], seed=42)
```

This samples from several streams at stated proportions — the standard way to mix languages or domains for pretraining without materialising a combined file.

Do this now

Load a dataset of several gigabytes without streaming and watch your memory usage while you index random rows. It will not move. Then check `ds.cache_files` and note where those bytes actually live, because that directory is the one that fills your disk.

BEFORE YOU MOVE ON

A `map` function reads a threshold from a module-level variable. The threshold is changed and the pipeline re-run, but the output is unchanged. What happened?

A. Memory mapping served the previous transformation's pages from the OS cache rather than recomputing them.

B. Arrow files are immutable, so a second `map` over the same dataset is silently ignored until the cache is cleared.

C. The cache fingerprint hashes the function itself, which did not change, so the stale result was reused.

D. Streaming datasets apply transformations lazily, so the change takes effect only once the iterator is exhausted and restarted.

41 · 9 min

Transforming a million rows without waiting an hour

THE ONE THING TO KEEP

`map(batched=True)` hands your function a dict of lists and can return a different number of rows than it received, which makes it both the fast path and the tool for chunking — while `set_transform` applies on access for anything random that should not be cached.

The naive version, and the fast one

```
# slow: one Python call per row
ds = ds.map(lambda r: {"n_chars": len(r["text"])})

# fast: one Python call per thousand rows
ds = ds.map(
    lambda b: {"n_chars": [len(t) for t in b["text"]]},
    batched=True, batch_size=1000,
)
```

In batched mode the function receives a dict of **lists** and must return a dict of lists of the same length. Ten to fifty times faster is typical, and the reason is not clever: the per-call overhead of crossing between Arrow and Python dominates, and batching pays it a thousand times less often.

For tokenization the gain is larger still, because the Rust tokenizer parallelises across a batch internally:

```
def tok_fn(batch):
    return tok(batch["text"], truncation=True, max_length=512)

ds = ds.map(tok_fn, batched=True, remove_columns=["text"])
```

`remove_columns` drops the original text once you have the ids. On a large corpus this is the difference between a 30 GB cached file and a 6 GB one.

Parallelism across cores

```
ds = ds.map(fn, batched=True, num_proc=8)
```

This forks processes, each mapping a shard, and then concatenates. Two cautions that cost people real time:

- **Fast tokenizers already use threads.** Combining `num_proc` with them can oversubscribe the CPU and run slower. The library will warn about `TOKENIZERS_PARALLELISM`; set it to `false` when using `num_proc`, or use one or the other.
- **Anything holding a GPU does not fork well.** For GPU work, use one process and a large batch, not eight processes fighting over one card.

A batch does not have to stay the same size

This is the feature people miss. A batched map may return more or fewer rows than it received, which makes it the tool for splitting and dropping:

```
def chunk(batch, size=800, overlap=100):
    out = {"chunk": [], "doc_id": []}
    for doc_id, text in zip(batch["id"], batch["text"]):
        for i in range(0, len(text), size - overlap):
            out["chunk"].append(text[i:i + size])
            out["doc_id"].append(doc_id)
    return out

chunks = docs.map(chunk, batched=True,
remove_columns=docs.column_names)
```

One thousand documents in, forty thousand chunks out, with the source id carried along. `remove_columns=docs.column_names` is required here, because the old columns no longer have the right length and the library will refuse the mismatch.

`filter`, and doing it in one pass

```
ds = ds.filter(lambda b: [len(t) > 50 for t in b["text"]],
batched=True, num_proc=4)
```

Filtering reads everything, so chain your conditions into one predicate rather than calling `filter` four times. Four filters are four full passes.

`with_transform`: when you do not want a new file

`map` materialises a new Arrow file. When the transformation is cheap and varies per epoch — image augmentation, random cropping, dynamic masking — that is wasteful and wrong, because you want different randomness each time.

```
def augment(batch):
    batch["pixel_values"] = [aug(img) for img in
batch["image"]]
    return batch

ds.set_transform(augment)      # applied on access, nothing
written
```

Rule: `map` for deterministic preprocessing you want cached, `set_transform` for anything random or cheap.

Making it reproducible

Every map that involves randomness needs a seed passed in, not read from global state, or your cache and your results diverge from each other. And keep the transformation code in a module rather than a notebook cell, because a notebook cell that was edited and re-run is not a record of what produced your data.

Sorting, grouping and joining

`sort`, `class_encode_column` and `flatten_indices` cover most reshaping. There is no join; for that, convert to pandas or Polars, join, and convert back:

```
import pandas as pd
from datasets import Dataset
merged = Dataset.from_pandas(ds.to_pandas().merge(other_df,
on="id"))
```

That materialises everything in RAM, so it is fine for a few million rows and not for a corpus. For genuinely large joins, DuckDB over the parquet files is the next lesson and the right tool.

Do this now

Take one `map` you have written per row and rewrite it batched. Time both on 50,000 rows. The number you get is the tax you have been paying on every dataset you have ever processed one row at a time.

BEFORE YOU MOVE ON

A chunking function returns 40,000 chunks from 1,000 documents inside `map(batched=True)` and raises an error about mismatched lengths. What is missing?

A. `num_proc` must be set so the output shards are concatenated rather than aligned against the input.

B. The original columns must be removed; they still have the input's length.

C. The batch size must divide evenly into the output count, so that each batch produces a whole number of chunks.

D. The function must return a list of dicts rather than a dict of lists when the row count changes.

42 · 8 min

Query a dataset with SQL before downloading it

THE ONE THING TO KEEP

The Hub auto-converts public datasets to parquet, so DuckDB can run SQL against them over HTTP and answer class balance, length distribution, duplication and train–test overlap in seconds without downloading anything.

The conversion nobody mentions

When a dataset is published publicly on the Hub, the platform converts it to **Parquet** automatically and serves those files. The dataset viewer on the page is reading them. So is anything else that can read parquet over HTTP.

That gives you a capability most people never use: running real SQL against a dataset you have not downloaded, on a laptop, in seconds.

DuckDB, which is free and is one pip install

```
pip install duckdb
```

```
import duckdb

q = """
SELECT label, COUNT(*) AS n
FROM 'hf://datasets/imdb/plain_text/train-*.parquet'
GROUP BY label
"""
print(duckdb.sql(q).df())
```

DuckDB reads parquet metadata first, works out which row groups it needs, and fetches only those byte ranges. A count over a 50 GB dataset can transfer

a few megabytes, because column statistics live in the footer and a `GROUP BY` on one column never touches the others.

The file paths are discoverable from the API:

A count over a fifty-gigabyte corpus can transfer a few megabytes, because parquet keeps its column statistics in the file footer and a query on one column never touches the others. This is why class balance is a question you can answer on a phone before downloading anything.

```
import requests
r = requests.get("https://datasets-server.huggingface.co/par-
quet?dataset=imdb").json()
for f in r["parquet_files"][:3]:
    print(f["split"], f["url"], f["size"])
```

The five questions worth asking before you download

1. **How big is each class?** A dataset described as balanced frequently is not.

``` sql SELECT label, COUNT() *FROM 'hf://datasets/.../train-*.parquet' GROUP BY label ```

1. **How long are the texts?** This decides your `max_length` and therefore your compute bill.

``` sql SELECT MIN(len), MAX(len), MEDIAN(len), QUANTILE_CONT(len, 0.95) AS p95 FROM (SELECT LENGTH(text) AS len FROM '...') ```

1. **Are there duplicates?**

``` sql SELECT COUNT(*) - COUNT(DISTINCT text) AS dupes FROM '...' ```

1. **Does the test split overlap the train split?** The leakage check, and the one that invalidates published results:
```

```sql SELECT COUNT() FROM 'hf://datasets/.../test-.parquet' t JOIN 'hf://datasets/.../train-*.parquet' r ON t.text = r.text ```

1. **What does the rare class actually look like?** Pull twenty rows and read them. Ten minutes of reading beats an hour of aggregate statistics, every time.

The viewer itself does more than you think

On the dataset page: full-text search across rows, sorting by column, filtering, and a per-column statistics panel showing distributions, null counts and distinct values. For a first look this is faster than writing anything, and it costs no download.

It also honestly shows what many cards do not — that a "multilingual" corpus is 94% English, or that a column described as clean text is 3% null.

Turning a query into a view of the whole corpus

One more pattern is worth having. Aggregates answer what is in a dataset; a sample answers what it is like. Combine them:

```
SELECT label, text
FROM 'hf://datasets/.../train-*.parquet'
USING SAMPLE 20 ROWS
```

DuckDB's `USING SAMPLE` pulls a random subset without reading everything, so you can read twenty real rows of a terabyte corpus on a phone tethering connection. Reading twenty rows tells you about formatting, boilerplate, truncation and encoding problems that no aggregate will ever surface.

The limits

- **Public datasets only** by default. Gated and private ones need a token, and very large ones may be only partially converted, which the viewer states.
```

- **Some datasets have no viewer at all** — a custom loading script, an unsupported format, or an explicit opt-out. Absence is not a red flag by itself, but it does mean you are back to reading files by hand.
- **The parquet copy can lag** a very recent update by a short period.

Querying your own files the same way

DuckDB is not a Hub feature; it is a local database that happens to read remote parquet. Everything above works on a folder of your own files:

```
duckdb.sql("SELECT COUNT(*) FROM 'data/*.parquet' WHERE lang =
'hi'")
```

And it goes the other way, straight into the library:

```
from datasets import Dataset
df = duckdb.sql("SELECT text, label FROM '...' WHERE
LENGTH(text) BETWEEN 50 AND 2000").df()
ds = Dataset.from_pandas(df)
```

SQL for selection and aggregation, `datasets` for the training path. That combination handles almost everything without Spark, without a cluster, and on hardware you already own.

Do this now

Pick any public dataset you have considered using and run the five queries above against it without downloading it. Most people find at least one thing the card did not say — usually the class balance or the length distribution, and occasionally the train–test overlap.

BEFORE YOU MOVE ON

Why can a `GROUP BY` over a 50 GB Hub dataset transfer only a few megabytes?

A. The Hub pre-computes aggregate statistics for every column and DuckDB reads those instead of the rows.

B. Parquet is columnar, so only the needed column's byte ranges are fetched.

C. DuckDB samples a subset of row groups and extrapolates, which is accurate enough for grouped counts.

D. The query runs on the Hub's servers and only the result set crosses the network.

43 · 9 min

Turning your files into a dataset

THE ONE THING TO KEEP

Building a dataset is `from_dict`, `from_generator` or a folder convention plus explicitly typed `Features`, and publishing is one call — so the decisions that matter are the ones before the push: personal data, the right to redistribute, a licence, and starting private because public cannot be undone.

From whatever you have

```
from datasets import Dataset, DatasetDict

ds = Dataset.from_dict({"text": texts, "label": labels})
ds = Dataset.from_pandas(df)
ds = Dataset.from_list([{"text": "...", "label": 1}, ...])
```

For anything too large to hold in memory, use a generator so rows stream in:

```
def gen():
    for path in pathlib.Path("notes").glob("*.txt"):
        yield {"id": path.stem, "text":
path.read_text(encoding="utf-8")}

ds = Dataset.from_generator(gen)
```

For media, the folder conventions do the labelling:

```
ds = load_dataset("imagefolder", data_dir="photos")   #
photos/cat/*.jpg -> label 'cat'
ds = load_dataset("audiofolder", data_dir="clips")
```

A `metadata.csv` in the folder, with a `file_name` column, attaches captions, transcripts or extra fields to each file.

Set the types deliberately

```
from datasets import Features, Value, ClassLabel

features = Features({
    "text": Value("string"),
    "label": ClassLabel(names=["neg", "neu", "pos"]),
    "source": Value("string"),
})
ds = Dataset.from_dict(data, features=features)
```

`ClassLabel` stores the names with the data, so nobody downstream has to guess what 2 means. Without it your labels are bare integers and the meaning lives in a message somebody sent you in March. This is the same class of mistake as a model with `LABEL_0` outputs, and it is equally cheap to prevent.

Include a `source` or `provenance` column from the start. Six months later, every argument about a surprising result begins with where a row came from.

Publishing

```
ds.push_to_hub("your-name/support-tickets", private=True)
```

That creates the repository if needed, writes parquet shards, and commits. Loading it back is `load_dataset("your-name/support-tickets")`.

Start private. Making a private dataset public later is one click; un-publishing something that has been indexed, cloned and mirrored is not possible. The asymmetry should decide the default.

Push a `DatasetDict` and the splits are preserved:

```
DatasetDict({"train": tr, "validation": va, "test":
te}).push_to_hub(repo)
```

You can also push a configuration alongside others, which is how one repository holds several variants:

```
ds_hi.push_to_hub(repo, config_name="hindi")
ds_en.push_to_hub(repo, config_name="english")
```

Before you push anything

This list is short and every item on it has ruined somebody's week:

1. **Personal data.** Names, emails, phone numbers, addresses, account numbers, and the sentence where somebody names their employer. Run an automatic redaction pass, then read a sample by hand, because automatic redaction misses context. If the data is about people and you cannot redact it, the dataset stays private.
2. **Do you have the right to publish it?** Scraped content, customer data, anything under a contract, anything with a licence that forbids redistribution. Being able to download something is not permission to republish it.
3. **Credentials in the rows.** API keys pasted into support tickets are extremely common. Grep for them.

4. **A licence field.** Decide before publishing, not after. No licence is not the same as permissive.
5. **The intended use and the known gaps**, in the card.

The upload that silently changes your data

One behaviour surprises people. `push_to_hub` writes the dataset as it currently exists, including every transformation you applied in the session. If you tokenised and then pushed, you have published token ids rather than text, and nobody else can use it with a different model.

Publish the raw form. Keep preprocessing in a script in a repository beside it, so somebody with a different tokenizer can run your pipeline rather than inherit your choices. The rule generalises: **publish the thing, not your view of the thing.**

Keeping it updated

A dataset repository is git, so a new push is a commit and old versions remain reachable:

```
ds = load_dataset("your-name/support-tickets", revision="v1.0")
```

Tag releases when a number you published was computed against a specific version. A model card that says "trained on our dataset" without a revision is a claim nobody can check, including you.

Very large uploads

`push_to_hub` handles gigabytes. For hundreds of gigabytes, shard before uploading, use `upload_large_folder`, and expect to resume — a four-hour upload will be interrupted. The chunk-level storage behind the Hub deduplicates, so re-pushing a dataset where one shard changed does not resend everything.

Do this now

Take a folder of your own files — notes, tickets, transcripts, photos — and make it a private dataset with typed features and a source column. Then load it back with `load_dataset` and confirm the types survived. That round trip is the whole skill, and it takes ten minutes.

BEFORE YOU MOVE ON

Why is `ClassLabel` preferable to storing labels as plain integers in a dataset you intend to share?

A. It compresses the column, since integer categories are stored as dictionary-encoded values rather than full integers.

B. It enforces a closed set of values, so rows with an unexpected label are rejected at write time.

C. The class names travel with the data, so nobody downstream has to guess what 2 meant.

D. It enables stratified splitting, which cannot be performed on an untyped integer column.

44 · 8 min

The card is where a dataset stops being a file

THE ONE THING TO KEEP

A dataset card's job is provenance — where rows came from, under what terms, labelled by whom, with what agreement — because a licence tag cannot grant rights the compiler never held, and a dataset without recorded provenance can never be assessed against whatever the law turns out to be.

What a dataset card has to answer

A model card answers "what will this do". A dataset card answers something harder: "where did this come from, and what am I allowed and likely to get wrong with it". Four questions carry most of the weight.

Where did the rows come from? Scraped, licensed, generated by a model, written by annotators, donated by users. This single sentence determines legality, bias and whether a benchmark is contaminated. A card that will not say is telling you something.

What is the licence, and does it match the source? A dataset scraped from a site with restrictive terms does not become permissive because its uploader ticked Apache 2.0. The rights a dataset can grant are bounded by the rights its compiler held. Where a card's licence and its described origin disagree, the disagreement is the finding.

Who or what labelled it? Crowd workers paid per item, domain experts, automatic rules, or another model. Each produces a different error profile. Labels from a model carry that model's biases into anything you train, and if the labelling model is the one you later benchmark against, your evaluation is circular.

What is in it that you would not expect? Duplicates, a dominant language, a date range, a skewed source mix, offensive content. The best cards state this plainly; most do not state it at all.

The structured part

The front matter is YAML, and it is what the Hub indexes:

```
---
license: cc-by-4.0
language:
  - hi
  - en
task_categories:
  - text-classification
size_categories:
  - 10K<n<100K
annotations_creators:
  - expert-generated
source_datasets:
  - original
configs:
  - config_name: default
    data_files:
      - split: train
        path: data/train-*.parquet
---
```

`configs` is the functional part — it tells `load_dataset` which files belong to which split, which is why a well-configured repository loads with no arguments.

Writing the prose half

A usable card, in order:

1. **One sentence** saying what a row is. "Each row is a customer support message in Hindi or English with an intent label from a fixed set of nine."
2. **Collection.** When, from where, by what method, over what period.
3. **Labelling.** Who, with what instructions, and — the number almost nobody publishes — the **inter-annotator agreement**. If two labellers agreed on 78% of items, no model trained on it should be expected to exceed that by much, and knowing 78% before you start saves you weeks of chasing a ceiling.

4. **Splits**, with counts and how they were made. Say if the split is random, and say if it is by time or by document, because that choice decides whether your evaluation is honest.
5. **Known problems.** Duplicates left in, a class that is mostly one source, text that is machine-translated, a date range that stops.
6. **Personal data**, and what was done about it.
7. **What it should not be used for.** The most useful section and the rarest.

The contamination note

If your dataset contains text that appears in common web crawls — and almost anything published before 2023 does — then models trained on the web may have seen it. Say so. A benchmark that has leaked into pretraining corpora still has uses, and reporting a number on it as though it were held out is misleading. This is unresolved across the field, not a solved problem with a checkbox, and the honest position is disclosure rather than a claim of cleanliness.

The legal shape, stated honestly

Whether training a model on copyrighted material is permitted differs by country and is actively contested. Several jurisdictions have text-and-data-mining exceptions with conditions attached; others do not; litigation is ongoing in more than one. Nothing here is legal advice and nobody writing a dataset card can resolve it.

What you can do is make the question answerable by somebody qualified: record the source of every row, record the terms under which you obtained it, keep a `source` column, and keep the date. A dataset whose provenance is documented can be assessed later under whatever the law turns out to be. A dataset whose provenance was never recorded cannot be, and that is the failure that closes doors.

Do this now

Open the card of a dataset you have used and try to answer the four questions at the top of this lesson. If you cannot answer two of them, you do not know what you trained on — and that is a finding worth writing down before the next run.

BEFORE YOU MOVE ON

A dataset card declares Apache 2.0 while describing rows scraped from a site whose terms forbid redistribution. How should this be read?

A. The declared licence governs, since the uploader assumed the legal risk by publishing under it.

B. Scraping publicly accessible pages places the content outside the original terms, so the tag is consistent.

C. The tag cannot grant rights the compiler did not hold, so the licence and the stated origin conflict.

D. Only the source site can raise an objection, so the licence is valid for anyone who is not that site.

45 · 8 min

Media columns decode when you touch them

THE ONE THING TO KEEP

`Audio` and `Image` columns decode lazily on access, so `cast_column` can resample or defer decoding for free — and the silent failures in media work are a sampling rate that does not match the model, processor constants borrowed from another model, and augmentation frozen into a cache by `map`.

What is actually stored

```
ds = load_dataset("PolyAI/minds14", "en-US", split="train")
print(ds.features["audio"])
# Audio(sampling_rate=8000, mono=True, decode=True, id=None)

row = ds[0]
print(row["audio"]["array"].shape, row["audio"]
["sampling_rate"])
```

The stored column holds a path or the encoded bytes. The decoded waveform appears only when you access the row — `Audio` and `Image` are lazy decoding types. So a dataset of 100,000 photographs takes the disk of 100,000 compressed JPEGs, not of 100,000 raw arrays, and iterating decodes one at a time.

Two consequences worth holding:

- **Never call `ds.map` over a media column carelessly.** If your function touches `row["image"]`, every image decodes and the result is written back as decoded data, which can be fifty times the size. Use `set_transform` for augmentation, or return only the small derived columns you need.
- **`ds.cast_column("image", Image(decode=False))`** gives you paths and bytes without decoding, which is what you want for deduplication, hashing or copying files.

Sampling rate is the audio bug

Every speech model expects a specific rate. Whisper and most modern models want 16,000 Hz. Telephone corpora are often 8,000. Feed the wrong rate and you get no error and a much worse transcript, because the model hears something pitched and paced wrongly.

```
from datasets import Audio
ds = ds.cast_column("audio", Audio(sampling_rate=16_000))
```

`cast_column` resamples lazily on access. It costs nothing until the row is read, and it fixes the single most common silent failure in audio work. Check the model's expected rate in its processor config, not in its readme.

Note that upsampling 8 kHz telephone audio to 16 kHz does not restore information that was never recorded. It makes the model's input well-formed; it cannot make a narrowband recording wideband. Expect a genuinely higher error rate on telephone audio and do not attribute it to the resampling.

Image preprocessing belongs to the processor

```
from transformers import AutoImageProcessor
proc = AutoImageProcessor.from_pretrained(name)
print(proc.image_mean, proc.image_std, proc.size)
```

Every vision model has its own resize dimensions and normalisation constants. Using another model's constants produces a quiet accuracy drop with no error — the same shape of failure as the wrong sampling rate. Always load the processor from the same repository as the weights, exactly as with tokenizers.

```
def prep(batch):
    batch["pixel_values"] = proc(images=batch["image"], re-
turn_tensors="pt")["pixel_values"]
    return batch

ds.set_transform(prep)      # on access, not materialised
```

Augmentation is per epoch, so it is a transform

Random crops, flips, colour jitter and noise must differ between epochs. `map` would freeze one random draw into a cached file and train on the same "random" augmentation every time — a subtle bug that shows up as a model that overfits despite augmentation. `set_transform` re-runs on every access, which is what augmentation means.

Streaming media

```
ds = load_dataset("mozilla-foundation/common_voice_17_0", "hi",
                  split="train", streaming=True)
for row in ds.take(3):
    print(row["sentence"], row["audio"]["array"].shape)
```

Speech corpora are hundreds of gigabytes. Streaming is not an optimisation here, it is the only way most people can touch them at all. Many require accepting terms on the dataset page first, which is the gating mechanism from the tokens lesson.

Video, briefly

Video support exists and is younger. In practice most video work still means extracting frames and audio with `ffmpeg` and treating the result as an image dataset plus an audio dataset — which is free, scriptable, and avoids depending on decoder support that varies by environment.

What to check on any media dataset before training

1. **Sampling rate** or **image size** distribution — are they uniform, and do they match the model?
2. **Duration or dimension outliers.** One eight-hour recording in a corpus of ten-second clips will consume a whole batch and may exhaust memory.
3. **Corrupt files.** Iterate once with a try/except and count failures. There are always some.
4. **Silence and blank frames.** Cheap to detect, and they train the model on nothing.
5. **Speaker or source overlap between splits.** The media equivalent of leakage: the same speaker in train and test makes your word error rate optimistic in a way that does not survive deployment.

Do this now

Load any audio dataset, print its sampling rate, and compare it against what your chosen model's processor expects. If they differ, you have just found the reason a transcript was worse than you expected.

BEFORE YOU MOVE ON

A team applies random crops and flips with `ds.map` and finds the model overfits as if no augmentation were present. What is the mechanism?

A. `map` caches its output, so one random draw is frozen and every epoch trains on identical augmented images.

B. Augmentation applied before the image processor is undone by the processor's resize and normalisation step.

C. Lazy decoding means the augmentation function received encoded bytes rather than pixel arrays, so it silently did nothing.

D. Cached Arrow files store images losslessly, so the augmented copies are too similar to the originals to add variety.

46 · 10 min

Duplicates, leakage, balance, and label noise

THE ONE THING TO KEEP

Before training, four checks decide whether a result means anything: exact and near duplicates, leakage along the axis deployment must generalise across, class balance against the metric you will quote, and label noise that sets a ceiling no model can pass.

Four checks, an hour, before any training

This is the least glamorous lesson in the course and the one with the highest expected value. Every check below has, in somebody's project, been the difference between a result and an illusion.

1. Duplicates

Near-duplicates inflate everything. A dataset with 400 copies of one automated message looks like 400 rows and contains one row of information. Trained on, the model over-weights it; evaluated on, your test set is a repeated question.

Exact duplicates first, because they are free:

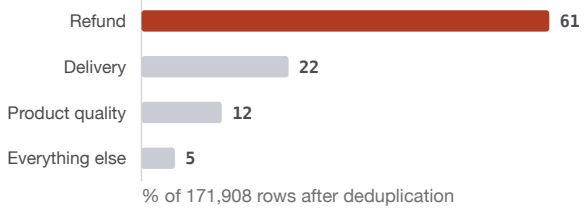
```
import hashlib

def norm(t):
    return " ".join(t.lower().split())

seen, keep = set(), []
for i, t in enumerate(ds["text"]):
    h = hashlib.md5(norm(t).encode()).hexdigest()
    if h not in seen:
        seen.add(h); keep.append(i)
ds = ds.select(keep)
print(f"removed {len(ds['text']) - len(keep)} exact
duplicates")
```

Near-duplicates need MinHash or embeddings. The `datatrove` library implements MinHash deduplication at corpus scale; for a few hundred thousand rows, embedding everything and dropping pairs above a cosine similarity of about 0.95 works and uses the tools from module six.

A support queue's four intents, counted before training



A model that always answers refund scores 61 per cent accuracy on this set and has learned nothing at all. That number will be the first one anybody quotes, which is why macro-F1 and per-class recall belong in the same sentence as it.

The diagnostic that tells you whether to bother: **if any single normalised string is more than 2% of your dataset, it is a template, not a sample.**

2. Leakage between splits

The check that invalidates published numbers:

```
train_hashes = {hashlib.md5(norm(t).encode()).hexdigest() for t
in ds["train"]["text"]}
overlap = sum(hashlib.md5(norm(t).encode()).hexdigest() in
train_hashes
              for t in ds["test"]["text"])
print(f"{overlap} of {len(ds['test'])} test rows appear in
train")
```

Exact overlap is the easy case. The subtler forms are the ones that catch experienced people:

- **Grouped leakage.** Three chunks from the same document split across train and test. The model has seen the neighbouring paragraph. Split by document, not by chunk.
- **Temporal leakage.** A random split of time-ordered data lets the model train on the future and predict the past. If deployment means predicting forward, split by date.
- **Speaker or user leakage.** The same person in both splits. You measure memorising them.
- **Preprocessing leakage.** Computing normalisation statistics, a vocabulary, or an imputation value over the whole dataset before splitting. The test set has influenced the pipeline.

The general rule: **split along the axis your deployment will have to generalise across.** If the model will see new customers, split by customer.

3. Class balance, and what it does to your metric

```
from collections import Counter
c = Counter(ds["label"])
for k, v in c.most_common():
    print(f"{k}: {v} ({v/len(ds):.1%})")
```

At 98% negative, a model that always says negative scores 98% accuracy. That number is worthless and it will be the first number anybody quotes. Report per-class precision and recall, or macro-F1, and say which you used.

What to do about imbalance, in the order usually worth trying:

1. **Nothing.** Choose a metric that is not fooled and a threshold set on a validation set. This is right more often than people expect.
2. **Class weights in the loss**, which costs one argument and no data.
3. **Collect more of the rare class**, which is the only approach that adds information.
4. **Undersample the majority**, which throws information away.
5. **Oversample or synthesise the minority**, which risks the model memorising a handful of examples many times.

4. Label noise

Read a hundred rows. Not a summary, the rows.

An error rate of 5–10% in labels is normal in crowd-sourced data, and it puts a ceiling on measured accuracy that no model can pass — a perfect model scores 92% against 8% wrong labels, and the missing 8% is the labels, not the model. If your card reports inter-annotator agreement, that figure is roughly your ceiling.

The cheap detector: train a quick model, look at the examples it is most confident about and wrong on. A large share of them are mislabelled rather than hard. This takes twenty minutes and routinely finds systematic errors — a whole category labelled by a rule that was wrong.

The other two things to look at

Language and script mix. `langdetect` or `fasttext` over a sample. A "multilingual" dataset that is 94% English trains a model that is 94% English.

Length distribution. The 95th percentile decides your `max_length` and therefore your compute. A median of 80 tokens with a maximum of 40,000 means one row will dominate a batch.

Write the results down

```
rows 184,203 → 171,908 after exact dedup (6.7%)
test/train exact overlap: 0 (after dedup)
split: by ticket_id, not by row
classes: refund 61%, delivery 22%, quality 12%, other 5%
langs: en 71%, hi 18%, hinglish 11% (sample of 2,000)
length p50 84 tokens, p95 612, max 31,004 → max_length 640
read 100 rows: 6 look mislabelled (4 refund→delivery)
```

Eight lines. It goes in the dataset card and in the repository. Every surprising result later starts by reading it, and a team that has it argues about causes instead of about facts.

Do this now

Run the duplicate check and the leakage check on the dataset you are currently using. Most people find something on the first attempt — and finding it before training is worth more than any model choice in this course.

BEFORE YOU MOVE ON

A document-chunking pipeline splits chunks randomly into train and test, and the model scores far better offline than in production. What is the most likely cause?

A. The chunks are too short, so the model memorises phrasing rather than learning the task.

B. Random splitting failed to stratify by class, so the test set's class mix differs from production.

C. Neighbouring chunks of one document land in both splits, so the test material was seen.

D. Production input is not chunked the same way, so the model receives inputs of a length it never saw.

47 · 9 min

From a dataset to batches a trainer can eat

THE ONE THING TO KEEP

Training needs `input_ids`, `attention_mask` and a column literally named `labels`, with padding done per batch by a collator rather than at map time — and for instruction tuning the prompt positions are set to `-100` so loss is computed only on the response.

The three things the model needs

Whatever you are training, the loop wants tensors of the right shape with the right labels. For text that means: `input_ids`, `attention_mask`, and `labels`. Everything in this lesson is getting from a table of strings to those three.

Classification: labels are just a column

```
def prep(batch):
    return tok(batch["text"], truncation=True, max_length=512)

ds = ds.map(prep, batched=True, remove_columns=["text"])
ds = ds.rename_column("label", "labels")
ds.set_format("torch")
```

Three specific points. The column must be called **`labels`**, plural, because that is what the model's forward signature expects — `label` silently becomes an unused column and the loss is never computed. **Do not pad here**; padding at map time pads everything to 512 and wastes compute. And `set_format("torch")` makes rows come back as tensors without copying the underlying Arrow data.

Dynamic padding, with a collator

```
from transformers import DataCollatorWithPadding
collator = DataCollatorWithPadding(tokenizer=tok)
```

The collator pads each batch to that batch's longest item rather than to a global maximum. Combined with length-grouped batching (`group_by_length=True` in `TrainingArguments`) this is frequently a twofold speed-up for free, for the reasons the batching lesson gave.

Causal language modelling: labels are the inputs

```
from transformers import DataCollatorForLanguageModeling
collator = DataCollatorForLanguageModeling(tokenizer=tok,
mlm=False)
```

With `mlm=False`, the collator copies `input_ids` into `labels` and sets padded positions to `-100`, which the loss ignores. The model shifts internally, so you do not shift yourself — doing it manually as well is a classic off-by-one that produces a model which predicts the current token instead of the next one and appears to learn beautifully while being useless.

What one instruction-tuning row has to contain

input_ids	The whole conversation, rendered by the model's own chat template
attention_mask	1 for real tokens, 0 for padding the collator adds per batch
labels	Plural. Named label instead and the loss is silently never computed.
-100 across the prompt	Loss falls on the response only, so the model does not learn to write the questions

Decode a collated batch and read both the input and the unmasked labels before you start a long run. A doubled beginning-of-sequence token, a chat template that did not render, and masking that covered the wrong span all show up here and nowhere else.

With `mlm=True` it masks 15% of tokens instead, for BERT-style training.

Instruction tuning: mask the prompt

For supervised fine-tuning on instruction data, you usually do not want loss on the prompt — only on the response. Otherwise the model spends its capacity learning to generate the questions people ask it.

The mechanism is setting prompt positions in `labels` to `-100`:

```
def prep(example):
    prompt = tok.apply_chat_template(example["messages"][:-1],
                                     tokenize=False, add_gener-
ation_prompt=True)
    full = tok.apply_chat_template(example["messages"], tok-
enize=False)

    p_ids = tok(prompt, add_special_tokens=False)["input_ids"]
    f_ids = tok(full, add_special_tokens=False)["input_ids"]

    labels = list(f_ids)
    labels[:len(p_ids)] = [-100] * len(p_ids)
    return {"input_ids": f_ids, "labels": labels,
            "attention_mask": [1] * len(f_ids)}
```

Note `add_special_tokens=False` on both encodes — the chat template has already inserted the special tokens, and letting the tokenizer add a second beginning-of-sequence marker shifts every offset by one and breaks the masking silently. TRL's `SFTTrainer` does all of this for you, which is why the training module reaches for it; doing it once by hand is how you understand what it is doing.

Packing, for pretraining

When training on plain text, padding wastes a lot. Packing concatenates documents and cuts fixed-length blocks:

```
def group(batch, block=1024):
    ids = sum(batch["input_ids"], [])
    n = (len(ids) // block) * block
    chunks = [ids[i:i+block] for i in range(0, n, block)]
    return {"input_ids": chunks, "labels": [c[:] for c in
chunks]}

lm_ds = tokenised.map(group, batched=True,
remove_columns=tokenised.column_names)
```

No padding at all, so every token in every batch is doing work. The cost is that documents bleed across block boundaries and attention crosses between unrelated texts. For pretraining this is standard and the effect is small; for instruction tuning it needs attention masking between packed examples, which the libraries handle and hand-rolled code usually does not.

Check the batch before you train

The cheapest hour you will ever save:

```
batch = collator([ds[i] for i in range(4)])
print({k: v.shape for k, v in batch.items()})
print(tok.decode(batch["input_ids"][0]))
lab = batch["labels"][0]
print(tok.decode([t for t in lab if t != -100]))
```

Read that decoded text. It is exactly what the model will see. Three things show up here and nowhere else: a doubled beginning-of-sequence token, a chat template that did not render, and prompt masking that covered the wrong span. Each of those costs a full training run if you find it afterwards.

Do this now

Print one collated batch for whatever you plan to train, decode both the input and the unmasked labels, and read them out loud. If they are not what you intended, you have just saved the cost of the run.

BEFORE YOU MOVE ON

A fine-tune runs without error, the loss falls, and the resulting model generates plausible user questions rather than answering them. What was most likely wrong in data preparation?

A. Padding was applied at map time to a fixed maximum, so the model learned from padded positions.

B. The label column was named `label` rather than `labels`, so no loss was computed on the responses.

C. The chat template was applied after tokenisation, so role markers were encoded as ordinary text.

D. Prompt tokens were not masked to `-100`, so the model was trained on the questions too.

CASE STUDY · MODULE 5

Ninety-four in the lab, sixty-one in the field

A state agriculture department's pest advisory in Nagpur, forty-one thousand photographs sent in by cotton farmers over two seasons, and a figure already printed in a press note.

The advisory's classifier identified five cotton pests from a photograph a farmer sent from a phone. It reported ninety-four per cent accuracy on a held-out test split, and the press note said so.

The advisory itself was a genuinely good idea and popular. A farmer photographed a leaf, sent it, and got back a named pest and a recommendation within a few minutes, in Marathi, at no cost. Eleven thousand farmers had used it in the second season.

Over the following month the department's own agronomists checked the advisory's answers against what they found in the field. They recorded sixty-one per cent.

Nothing was wrong with the model. The split was the problem, and it had been made the way almost everybody makes one: randomly, over photographs.

A farmer reporting a problem does not send one photograph. He sends six — the same plant, the same field, the same light, minutes apart. A random split scattered those six across train and test. For a large share of the test set the model had already seen the neighbouring photograph of the same leaf, and what the ninety-four per cent measured was its ability to recognise a photograph it had effectively already been trained on.

An hour of checks that should have run before the training found two more things. Four thousand one hundred exact duplicates, nearly all of them one automated thumbnail resent by the intake system when a message failed. And a class balance nobody had written down: one pest was fifty-eight per cent of the rows, because collection had started in its season, while the rarest was three per cent — and the rare one was the one that costs a farmer the crop.

The decision

Arun Deshmukh, who led the team, had a number in a press note and a minister's office that liked the number.

Re-splitting by contributor and retraining meant about two weeks, and it meant publishing a lower figure after a higher one, which in a government department is a specific and well-understood kind of cost. Somebody would have to explain, in writing, why the first number had been wrong.

Doing nothing meant the advisory carried on telling farmers something that was right three times in five, in a season, about a pest that eats cotton.

There was also a third thing on the table that nobody had asked for. The rare class at three per cent meant that whatever the headline accuracy said, the advisory's performance on its most expensive case had never been measured separately at all.

Arun made one decision before any of the others that made the rest possible. He asked the agronomists to keep recording field agreement, weekly, whatever the model said. It cost each of them about twenty minutes a week and it is the only reason anybody knew there was a problem at all. A held-out test split can be wrong in a way that no amount of staring at it reveals; a measurement taken outside the pipeline cannot be fooled by anything inside it.

The duplicate check is worth its own sentence. Four thousand one hundred copies of one automated thumbnail is two things at once: a training set that over-weights a picture of nothing, and a test set that repeatedly asks the same non-question. It took one query to find and one line to remove, and it had been sitting in the data for two seasons while three people argued about model architectures.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They re-split by contributor identifier rather than by photograph, dropped the exact duplicates, and retrained. Test accuracy fell to sixty-seven per cent — within six points of what the agronomists had measured in the field. That agreement was the first evidence the team had that their evaluation measured anything real, and Arun used exactly that argument with the minister's office.

Per-class recall told the sharper story: eighty-one per cent on the majority pest and thirty-four on the rare, expensive one.

The correction went out as an update rather than a retraction. It said the earlier figure had been measured on a split that let the model see neighbouring photographs of the same plant, that the corrected figure was sixty-seven per cent, and that recall on the pest that matters most was thirty-four per cent and was now the work. It also announced a product change that came straight out of that number: the advisory would show two candidates and a button that sends the photograph to an agronomist, instead of one confident label.

Nobody at the department enjoyed the week. The advisory is still running.

Name the axis the split should have been made along, and state the general rule it follows.

By contributor, because a contributor sends many photographs of the same plant and the deployment has to generalise to farmers it has never seen. The general rule is to split along the axis your deployment must generalise across: by customer if you will see new customers, by document if chunks come from documents, by date if you are predicting forward, by speaker if the users are new speakers. A random row split is only correct when rows are genuinely independent, which they rarely are.

Why did the team treat ninety-four and sixty-one collapsing to sixty-seven as good news?

Because the laboratory number and the field number finally agreed. Before the fix, the evaluation measured something that did not exist outside the test split, so no decision could be taken from it. After the fix, a six-point gap between held-out measurement and field observation is close enough that the measurement can be

used to compare the next model against this one. An evaluation that tracks reality is worth more than a high number that does not.

The rare class sat at thirty-four per cent recall and changed the product. Why is that the right number to act on, and overall accuracy the wrong one?

Because the classes are not equally consequential and they are not equally common. At fifty-eight per cent majority, a model can post a respectable overall accuracy while barely detecting the pest that destroys a crop, and the overall figure hides exactly that. Per-class recall names it. It also converts directly into a product decision: at thirty-four per cent you do not show one confident label, you show two candidates and a route to a human.

MODULE 5 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 You will make three training passes over a 200 GB corpus and you have 400 GB of free disk. Which is right, and why?

A. Stream it, because streaming never writes anything to disk at all.

B. Download it, because re-streaming for each epoch re-downloads everything.

C. Stream it, because memory mapping would need the data to fit in RAM.

D. Download it, because a streamed dataset does not carry its label columns with it.

2 Shuffling a 40 GB dataset returns instantly and uses no extra memory. What makes that possible?

A. Arrow already stores rows in a randomised order on disk.

B. The shuffle is deferred and actually happens during the first training step.

C. Shuffle builds an indices map, and the underlying table is never copied.

D. The rows are compressed, so moving them around costs almost nothing.

3 You need to turn 1,000 documents into 40,000 chunks. Which is true of map with batched=True?

A. It may return a different number of rows, if you remove the old columns.

B. It must return exactly as many rows as it was given in the batch.

C. It cannot change row counts at all, so filter has to be used for that instead.

D. It changes row counts only when num_proc is set to more than one.

4 A GROUP BY over one column of a 50 GB public dataset transfers only a few megabytes. What makes that possible?

A. The Hub caches the result of every query for about a month.
B. DuckDB takes a random sample rather than reading the whole table.
C. The dataset viewer has already computed and cached every possible aggregate in advance.
D. Parquet keeps column statistics in its footer, so only needed ranges are fetched.

5 A model scores 94 per cent on a random split of chunked documents and 61 per cent in deployment. What fits best?

A. Chunks of one document landed in both splits, so the model had seen neighbours.
B. The test split was too small for the measurement to be reliable.
C. The deployment runtime quantized the model and lost accuracy.
D. The class balance shifted between the training period and the deployment period.

6 Training runs without error, and the loss is completely flat from step one. Which of these produces exactly that?

A. The learning rate is somewhat lower than it should be for this model.
B. Gradient checkpointing is enabled, which is recomputing activations.
C. The label column is named label rather than labels, so loss is never computed.
D. The batch size is far too small for a model of this size to learn stably at all.

MODULE 6

Embeddings, and search that understands

An embedding is a list of numbers that places a piece of text, an image or a clip of audio somewhere in a space where nearby means similar. It is the concept under semantic search, retrieval-augmented generation, recommendation, clustering and deduplication — and it runs on a phone. This block builds a working search over your own documents from first principles, then measures it honestly instead of trusting a leaderboard.

> BY THE END OF THIS MODULE YOU CAN
>
> Build and measure a semantic search over your own documents — choosing an embedding model against your language and text length, chunking deliberately, normalising vectors for the similarity you use, adding a cross-encoder reranker, and reporting recall@k on questions you wrote yourself

48 · 9 min

A list of numbers where near means similar

THE ONE THING TO KEEP

An embedding is a fixed-length vector whose geometry encodes meaning, produced by a model trained contrastively to place similar texts close together — so search, clustering, deduplication and few-shot classification are all the same operation, and its central weakness is that similarity is not relevance and negation barely moves the vector.

The smallest true description

An embedding model takes a piece of text and returns a fixed-length list of floating-point numbers — 384, 768 or 1,024 of them, usually. The same model always produces the same length, whatever the input length. That is the whole interface.

What makes it useful is a property of how the model was trained: texts with similar meaning land close together in that space, and texts with different meaning land far apart. "Where is my parcel?" and "My delivery has not arrived" share almost no words and sit close together. "Bank of the river" and "bank account" share a word and sit far apart.

```
from sentence_transformers import SentenceTransformer

m = SentenceTransformer("sentence-transformers/all-MiniLM-L6-
v2")
v = m.encode("Where is my parcel?")
print(v.shape, v[:5])
# (384,) [ 0.041 -0.113  0.028  0.067 -0.019]
```

Those 384 numbers mean nothing individually. No dimension is "formality" or "topic". Only the geometry is meaningful — distances and angles between

vectors. People waste time trying to interpret single dimensions; there is nothing there to find.

Where the vectors come from

A transformer produces one vector per token. An embedding model reduces that to one vector per text, usually by **mean pooling** — averaging the token vectors, weighted by the attention mask so padding contributes nothing — or by taking the `[CLS]` position, or by a small trained head.

That pooling step is why you should not roll your own from a base model. `AutoModel` gives you token vectors; getting from there to a good sentence vector requires the pooling the model was trained with, and guessing gives you something that works badly enough to be misleading. `sentence-transformers` stores the right pooling alongside the weights and applies it for you.

The deeper point: an embedding model is trained with a **contrastive objective** — shown pairs that should be close and pairs that should be far, and adjusted until they are. A base language model was never asked to do this, which is why `bert-base-uncased` produces famously poor sentence embeddings despite being a perfectly good language model. The capability is trained in, not inherent.

Why this is worth your attention

Once text is a vector, several different-looking problems become one problem:

- **Search** — embed the query, find the nearest documents. Works across paraphrase and, with the right model, across languages.
- **Retrieval-augmented generation** — the same search, with the results pasted into a prompt.
- **Clustering** — group vectors, get topics nobody labelled.
- **Deduplication** — pairs above a similarity threshold are near-duplicates, which is the check the data-quality lesson needed.
- **Classification with no training data** — embed a handful of labelled examples per class, classify by nearest centroid. This works surprisingly well at twenty examples per class.

- **Recommendation** — items near the ones somebody liked.

All six are the same two lines of code with different post-processing.

What it costs

This is the part that decides whether you can use it, and the numbers are friendly. `all-MiniLM-L6-v2` is 90 MB, six transformer layers, and encodes several hundred short texts per second on an ordinary CPU. Embedding 100,000 documents on a laptop is minutes, not hours, and it is a **one-off**: store the vectors and searching them afterwards is arithmetic.

A million 384-dimensional float32 vectors is 1.5 GB — fits in RAM. At float16 it is 750 MB. This is why semantic search over a substantial corpus is genuinely free, and why the module-three advice about small machines holds here.

The limitations, by mechanism

Similarity is not relevance. Two texts about the same topic are close whether one answers the other or not. "How do I cancel my subscription?" is close to "How do I renew my subscription?" because they are near-identical in form and topic, and one is a wrong answer to the other. Embeddings retrieve the neighbourhood; deciding within it is what the reranker lesson is for.

Negation is weak. "The battery lasts all day" and "The battery does not last all day" are close in most embedding spaces, because the words are nearly identical and one token is doing all the semantic work. For any task where negation matters, this is a real failure and not a tuning problem.

Long text averages away. Pool a 3,000-word document into 384 numbers and the specific fact you wanted is diluted by everything else in it. This is the whole reason chunking exists.

Out-of-domain is out-of-domain. A model trained on web English places medical abbreviations or Hinglish code-switching roughly at random. Check on your own text.

Do this now

Encode five sentences: two paraphrases, two on the same topic with opposite meaning, and one unrelated. Print all ten pairwise similarities. The paraphrase pair will be high, the unrelated pair low — and the opposite-meaning pair will be uncomfortably high, which is the most useful thing this lesson has to show you.

BEFORE YOU MOVE ON

A support search built on embeddings keeps returning "How do I renew my subscription?" for the query "How do I cancel my subscription?". What does this reveal?

A. Embeddings measure topical and structural similarity, which is high for two questions that differ by one decisive word.

B. The vectors were not normalised, so cosine similarity is being distorted by differences in text length.

C. The model's context window truncated the query, removing the word that distinguishes the two intents.

D. The index holds too few documents, so the nearest neighbour is the closest available rather than a genuinely relevant one.

49 · 8 min

The library, and the four calls you need

THE ONE THING TO KEEP

`SentenceTransformer.encode` with `normalize_embeddings=True` plus one matrix multiply is a complete search engine over a hundred thousand documents, and the two details that silently cost accuracy are forgetting a family's query prefix and letting a vectors array drift out of order with its metadata.

Install and encode

```
pip install sentence-transformers
```

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("BAAI/bge-small-en-v1.5")

vectors = model.encode(
    ["first document", "second document", "third document"],
    batch_size=64,
    normalize_embeddings=True,
    show_progress_bar=True,
)
print(vectors.shape)     # (3, 384)
```

Four arguments worth setting every time:

- **`batch_size`** — 32 to 128 on a CPU, larger on a GPU. Encoding a list is far faster than encoding in a loop, for the same reasons as module two.
- **`normalize_embeddings=True`** — scales every vector to unit length, which makes dot product and cosine similarity identical and lets you use the faster one. Set it and then be consistent; mixing normalised and unnormalised vectors in one index gives you nonsense.

- `show_progress_bar` — for anything over a few thousand items, so you know whether to make tea.
- `convert_to_tensor=True` when you want torch tensors on the GPU rather than NumPy arrays.

Similarity, in one call

```
sim = model.similarity(vectors[:1], vectors)
print(sim)      # tensor([[1.0000, 0.6421, 0.3117]])
```

`model.similarity` uses whatever the model was trained for — cosine for most, dot product for a few — which removes a decision you would otherwise get wrong. With normalised vectors you can also do it by hand:

```
import numpy as np
scores = vectors @ query_vec          # a plain matrix multiply
top = np.argsort(-scores)[:5]
```

One matrix multiply is a complete search engine over a hundred thousand documents, and it runs in milliseconds. Do not reach for a vector database until you have measured that this is too slow.

The prompt prefixes almost nobody reads about

Several of the best models were trained with an instruction prefix on the query but not on the documents. BGE, E5, GTE and Nomic families all do some version of this. Omit it and you lose several points of accuracy for no visible reason.

```
# E5 family
docs = model.encode([f"passage: {d}" for d in documents])
q = model.encode("query: where is my parcel")

# BGE family, via the library's own prompt support
q = model.encode("where is my parcel", prompt_name="query")
```

The model card says which. Read it before benchmarking anything — a large share of "this model scored worse than advertised" reports are this.

Matryoshka: shorter vectors on purpose

Some recent models are trained so that the first *n* dimensions are usable on their own. You can truncate 768 down to 256 and keep most of the quality:

```
model = SentenceTransformer("nomic-ai/nomic-embed-text-v1.5",
trust_remote_code=True)
v = model.encode(texts, truncate_dim=256)
```

That is a third of the memory and roughly a third of the search time, for a small accuracy cost. On a million documents it is the difference between fitting in RAM and not. It only works for models trained this way — truncating an ordinary model's vectors destroys them, because nothing made the early dimensions more important than the late ones.

Re-encoding is the cost that bites later

Every vector in your index was produced by one model at one revision. Change the model and every stored vector is worthless — you must re-encode the whole corpus. On a million chunks that is hours of compute and, if you are calling a hosted API, a real bill.

Two habits follow. Pick the model with the thirty-question test **before** encoding at scale rather than after. And keep the chunk text stored alongside the vectors, not only the vectors, so re-encoding is a local job rather than a re-extraction from source PDFs you may no longer have.

Saving and reloading vectors

```
import numpy as np
np.save("vectors.npy", vectors.astype("float16"))
vectors = np.load("vectors.npy").astype("float32")
```

Store at float16 and cast back for arithmetic: half the disk, no measurable retrieval loss. Store the document ids and texts alongside in a separate file, in the same order, and never sort one without the other — a vectors array whose order has drifted from its metadata is the single most annoying bug in this area, because every result is plausible and wrong.

Write the model name and revision into the file too. Vectors from two different models are not comparable and there is nothing in the array to tell you which produced it.

Do this now

Encode 1,000 sentences of your own, save them as float16, reload, and search with a matrix multiply. Time the search. It will be a few milliseconds, and that number is why most projects never need a vector database.

BEFORE YOU MOVE ON

A team truncates 768-dimensional vectors from an ordinary embedding model to 256 dimensions to save memory, and retrieval quality collapses. Why did it work for a colleague using a different model?

A. The colleague normalised vectors after truncating, which restores the geometry that truncation distorts.

B. The colleague's model has fewer dimensions to begin with, so proportionally less information was lost.

C. The colleague's model was trained so its early dimensions carry most of the information.

D. Truncation requires re-encoding the corpus at the shorter length, which the colleague did and the team did not.

50 · 9 min

Picking the model, and reading MTEB without being fooled

THE ONE THING TO KEEP

Maximum sequence length and language coverage decide an embedding model more often than its benchmark rank does, and MTEB's average column mixes tasks you do not have — so use it for a shortlist of three and settle it with thirty questions of your own.

Four specifications, in order of how often they decide it

1. Maximum sequence length. Most embedding models truncate at 512 tokens, and many at 256. Text beyond that is silently discarded — the same silent truncation as module two, with the same consequence. If your chunks are longer than the limit, you are embedding their beginnings. Check `model.max_seq_length` and chunk to fit it.

2. Language. An English-trained model on Hindi, Bengali or Tamil produces vectors that are not random but are close to useless — everything is moderately similar to everything. The multilingual families (`paraphrase-multilingual-MiniLM` , `multilingual-e5` , `BGE-M3` , `LaBSE` , `jina-embeddings-v3`) are trained across many languages and, crucially, place translations of the same sentence close together. That cross-lingual property lets somebody search Hindi documents with an English query, which for a lot of this readership is the whole feature.

3. Dimensions. 384 is small and fast; 768 is the common middle; 1,024 and above are slower and heavier to store. More dimensions is not reliably better — a well-trained 384 model regularly beats a poorly trained 1,024 one. Dimensions decide cost with certainty and quality only loosely.

4. Size on disk. 90 MB to 2 GB. This decides whether it runs on a phone, in a browser or on a free Space.

MTEB, and its three traps

The Massive Text Embedding Benchmark scores models across dozens of tasks in several categories. It is the best available survey and it is a leaderboard, with everything the leaderboard lesson says about leaderboards.

Trap one: the average. The headline column averages retrieval, classification, clustering, reranking and semantic similarity. If you are building search, only the retrieval column matters, and the ordering within it differs from the overall ordering. Sort by the task you actually have.

Trap two: overfitting to the benchmark. MTEB tasks are public. Training on data that resembles them raises the score without raising real quality, and several high-ranking models have been credibly argued to do this. A model that is two points better on MTEB and was released by a team optimising for MTEB is not reliably two points better on your tickets.

Trap three: the language you need is not there. MTEB's multilingual coverage has grown and is still thin for many Indian languages. A model absent from a language's table is not necessarily bad at it; nobody measured.

Use MTEB to build a shortlist of three. Decide between them yourself.

The decision in practice

- **English, small, fast, free everywhere** — `all-MiniLM-L6-v2` (90 MB, 384 dims). Still the right default for a first version, and the most-exercised embedding model on the Hub.
- **English, better quality** — `BAAI/bge-base-en-v1.5` or `gte-base`. 768 dims, around 400 MB.
- **Multilingual, including Indic languages** — `intfloat/multilingual-e5-base` or `BAAI/bge-m3`. The latter handles 8,192 tokens and does dense and sparse retrieval together.
- **In a browser or on a phone** — a quantized ONNX build of MiniLM, which transformers.js loads directly.
- **Long documents** — a model with a long window, or chunking, and chunking is usually better.

The test that actually decides it

Thirty questions, written by you, against your own documents, each with the document you know should come back.

```
for name in shortlist:
    m = SentenceTransformer(name)
    D = m.encode(docs, normalize_embeddings=True)
    Q = m.encode(queries, normalize_embeddings=True, prompt_-
name="query")
    ranks = [(-(Q[i] @ D.T)).argsort().tolist().index(gold[i])
for i in range(len(Q))]
    print(name,
          "recall@1", sum(r == 0 for r in ranks) / len(ranks),
          "recall@5", sum(r < 5 for r in ranks) / len(ranks))
```

Twenty minutes, and it answers the question MTEB cannot: which of these is best **on your text, in your languages, at your chunk length**. Differences of five or ten points show up clearly at thirty questions. Differences of one point do not, and you should not act on them.

Cost, since people ask

Hosted embedding APIs charge per token and are convenient. A local MiniLM costs nothing after the download and runs on the hardware you have. For under a few million documents, local is cheaper, private, and fast enough — and it does not change under you when a provider deprecates a model, which is a real failure mode that invalidates an entire stored index.

Do this now

Build the thirty-question set before you pick a model. It is the artefact that makes every later decision in this module checkable, and it is the one thing a leaderboard can never give you.

BEFORE YOU MOVE ON

Why does sorting MTEB by its headline average frequently pick the wrong model for a search application?

A. The average is computed over models rather than tasks, so it reflects relative rank instead of absolute quality.
B. The average weights larger models more heavily, since they appear in more task categories.
C. Retrieval scores are computed on public corpora, so they are the least trustworthy column in the table.
D. It blends retrieval with clustering and similarity tasks, which rank models differently.

51 · 8 min

Cosine, dot product, and the normalisation that connects them

THE ONE THING TO KEEP

On unit-length vectors cosine, dot product and Euclidean distance rank identically, so normalising at encode time removes the choice — and because each model spreads its space differently, similarity scores are for ranking rather than for thresholds copied from anyone else.

Three measures, and when they differ

Cosine similarity is the cosine of the angle between two vectors. It ignores length entirely and ranges from −1 to 1, though in practice text embeddings rarely go below about 0 because the vectors occupy a narrow cone of the space.

Dot product multiplies element-wise and sums. It is cosine multiplied by both vectors' lengths, so it rewards longer vectors.

Euclidean distance is straight-line distance. Smaller is better, which inverts every comparison and is a routine source of reversed results.

The connection that makes this simple: **on unit-length vectors, ranking by cosine, dot product and Euclidean distance gives the identical order.** Normalise once at encode time and the choice stops mattering:

```
v = model.encode(texts, normalize_embeddings=True)
scores = v @ q          # dot product == cosine, and it is the
fastest
```

If you do not normalise, dot product favours documents whose vectors happen to be long, which correlates weakly with length and topic frequency — so unnormalised dot-product search has a subtle bias toward a particular kind of document and nobody can see it in the results.

Normalise everything at encode time, then use dot product. That one rule removes the entire category.

What the numbers mean, which is less than you think

A cosine of 0.82 between two texts means nothing on its own. Embedding models differ in how they spread their space: one model's "clearly related" is 0.85, another's is 0.55. Absolute thresholds do not transfer between models, and a threshold copied from a tutorial written for a different model is a wrong threshold.

Two consequences:

1. **Rank, do not threshold**, wherever you can. "Top five" is portable; "above 0.8" is not.
2. **When you must threshold** — deduplication, a cutoff for "no good answer" — derive it from your own data. Encode a few hundred known-related and known-unrelated pairs, plot the two distributions, and put the threshold where they separate. That takes fifteen minutes and produces a number that is true for your model and your text.

The high floor

Encode two unrelated sentences with a typical model and you may see 0.3 or 0.4 rather than 0. This is normal. Text embeddings occupy a narrow region of the space, so nothing is ever truly orthogonal to anything else.

It means a similarity of 0.5 is not "half related". Calibrate on your own pairs before you interpret any score, and be suspicious of any product that shows users a raw similarity as a percentage — it reads as a confidence and is not one.

When Euclidean is the right choice

For clustering algorithms that assume Euclidean geometry — k-means in particular — use Euclidean distance on normalised vectors, which is equivalent to cosine for ranking and is what the algorithm's mathematics expects. For pure retrieval, dot product on normalised vectors is faster and equivalent.

Vector databases ask you to choose a metric when you create an index. Choose the one matching how you normalised, and write it down, because changing it later means rebuilding the index.

A practical check

```
import numpy as np
v = model.encode(sample_texts)
print("norms:", np.linalg.norm(v, axis=1)[:5])
```

If those print as 1.0, the vectors are normalised. If they print as 8.3, 11.2, 6.7, they are not, and any dot-product search over them is length-biased. This one line catches a bug that otherwise shows up as "search is a bit worse than I expected" and is never diagnosed.

Maximal marginal relevance, briefly

Top-k by similarity often returns five near-identical chunks — because if one paragraph matches, its neighbours do too. Maximal marginal relevance picks

the next result by balancing similarity to the query against dissimilarity to what is already selected:

```
# pick k items trading off relevance (to q) against redundancy
(to chosen)
score = lam * sim_to_query[i] - (1 - lam) * max(sim(i, j) for j
in chosen)
```

With `lam` around 0.7 you get results that cover the question rather than repeating one part of it. For retrieval feeding a language model this matters a great deal: five copies of one paragraph waste the context window that five different paragraphs would have filled.

Do this now

Encode your corpus twice, once normalised and once not, and compare the top five for three queries under dot product. Where the results differ, look at the lengths of the documents that moved. That is the bias, made visible.

BEFORE YOU MOVE ON

A tutorial's threshold of 0.8 for "related" produces almost no matches when applied with a different embedding model. Why?

A. The new model returns Euclidean distances rather than cosine similarities, so the comparison direction is inverted.

B. Models spread their embedding space differently, so the score at which texts are genuinely related varies by model.

C. The new model's vectors are unnormalised, so cosine similarity cannot exceed the shorter vector's length.

D. Higher-dimensional spaces push all pairwise similarities toward zero, so thresholds must scale with dimension count.

52 · 9 min

Chunking decides what your search can find

THE ONE THING TO KEEP

Chunk size and boundaries decide what retrieval can find at all, so split on structure rather than character counts, count in the model's own tokens because scripts differ, carry the document and section title into the embedded text, and consider retrieving small chunks while passing their parent sections to the model.

Why a whole document is the wrong unit

Embed a 40-page manual into one 768-number vector and you get the average of everything in it. A query about one specific procedure matches weakly, because that procedure is one fortieth of what the vector describes. Meanwhile the model truncated at 512 tokens anyway, so thirty-nine of those pages were never read.

Chunking splits documents into pieces small enough to be about one thing. It is the step that decides what your system can find, and it is almost always the highest-leverage thing to change when retrieval is disappointing — more so than swapping the model.

Sizes, with actual numbers

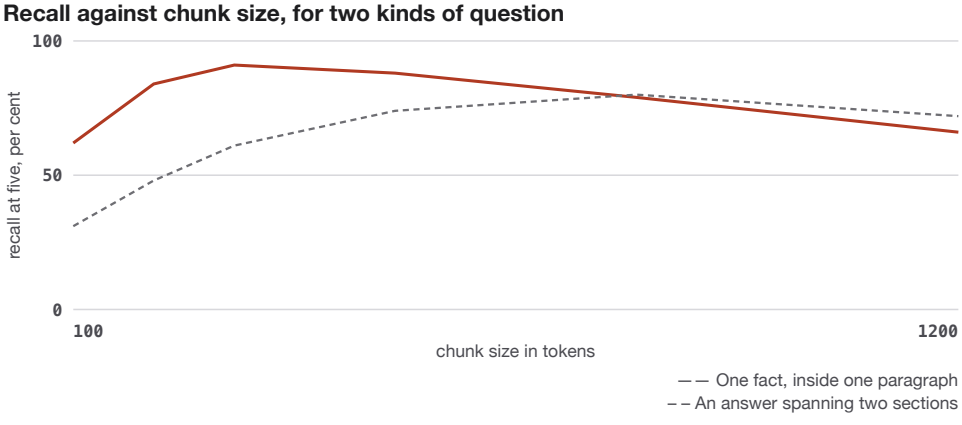
- **Under 100 tokens** — precise matches, no context. The retrieved chunk often does not contain enough to answer.
- **200–400 tokens** — the usual sweet spot for question answering over documents.
- **500–800 tokens** — better for summarising and broad questions, at the edge of many models' windows.
- **Over 1,000 tokens** — beyond most embedding models' limits, and dilute.

Start at 300 tokens with 50 tokens of overlap and adjust once you have measured.

Overlap exists because a fact can straddle a boundary. A 50-token overlap means a sentence cut in half appears whole in one of the two chunks. It costs storage proportional to the overlap fraction and prevents a failure that is otherwise invisible.

Split on structure, not on character counts

The naive version cuts every 1,500 characters, mid-word, mid-sentence, mid-table. Better, in ascending order of effort and quality:



One corpus, one embedding model, thirty questions, and chunk size the only thing changed. The peak is in a different place for the two kinds of question, which is the argument for retrieving small chunks and passing their parent sections to the model rather than picking one size for everything.

By separator, recursively. Try paragraph breaks first; if a piece is still too long, split on sentences; if still too long, on words. A dozen lines of Python, and it keeps paragraphs whole whenever it can:

```
def split(text, limit=1200, seps=("\n\n", "\n", ". ", " ")):
    if len(text) <= limit or not seps:
        return [text]
    sep, rest = seps[0], seps[1:]
    parts, out, cur = text.split(sep), [], ""
    for p in parts:
        if len(cur) + len(p) + len(sep) <= limit:
            cur += (sep if cur else "") + p
        else:
            if cur: out.append(cur)
            out.extend(split(p, limit, rest) if len(p) > limit
else [p])
            cur = ""
    if cur: out.append(cur)
    return out
```

By document structure. Markdown headings, HTML sections, PDF pages, slide boundaries. A chunk that is one section of a manual is a chunk about one thing, by construction. This is the best option when the structure exists.

By semantics. Embed each sentence, cut where consecutive sentences become dissimilar. Elegant, slower, and in most measured comparisons only slightly better than splitting on headings. Try the cheap thing first.

Carry the context with the chunk

A chunk stripped of its origin is much harder to use. Store metadata alongside every vector, and prepend the important parts to the text you embed:

```
chunk_text = f"{doc_title} — {section_heading}\n\n{body}"
```

That prefix is a few tokens and it fixes a common failure: a chunk reading "Enter the code and press confirm" is unfindable and useless without "Resetting your PIN" attached to it.

Keep, at minimum: source document id, title, section, page or position, and the date. The date matters because retrieval over a corpus containing three versions of a policy will happily return the 2019 one.

Tokens, not characters

A 1,500-character limit is roughly 375 English tokens and roughly 900 Hindi tokens on an English-centric tokenizer. Chunk in characters and your Hindi chunks silently exceed the model's window while your English ones sit comfortably inside it — a bug that makes a multilingual system worse in exactly the language it was added for. Count with the model's own tokenizer:

```
len(model.tokenizer.encode(chunk))
```

Retrieve small, give large

A technique worth knowing because it resolves the size trade-off. Embed and search over small chunks, then give the language model the **parent** section that chunk came from. You get the precision of a small chunk for matching and the completeness of a large one for answering. It costs one extra column in your metadata — the parent id — and it is usually the single best change after fixing chunk size.

Do this now

Take ten questions you would ask of your own documents. For each, find by hand the passage that answers it, and measure its length in tokens. That distribution is your chunk size, derived from your actual content rather than from a default.

BEFORE YOU MOVE ON

A bilingual system chunks at 1,500 characters. English retrieval is good and Hindi retrieval is poor, using the same multilingual model. What is the likely mechanism?

A. Hindi text produces far more tokens per character, so its chunks exceed the model's window and are truncated.

B. The multilingual model has weaker Hindi representations, so no chunking strategy would fix the gap.

C. Devanagari characters occupy multiple bytes, so character-based splitting cuts words mid-codepoint and corrupts the text.

D. Hindi documents are longer on average, so each one produces more chunks and dilutes the ranking.

53 · 9 min

NumPy first, FAISS second, a database rarely

THE ONE THING TO KEEP

Exact search with one matrix multiply handles a million vectors in tens of milliseconds, so NumPy is the right first index and FAISS or a vector database earns its place only at genuine scale or when filtered, updating, shared indexes are needed — and adding BM25 alongside fixes what embeddings are worst at.

The version that is enough for most projects

```
import numpy as np, json
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("BAAI/bge-small-en-v1.5")
V = model.encode(chunks, normalize_embeddings=True,
batch_size=64).astype("float32")
np.save("vectors.npy", V)
json.dump(meta, open("meta.json", "w"))          # same order
as V

def search(q, k=5):
    qv = model.encode(q, normalize_embeddings=True,
prompt_name="query")
    scores = V @ qv
    idx = np.argpartition(-scores, k)[:k]
    idx = idx[np.argsort(-scores[idx])]
    return [(float(scores[i]), meta[i]) for i in idx]
```

That is a complete semantic search engine in twelve lines, with no server and no dependency beyond NumPy. On 100,000 vectors of 384 dimensions it searches in a handful of milliseconds. On a million it is tens of milliseconds, which is still fine behind a web request.

`argpartition` rather than a full sort matters at scale: it finds the top k without ordering everything, and on a million items that is several times faster.

When NumPy stops being enough

Two thresholds, and they are further away than people assume:

- **Memory.** A million 384-dimensional float32 vectors is 1.5 GB. Ten million is 15 GB, and now you have a problem.
- **Latency.** Exact search is linear in corpus size. Beyond a few million, milliseconds become hundreds of milliseconds.

Below those, exact search is not only sufficient — it is *better*, because it is exact. Every approximate index trades recall for speed, and paying that price before you need it is a common and avoidable mistake.

FAISS, when you do cross the line

```
pip install faiss-cpu
```

```
import faiss

index = faiss.IndexFlatIP(V.shape[1])     # exact inner product
index.add(V)
D, I = index.search(qv.reshape(1, -1), 5)
```

`IndexFlatIP` is still exact — FAISS with a flat index is a faster matrix multiply, not an approximation, and it is a sensible first step. Approximation comes when you ask for it:

```
index = faiss.IndexHNSWFlat(V.shape[1], 32)   # graph index,
approximate
index.hnsw.efConstruction = 200
index.add(V)
index.hnsw.efSearch = 64                      # higher = better
recall, slower
```

HNSW builds a navigable graph and walks it. It is fast, it uses more memory than the raw vectors, and `efSearch` is the dial between recall and latency. **Measure the recall you are losing** by comparing against exact search on a few hundred queries — approximate indexes are routinely deployed with nobody knowing whether they return 99% or 80% of the right answers.

`IndexIVFPQ` compresses vectors as well, trading more accuracy for much less memory. It is what you reach for at tens of millions of vectors on one machine.

Vector databases, and when they are actually warranted

Qdrant, Weaviate, Milvus, Chroma, pgvector and the rest give you persistence, metadata filtering, incremental updates, concurrency and replication. Those are real features and none of them is about search quality.

Reach for one when you need:

- **Filtered search** — "nearest neighbours where `language = 'hi'` and `date > 2024`". Doing this properly is harder than it looks, because filtering after retrieval can leave you with two results out of five.
- **Frequent updates** — documents added and deleted continuously, without rebuilding.
- **More vectors than fit on one machine.**
- **Several services sharing one index.**

If you already run PostgreSQL, `pgvector` is usually the right answer: your metadata and your vectors live in one place, filtering is ordinary SQL, and you add no new operational surface. That is a genuinely free option and it is under-recommended because it is not exciting.

Keyword search is not obsolete

Embeddings are bad at exact identifiers: a part number, an error code, a surname. BM25 — classic keyword ranking — handles those perfectly and runs in-process:

```
pip install rank_bm25
```

Combining both and merging the ranked lists (reciprocal rank fusion is a four-line function) reliably beats either alone, typically by a wide margin on real corpora with codes and names in them. Hybrid retrieval is not an advanced technique; it is the default that most systems should start with.

Do this now

Build the NumPy index over your own documents and search it. Then add BM25 over the same chunks and compare the top five for three queries containing a specific code or name. The gap you see is the argument for hybrid search.

BEFORE YOU MOVE ON

A team replaces exact NumPy search over 80,000 chunks with an HNSW index and finds quality has dropped slightly, with no speed problem before or after. What went wrong in the decision?

A. HNSW requires unnormalised vectors, so the switch silently changed the similarity being computed.

B. The graph parameters were left at defaults, and tuning `efConstruction` would restore exact results.

C. They adopted an approximate index below the scale where exactness cost anything.

D. HNSW indexes must be rebuilt after every query batch, so the index had drifted out of date with the corpus.

54 · 8 min

The second pass that fixes the first

THE ONE THING TO KEEP

A bi-encoder compares precomputed vectors and scales, while a cross-encoder reads query and document together and judges whether one answers the other — so retrieving 50 cheaply and reranking them expensively buys most of the accuracy at a fraction of the cost.

Two architectures, and why both exist

A **bi-encoder** — everything so far — encodes the query and the document separately and compares vectors. The document's vector is computed once, in advance, so searching a million documents is a million dot products. Fast, and the comparison is crude, because the model never saw the query and the document together.

A **cross-encoder** takes the query and the document as one input and outputs a single relevance score:

```
from sentence_transformers import CrossEncoder

ce = CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")
print(ce.predict([("where is my parcel", "Your order shipped on
Tuesday."),
                  ("where is my parcel", "How to change your
password.")]))
# [ 8.21 -6.94 ]
```

Every query token can attend to every document token, so the model can tell that a document *answers* a question rather than merely being about the same topic. That is exactly the weakness the first lesson named.

The cost: nothing can be precomputed. Scoring one query against a million documents means a million forward passes. It is completely impractical as a

search method.

The pattern: retrieve then rerank

```
candidates = search(query, k=50)                    # bi-en-
coder, milliseconds
pairs = [(query, c["text"]) for c in candidates]
scores = ce.predict(pairs)                          # 50 for-
ward passes
best = [candidates[i] for i in scores.argsort()[::-1][:5]]
```

Fifty forward passes on a small cross-encoder is roughly 50–200 ms on a CPU and a few milliseconds on a GPU. You get most of the cross-encoder's accuracy at a tiny fraction of its cost, because the expensive model only ever sees fifty documents.

Where the time goes in one retrieval request

About a hundred and twenty milliseconds before any generation begins, and the rerank is four fifths of it. So that is the step to tune, by retrieving fewer candidates or moving the cross-encoder to a GPU. It is also where most of the accuracy comes from, which is why it is worth the time.

The gains are large and well replicated: on standard retrieval benchmarks, reranking the top 50 commonly improves precision at 5 by ten to twenty points over the bi-encoder alone. Very few single changes in this area do that.

Choosing how many to retrieve

The reranker can only reorder what it is given. Retrieve 10 and the right answer at rank 40 is lost forever; retrieve 200 and latency grows linearly.

The number to measure is your bi-encoder's **recall@k** — how often the correct document appears in the top k at all. Retrieve at the k where recall plateaus, typically 30 to 100. If recall@50 is 0.92 and recall@200 is 0.93, retrieving 200 buys you almost nothing and quadruples the rerank cost.

This is why the thirty-question set exists. Without it, both numbers are guesses.

Which reranker

- `cross-encoder/ms-marco-MiniLM-L-6-v2` — 90 MB, English, fast, the sensible default.
- `BAAI/bge-reranker-base` and `-large` — stronger, multilingual coverage including Chinese and several Indic languages.
- `jina-reranker` family — long inputs.

All run on CPU. A reranker is usually smaller than the generative model it feeds, and it is the cheapest quality improvement in a retrieval pipeline.

The language-model reranker, and its trade

You can also ask a general instruction model to rate relevance from 0 to 10. It works, needs no extra model if you already have one, and is slower, more expensive and less consistent than a trained cross-encoder — the same prompt on the same pair can score differently across runs unless you fix the sampling. Use it when adding a dependency is genuinely impossible; prefer the trained model otherwise.

What reranking cannot fix

- **A document that was never retrieved.** If chunking lost the fact, or the embedding model does not handle your language, no reranker helps. Fix retrieval first.
- **A document that does not exist.** The corpus is the ceiling.
- **Diversity.** A reranker will happily put five near-identical chunks at the top. Apply maximal marginal relevance after reranking, not before.

Latency, honestly

A full pipeline — embed query, search, rerank 50 — is roughly 5 ms to embed, 10 ms to search, 100 ms to rerank on CPU. Around 120 ms, before any genera-

tion. That is acceptable behind a web request and noticeable in a tight loop, and the rerank step dominates, so it is where you tune.

Do this now

Run your thirty questions with and without reranking and compare recall@1. This is normally the largest single improvement available in a retrieval system, and it takes twenty lines.

BEFORE YOU MOVE ON

Why can a cross-encoder distinguish "How do I cancel?" from "How do I renew?" when a bi-encoder cannot?

A. It is trained on relevance labels rather than similarity pairs, which teaches it to weight decisive words more heavily.
B. It processes query and document together, so attention can compare the decisive words directly.
C. It outputs an unbounded score rather than a bounded similarity, so small differences are not compressed near the top.
D. It encodes documents at query time, so the document vector can be conditioned on the query's length and language.

55 · 8 min

One space for pictures and words

THE ONE THING TO KEEP

CLIP-style models place images and their captions in one shared space, so photographs become searchable by typed sentences and classifiable with no training — while the same shape does not mean the same space, and mixing vectors from different models in one index produces confident nonsense.

CLIP, and why it is interesting

CLIP was trained on hundreds of millions of image–caption pairs with one objective: put an image and its caption close together, and push mismatched pairs apart. The result is a **shared** space where a photograph and the sentence describing it are neighbours.

```
from sentence_transformers import SentenceTransformer
from PIL import Image

m = SentenceTransformer("clip-ViT-B-32")
img = m.encode([Image.open("kitchen.jpg")], normalize_embed-
dings=True)
txt = m.encode(["a photo of a kitchen", "a photo of a
motorbike"],
               normalize_embeddings=True)
print(img @ txt.T)     # [[0.31 0.09]]
```

Text and image vectors are directly comparable, which gives you two capabilities that used to require training:

Search photos with words. Embed 50,000 images once, then search them with a typed sentence. No tags, no labels, no training. This runs on a laptop and it is the single most immediately useful thing in this lesson.

Zero-shot classification. Write the labels as sentences, embed them, and classify each image by which label sentence is nearest. Accuracy depends heavily on how you phrase the labels — "a photo of a cat" typically beats bare "cat", because the training captions were sentences. That sensitivity to phrasing is a real limitation and an honest part of the method.

The limits, stated plainly

- **It is not an OCR system.** CLIP reads text in images unreliably. Use a dedicated OCR model.
- **It has no fine spatial sense.** "A red cup to the left of a blue plate" retrieves images with red cups and blue plates in any arrangement.
- **It counts badly.** Three of something and five of something are barely distinguished.
- **It inherits the biases of internet captions**, including the associations between people's appearance and occupations that those captions carry. For anything involving photographs of people, this is not a footnote — it is a reason to evaluate carefully before deploying, and sometimes a reason not to.
- **The base model's resolution is low** — 224 pixels square. Fine detail is resized away before the model sees it.

Newer families — SigLIP, and the multilingual variants — improve accuracy and, importantly, extend the text side beyond English. `clip-ViT-B-32-multilingual-v1` handles many languages on the text side while sharing the same image space.

Audio embeddings

The same idea exists for sound. CLAP places audio clips and text descriptions in a shared space, so you can search a sound library for "dog barking" without tags. Speaker-embedding models such as the ECAPA and x-vector families place recordings of the same speaker close together, which is how diarisation and speaker verification work.

```
from transformers import AutoProcessor, ClapModel
```

The practical caution is the same as module five's: sampling rate must match what the model expects, and a mismatch degrades results with no error.

Combining modalities in one search

Because vectors from a shared-space model are comparable, one index can hold both. A product catalogue can be searched by photograph or by description against the same set of vectors, with no separate pipeline.

What you must not do is mix vectors from **different** models in one index. A 512-dimensional CLIP vector and a 512-dimensional text-embedding vector have the same shape and no relationship whatsoever; the arithmetic runs and returns confident nonsense. Store the model name and revision with every index, and refuse to load an index whose recorded model does not match the one you are querying with. That check is four lines and prevents a bug that is almost impossible to diagnose from its symptoms.

Document images

A newer approach embeds page images directly rather than extracting text first — useful for scanned documents, forms and anything where layout carries meaning that plain text loses. It is more expensive per page and it skips a fragile OCR step, which on poor scans is frequently a net gain. Worth knowing it exists when a text-extraction pipeline is producing rubbish.

Do this now

Embed a folder of your own photographs with CLIP and search them by typing a sentence. It takes fifteen lines and it is the demonstration that makes embeddings click for most people — because you did not label anything and it works.

BEFORE YOU MOVE ON

Two models both output 512-dimensional vectors. Storing both in one index and searching produces plausible but wrong results. Why does nothing fail loudly?

A. Dot products between equal-length vectors are always defined, so the arithmetic succeeds regardless of whether the spaces relate.
B. Normalisation maps every model's output onto the same unit sphere, which makes vectors from different models genuinely comparable.
C. Both models were trained on overlapping web data, so their spaces are approximately aligned and results degrade only slightly.
D. The index validates dimensionality at insert time, so a mismatch would have been rejected if the spaces were incompatible.

56 · 9 min

Thirty questions, and the numbers they give you

THE ONE THING TO KEEP

Thirty questions of your own with known correct chunks turn every dial in a retrieval system into a measurable decision, and the failures sort into a short list of distinct causes — but a set that size detects fifteen-point differences and not three-point ones, so it should settle big choices and never small ones.

Build the set first

Everything in this module has a dial on it — model, chunk size, overlap, k, reranker, hybrid weight. Without measurement you are turning dials and trusting your impression of five queries, which is how people end up confidently worse than when they started.

The set is thirty to fifty questions, written by you, each paired with the chunk or document that should answer it.

```
{"q": "How long does a refund take?", "gold": ["policy-refunds-c3"]}
{"q": "मेरा पार्सल कहाँ है?", "gold": ["delivery-tracking-c1", "delivery-faq-c2"]}
```

Three rules for writing them:

1. **Phrase them as users do**, not as the document does. A question that reuses the document's words tests nothing; keyword search would already have found it.
2. **Include the hard cases** — a misspelling, a question in the other language, an abbreviation, a question whose answer is spread across two chunks.
3. **Include five questions with no answer in the corpus.** Your system should return nothing useful for those, and measuring that is how you find out whether it invents an answer.

Store the file in the repository. It is an asset, and it outlives every model you will try.

What thirty questions can settle, and what they cannot

Detected reliably	Not detected: do not act on it
• A chunk size that is wrong for your documents	• Two points between two shortlisted models
• An embedding model that does not handle your language	• Fifty candidates against sixty
• A missing query prefix on a BGE or E5 family model	• An overlap of forty tokens against fifty
• A reranker's effect, which is usually ten points or more	
• A corpus that simply does not contain the answer	

Thirty is chosen because it is small enough to read every failure and large enough to detect the differences worth acting on. Treating twenty-six out of thirty against twenty-seven out of thirty as a result is how people talk themselves into noise.

The three numbers

Recall@k — the fraction of questions whose correct chunk appears in the top k. This measures the retriever alone, and it is the ceiling on everything downstream. If recall@50 is 0.7, no reranker and no language model can answer the other 30%.

Precision@k or hit rate@1 — how often the top result is right. This measures ranking, and it is what a reranker improves.

Mean reciprocal rank — the average of 1 divided by the rank of the first correct result. It rewards being right at position one rather than position three, in one number.

```
def evaluate(search_fn, questions, k=10):
    rec = mrr = 0
    for item in questions:
        got = [d["id"] for d in search_fn(item["q"], k=k)]
        hits = [i for i, d in enumerate(got) if d in
item["gold"]]
        if hits:
            rec += 1
            mrr += 1 / (hits[0] + 1)
    n = len(questions)
    return {"recall@k": rec / n, "mrr": round(mrr / n, 3)}
```

Read the failures, do not only count them

Thirty is small enough to read, and that is the point of choosing thirty rather than three thousand. Print every question that failed alongside what came back instead. The causes sort into a short list, and each has a different fix:

- **The gold chunk was never retrieved** → chunking or the embedding model.
- **It was retrieved at rank 20** → add or improve reranking.
- **A near-duplicate outranked it** → maximal marginal relevance, or deduplicate the corpus.
- **The query used a code or a name** → add BM25.

- **The answer spans two chunks** → larger chunks, more overlap, or parent retrieval.
- **The corpus does not contain the answer** → not a retrieval problem, and worth knowing before you spend a week on one.

That list is the actual value of the exercise. A number tells you whether something changed; the failures tell you what to change.

What thirty questions can and cannot detect

Thirty items will reliably detect a difference of fifteen or twenty points — a wrong chunk size, a model that does not handle your language, a missing query prefix. It will **not** detect a difference of two or three points; that is well inside sampling noise, and acting on it is superstition.

So: use thirty to make the big decisions, and do not chase small ones. If a difference matters and looks small, you need several hundred questions, and building those is a different project with a different budget.

Freeze the set, and add to it

Once a set exists, every change is checkable. Keep the original frozen so numbers stay comparable over time, and add a question every time production surprises you. An incident that produced a wrong answer becomes a permanent test item, and the set slowly becomes a record of everything you have learned about your own corpus.

The honest note

This measures retrieval, not answers. A system can retrieve the right chunk and still produce a wrong answer from it, and evaluating generated text is a harder problem with its own course. Measuring retrieval separately is worth doing precisely because it is tractable — when the whole system is wrong, the first question is always whether the right material was even found.

Do this now

Write ten questions in the next fifteen minutes. Not thirty, ten. Run them, read the failures, and fix the one that appears most. That loop, repeated, is the entire discipline.

BEFORE YOU MOVE ON

An evaluation shows recall@50 of 0.68 and hit rate@1 of 0.61. Which change is most likely to help?

A. Adding a cross-encoder reranker, since the gap between the two numbers shows poor ranking within the retrieved set.

B. Retrieving 200 candidates instead of 50, so that more of the corpus is considered before ranking.

C. Revisiting chunking and the embedding model, because a third of the answers are never retrieved at all.

D. Applying maximal marginal relevance, since near-duplicates are crowding out the correct chunk.

CASE STUDY · MODULE 6

The part number the search could not find

A ninety-person machine tools manufacturer in Rajkot with four thousand two hundred pages of service manuals, and a field team that looks things up by part number.

Nilesh, the service manager, wanted an engineer standing next to a broken machine to find the right procedure on a phone. A consultant had quoted for a hosted vector database with an annual licence at about a quarter of the department's software budget.

The manuals were PDFs, some of them scanned from the 1990s and some produced last year, covering fourteen machine families. The service team's working method was a WhatsApp group and a man called Ramesh who had been there twenty-two years and knew where everything was. Ramesh was retiring in eighteen months, which is what started the project.

Priyanka, a trainee on a six-month placement, built the cheap version first, because it was two days rather than a procurement cycle. She split the manuals into chunks on their headings, embedded them with a ninety-megabyte model, saved the vectors to a file, and searched them with one matrix multiply. Thirty-one thousand chunks, a few milliseconds a query, no server.

On questions phrased in words it worked well. How do I reset the servo after a power cut returned the right page.

On the queries the service team actually types, it failed. TX-4410 seal kit. Error E-207. The 2019 machine at Kulkarni's. An embedding model places a part number roughly nowhere in particular, because the number carries no meaning it was trained to place, and the vector for TX-4410 is not close to the page that contains TX-4410.

The decision

Nilesh had a quote in front of him and a trainee asking for two more days.

The quoted product genuinely offered things the file did not: metadata filtering, incremental updates, several services sharing one index, and somebody to telephone when it broke. None of those was the problem the service team had. But a trainee on a placement is a support contract with an end date, and Nilesh had been burned before by a system whose author left.

What settled it was thirty questions. Priyanka took them out of the search log, kept the phrasing exactly as engineers had typed it, and recorded for each one the page that should come back. Four configurations, one afternoon.

Embeddings alone: the right page in the top five for fifty-three per cent of questions. Keyword ranking alone: sixty-one per cent, strong on codes and weak on paraphrases. Both, with their ranked lists merged: eighty-four per cent. Both plus a small cross-encoder rescoring the top fifty results: eighty-eight per cent, and the proportion where the right page was first went from forty-one per cent to sixty-six.

Two extra dependencies, both free, both installed with one command. About a hundred and thirty milliseconds a query, of which the rescoring step was a hundred.

The thirty questions took Priyanka a morning and taught her something the measurements did not. Nearly half the queries in the log contained an identifier of some kind — a part number, an error code, a machine serial, or a customer's surname standing in for a machine. That distribution is a fact about the service desk rather than about retrieval, and it explains why the keyword method on its own beat the embeddings on its own, which is the reverse of what every tutorial had led her to expect.

She also checked one thing that is easy to skip. She printed the lengths of the vectors after encoding, to confirm they were all 1.0 and that the search was therefore comparing angles rather than being biased toward long documents. They were not normalised in her first version. Fixing that one argument moved recall by three points before any of the other work, and it is the kind of defect that otherwise shows up only as a system that is slightly worse than expected and is never diagnosed.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They did not buy the database. They did write down, in the repository beside the code, the two conditions under which they would: more vectors than fit in memory on a service laptop, noted as a few million rather than the thirty-one thousand they had, and a need for filtered search across product lines with different access rights. Both were written as numbers, so the decision could be revisited later without anybody having to win the argument again.

The largest single improvement came from the rescoring model, which is ninety megabytes and runs on an ordinary processor. The second largest came from adding keyword search back alongside the embeddings, which most people assume is the old thing they are replacing.

The file of thirty questions has since outlived two changes of embedding model. Priyanka's placement ended in March. The system was still running in December, and the person who took it over could tell whether their changes helped, because the questions were in the repository.

Why does an embedding model fail on TX-4410 while keyword ranking succeeds?

An embedding places text by meaning learned from training data, and an arbitrary part number has no meaning to place — its vector lands in an unhelpful part of the space and is not near the page that contains it. Keyword ranking matches the literal token, which is exactly right for identifiers, error codes and surnames. The two methods fail in different places, which is why merging their ranked lists beats either one and why hybrid retrieval is a sensible default rather than an advanced technique.

The database's features were real. Why were they not a reason to buy it?

Because none of them addressed the failure the team actually had, which was retrieval quality on identifier queries. Filtering, incremental updates and shared indexes are operational features, not search-quality features; buying them would have left the part-number problem exactly where it was, at a quarter of the budget. The conditions they wrote down are the ones under which those features start to matter, and at thirty-one thousand chunks none had been met.

They wrote their buying conditions as numbers. Why does that matter more than the decision itself?

Because it makes the decision re-openable without re-arguing it. A stated threshold — a few million vectors, or filtered access across product lines — can be checked against reality by anybody, including the person who inherits the system. A decision recorded as a preference has to be defended by the person who made it, and when that person's placement ends the reasoning leaves with them.

MODULE 6 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 A query about cancelling a subscription retrieves a passage about renewing one, near the top. What does that demonstrate?

A. The embedding model is defective and should be replaced with a larger one.

B. Similarity is topical closeness, not whether one text answers the other.

C. The chunks were too long, so the specific fact was diluted by the average.

D. Cosine similarity was used in a place where dot product was required.

2 Vector norms print as 8.3, 11.2 and 6.7, and dot-product search is slightly worse than expected. What is wrong?

A. The vectors were stored at float16 and have lost too much precision.

B. They are unnormalised, so dot product quietly favours longer vectors.

C. Euclidean distance is required whenever the vectors are not unit length.

D. The vectors are corrupt and the corpus has to be encoded again from source.

3 A BGE or E5 family model scores several points below its published numbers on your own set. What is the common cause?

A. The model needs trust_remote_code in order to load its pooling layer.

B. The published numbers were measured on a GPU, while you are running this on a CPU.

C. The instruction prefix the family was trained to expect on queries is missing.

D. The vectors were stored at float16 rather than at full float32 precision.

4 Chunks are cut at 1,500 characters. English chunks fit the model's window and Hindi chunks are silently truncated. Why?

A. Embedding models truncate non-Latin scripts more aggressively by design.

B. Devanagari is stored at two bytes per character, so the chunk doubles.

C. Overlap settings are applied to Latin scripts only, so the Hindi chunks run long.

D. Characters are not tokens, and Devanagari costs far more tokens per character.

5 Your bi-encoder's recall@50 is 0.92 and its recall@200 is 0.93. What follows for the reranking step?

A. Retrieve 50; the extra 150 quadruple the cost for one point of ceiling.

B. Retrieve 200, because a reranker always ranks better with more candidates.

C. Drop the reranker entirely, because recall is already high enough.

D. Retrieve 10, because a cross-encoder only ever reorders the very top.

6 You have 31,000 chunks of 384 dimensions and a web request to answer. What should the first index be?

A. An approximate graph index, to leave headroom on latency as you grow.

B. A hosted vector database, so that metadata filtering is available later on.

C. A compressed index, because 31,000 vectors will not fit in memory.

D. A NumPy array and one matrix multiply, which is exact and fast enough.

MODULE 7

Training and adapting, with the libraries the Hub is built on

Adapting a model on the Hub is a different job from training one from scratch, and the libraries reflect that: Trainer for the loop, PEFT for adapters that are a thousandth the size of the model, TRL for instruction and preference tuning, and the Hub itself as durable storage for a run that will be interrupted. This block starts by arguing you probably should not fine-tune, then teaches you to do it properly when you should.

> BY THE END OF THIS MODULE YOU CAN
>
> Decide from evidence whether a task needs fine-tuning at all, run a LoRA or QLoRA adaptation with Trainer or SFTTrainer on a single free GPU, read a loss curve well enough to name what is wrong, and publish an adapter a stranger can apply to the base model

57 · 8 min

Three cheaper things to try first

THE ONE THING TO KEEP

Fine-tuning teaches behaviour and retrieval supplies knowledge, so a gap made of facts should never be trained into weights — and prompting, a larger model and a small specialised classifier all sit between the problem and a training run that will need redoing when the base model is replaced.

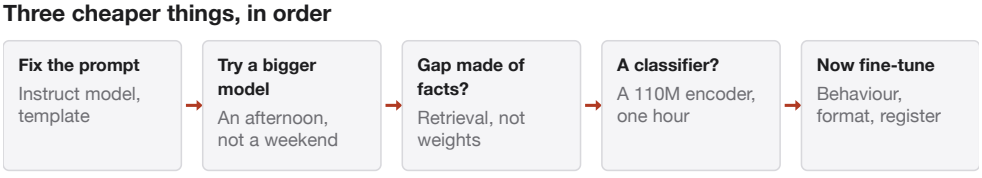
The question people skip

Fine-tuning is the most satisfying answer and usually not the first correct one. Before you spend a weekend on it, work through three cheaper options, in this order. Each of them fixes a different problem, and fine-tuning fixes only the third.

1. The prompt, and the right model

A surprising share of "the model cannot do this" turns out to be a base checkpoint used as a chat model, a missing chat template, or a prompt that never actually stated the constraint. Module two's checklist takes ten minutes and resolves this more often than anybody expects.

Then: few-shot examples. Three worked examples in the prompt frequently close most of the gap to a fine-tune for formatting and style tasks, cost nothing but context, and can be changed in a second rather than a weekend. If your task is "produce output shaped like this", try five examples before you try training.



Each step fixes a different problem and fine-tuning fixes only the last. The distinction worth memorising sits in the middle: fine-tuning teaches behaviour and retrieval supplies knowledge, so a gap made of facts should never be trained into weights.

And try a bigger model. An 8B model with a good prompt regularly beats a 1B model you have fine-tuned, at a fraction of the effort. Fine-tuning a small model to catch up with a large one is real work; renting the large one for a week to find out whether the task is even solvable is an afternoon.

2. Retrieval

If the gap is **knowledge** — your product's documentation, your policies, this year's prices — fine-tuning is the wrong tool and everybody learns this the hard way.

Training facts into weights is inefficient, unreliable and impossible to update. The model will produce fluent, confident, subtly wrong versions of the facts you trained on, because it learned the *shape* of your documents rather than their content. Retrieval puts the actual text in front of the model at the time it answers, it can be updated by adding a file, and it can cite its source.

The distinction worth memorising: **fine-tuning teaches behaviour, retrieval supplies knowledge.** A model that writes in your house style is fine-tuning. A model that knows your refund policy is retrieval. Most people arrive wanting the second and reach for the first.

3. A smaller, specialised model

For classification, extraction and tagging, fine-tuning a 110M encoder on a few thousand labelled examples is cheap, fast and usually more accurate than prompting a large generative model. That *is* fine-tuning, and it is the easy kind: an hour on a free GPU, a model that runs on a CPU afterwards, and a metric you can measure exactly.

When people say fine-tuning is hard, they mean adapting large generative models. Fine-tuning a small classifier has been routine since 2019 and remains the correct answer to a large class of problems.

When fine-tuning is genuinely the answer

- **A format or style you cannot express in a prompt**, or can express but not reliably enough.
- **A domain vocabulary** the model handles poorly — legal, clinical, a specific industry's shorthand.
- **A language or dialect** the base model is weak in.
- **Cost**: a fine-tuned 1B model matching a prompted 70B model on one narrow task is a very large saving if you run it a million times.
- **Latency**: a shorter prompt, because the instructions are in the weights.
- **Privacy**: the task must run on your own hardware, and only a small model fits.

Notice what these have in common. They are all about **how** the model behaves, not **what** it knows.

What it costs, honestly

- **Data.** A few hundred high-quality examples for style, a few thousand for a real capability. Making them is most of the work and there is no way around it.
- **Compute.** A LoRA on a 7B model over a few thousand examples is one to four hours on a free Colab or Kaggle GPU. A few dollars if rented. Cheaper than people think.
- **Evaluation.** You need a held-out set before you start, or you cannot tell whether it worked.
- **Maintenance.** A fine-tune is pinned to its base model. When a better base appears in four months, you redo it. This is the cost nobody budgets.

The decision, in one paragraph

Try the prompt. Try a bigger model. If the gap is knowledge, build retrieval. If the task is classification, fine-tune a small encoder. If the gap is behaviour on a generative task and the first three did not close it, and you have or can make several hundred good examples, then fine-tune — and the rest of this module is how.

Do this now

Write down what you want the model to do differently, then label it: knowledge, behaviour, or format. If it is knowledge, close this module and go back to the retrieval one. That single sentence saves more time than any technique here.

BEFORE YOU MOVE ON

A team fine-tunes a 7B model on their product documentation so it can answer customer questions. It becomes fluent about the product and gets specific details wrong. Why?

A. The learning rate was too low to memorise facts, so more epochs would fix the detail errors.

B. Training taught the shape of the documents rather than facts, which retrieval supplies directly.

C. The documentation was too small a dataset, and tens of thousands of examples would have made the facts stick.

D. The base model's pretraining contradicts the documentation, so the two sources of knowledge interfere with one another.

58 · 9 min

Trainer, and what it saves you writing

THE ONE THING TO KEEP

`Trainer` implements the loop, scheduling, mixed precision, checkpointing and resumption that are individually easy and collectively error-prone, and the two arguments that most often decide whether a run is usable are `id2label` at construction and `resume_from_checkpoint` on restart.

The whole thing, for a classifier

```
import numpy as np, evaluate
from transformers import (AutoModelForSequenceClassification,
AutoTokenizer,
                          TrainingArguments, Trainer,
DataCollatorWithPadding)

tok = AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = AutoModelForSequenceClassification.from_pretrained(
    "distilbert-base-uncased", num_labels=3,
    id2label={0: "neg", 1: "neu", 2: "pos"},
    label2id={"neg": 0, "neu": 1, "pos": 2},
)

ds = ds.map(lambda b: tok(b["text"], truncation=True,
max_length=256), batched=True)

f1 = evaluate.load("f1")
def metrics(p):
    preds = np.argmax(p.predictions, axis=1)
    return f1.compute(predictions=preds, references=p.la-
bel_ids, average="macro")

args = TrainingArguments(
    output_dir="out",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    num_train_epochs=3,
```

```
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
    metric_for_best_model="f1",
    push_to_hub=True,
    hub_model_id="your-name/reviews-classifier",
)

trainer = Trainer(
    model=model, args=args,
    train_dataset=ds["train"], eval_dataset=ds["validation"],
    data_collator=DataCollatorWithPadding(tok),
    compute_metrics=metrics,
)
trainer.train()
trainer.push_to_hub()
```

That is a complete, publishable fine-tune. Note `id2label` passed at construction — the thing whose absence produces `LABEL_0` outputs forever, and the cheapest fix in this course.

What Trainer is doing for you

It writes, correctly, the loop you would otherwise get subtly wrong: batching and shuffling, moving tensors to the device, the forward and backward passes, gradient accumulation, learning-rate scheduling and warmup, gradient clipping, mixed precision, evaluation at intervals, checkpoint saving and rotation, resumption, logging, and multi-GPU distribution through accelerate.

None of those is hard alone. Getting all of them right together, and keeping them right, is why nobody writes their own loop for ordinary work.

Metrics, and `evaluate`

```
pip install evaluate
```

```
acc = evaluate.load("accuracy")
f1  = evaluate.load("f1")
wer = evaluate.load("wer")          # speech
rouge = evaluate.load("rouge")      # summarisation
```

`evaluate` loads standard metric implementations from the Hub, which means everybody computing F1 is computing the same F1. That matters more than it sounds: `average="macro"` and `average="micro"` differ enormously on imbalanced data, and two teams quoting "F1" without saying which have not compared anything.

For a classifier on imbalanced classes, macro-F1 is usually the honest headline, with per-class precision and recall printed beside it.

Resuming, which you will need

```
trainer.train(resume_from_checkpoint=True)
```

On free hardware the session will end. With `save_strategy="epoch"` or `save_steps=200` and this one argument, a disconnection costs minutes rather than hours. Combine it with `push_to_hub=True` and checkpoints leave the machine as well, which is the module-three discipline applied to training.

`save_total_limit=2` stops checkpoints filling the disk — each one is a full copy of the model, and three copies of a 7B model is 42 GB.

The evaluation loop, and its cost

`eval_strategy="steps"` with `eval_steps=100` runs the whole validation set every hundred steps. On a large validation set that can take longer than the training between evaluations, and people then conclude that training is slow when most of the wall-clock time is measurement.

Two fixes. Keep the validation set small — a thousand rows is plenty for watching a curve, and the real evaluation happens once at the end on a larger held-out set. And set `per_device_eval_batch_size` explicitly, because it in-

herits the training batch size by default and evaluation has different memory behaviour.

```
eval_dataset=ds["validation"].select(range(1000)),
per_device_eval_batch_size=32,
```

Seq2seq and generation

```
from transformers import Seq2SeqTrainer,
Seq2SeqTrainingArguments

args = Seq2SeqTrainingArguments(...,
predict_with_generate=True, generation_max_length=128)
```

`predict_with_generate=True` makes evaluation actually generate text rather than scoring forced labels, which is what you need for ROUGE or word error rate. Without it your evaluation metric measures something other than what the model will do at inference, and looks better than reality.

When to write your own loop

When you need a custom loss, an unusual optimisation schedule, or a training procedure with several models in it — reinforcement learning, adversarial setups, some distillation schemes. Subclassing `Trainer` and overriding `compute_loss` covers most of these and keeps everything else:

```
class WeightedTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False,
**kw):
        labels = inputs.pop("labels")
        out = model(**inputs)
        loss = torch.nn.functional.cross_entropy(
            out.logits, labels, weight=class_weights.to(out-
.logits.device))
        return (loss, out) if return_outputs else loss
```

That is class weighting for imbalanced data, which the data-quality lesson recommended, in eight lines.

Do this now

Train a classifier on one per cent of a dataset for one epoch. It will be bad, and that is not the point: the point is that the whole loop — data, metrics, checkpoints, push — runs end to end in a few minutes. Get the plumbing right on something worthless before you spend hours on something you care about.

BEFORE YOU MOVE ON

A summarisation model reports strong ROUGE during training and produces poor summaries at inference. What is the most likely cause?

A. The evaluation set overlaps the training set, so the reported score reflects memorisation.

B. `predict_with_generate` was left off, so evaluation scored forced reference tokens.

C. ROUGE is unsuitable for summarisation and should be replaced with a model-based metric.

D. The generation config at inference uses different sampling parameters from those used during evaluation.

59 · 9 min

The six arguments worth understanding

THE ONE THING TO KEEP

Learning rate sets the scale of everything and differs tenfold between full fine-tuning and LoRA, while gradient accumulation buys a large effective batch on small hardware and `gradient_checkpointing`, 8-bit optimisers and shorter sequences are the ordered responses to running out of memory.

`TrainingArguments` has over a hundred fields

Six of them decide almost everything. The rest have sensible defaults and you can leave them alone until a specific problem says otherwise.

1. Learning rate

The most important number in the file, and the one wrong by an order of magnitude most often.

- **Full fine-tuning of a pretrained model**: `2e-5` to `5e-5`. Higher and you destroy what pretraining learned — the loss falls fast and the model gets worse at everything it used to do.
- **LoRA and adapters**: `1e-4` to `3e-4`, ten times higher, because you are training a small number of fresh parameters rather than nudging trained ones.
- **A newly initialised head only**: `1e-3` is reasonable.

If the loss explodes to NaN, the learning rate is too high. If the loss barely moves after a full epoch, it is too low. Those two symptoms are unambiguous and they are the fastest diagnostic you have.

2. Batch size, and the way to fake a big one

```
per_device_train_batch_size=4,
gradient_accumulation_steps=8,      # effective batch size = 32
```

Gradient accumulation runs four examples, keeps the gradients, runs four more, and updates after eight rounds. Mathematically close to a batch of 32, with the memory of 4. This is how a 16 GB free GPU trains with batch sizes that would otherwise need 80 GB, and it is the single most useful trick in the file.

It costs wall-clock time, not quality. Effective batch size is what matters for stability, and it is the product of the two numbers times the number of devices.

3. Epochs, and when to stop

Two to three epochs for most fine-tuning. More than that on a small dataset and the model memorises: training loss keeps falling, validation loss turns upward. That divergence is overfitting, and it is visible in the logged numbers long before it is visible in the output.

```
eval_strategy="steps", eval_steps=100,
load_best_model_at_end=True,
metric_for_best_model="eval_loss", greater_is_better=False,
```

With those set, you keep the best checkpoint rather than the last one, which is not the same thing and is frequently better.

4. Precision

```
bf16=True,      # Ampere-generation GPUs and newer
fp16=True,      # older cards such as the free-tier T4
```

Roughly twice as fast and half the memory. `bf16` where supported, for the exponent-range reason from module three. Never both.

5. Memory arguments, in the order to reach for them

When you hit out-of-memory, work down this list:

1. `per_device_train_batch_size` down, `gradient_accumulation_steps` up. Free, no quality cost.
2. `gradient_checkpointing=True`. Recomputes activations in the backward pass instead of storing them. Cuts activation memory substantially and costs roughly 20–30% speed. Almost always worth it.
3. `optim="adamw_bnb_8bit"`. Adam keeps two state tensors per parameter; 8-bit quantizes them, saving about three quarters of that. For a 7B model that is several gigabytes.
4. Shorter `max_length`. Attention memory grows with sequence length; halving it helps a lot.
5. LoRA instead of full fine-tuning, which is the next lesson and dwarfs all of the above.

6. Warmup and schedule

```
warmup_ratio=0.03,
lr_scheduler_type="cosine",
```

Warmup ramps the learning rate from zero over the first few per cent of steps. Without it, the first large update to a pretrained model can do damage that later training never recovers from. Cosine decay then brings the rate down smoothly toward the end. These two are close to free and close to always correct.

Length-grouped batching, for free speed

```
group_by_length=True,
```

Batches items of similar length together, so padding is minimal. On variable-length text this frequently cuts training time by a third or more, for one argument and no quality effect. It is the same insight as module two's sorting trick.

A working starting point

```
TrainingArguments(
    output_dir="out",
    learning_rate=2e-4,                  # LoRA
    per_device_train_batch_size=4,
    gradient_accumulation_steps=8,
    num_train_epochs=3,
    bf16=True,
    gradient_checkpointing=True,
    group_by_length=True,
    warmup_ratio=0.03,
    lr_scheduler_type="cosine",
    logging_steps=10,
    eval_strategy="steps", eval_steps=100,
    save_steps=100, save_total_limit=2,
    load_best_model_at_end=True,
    report_to="tensorboard",
    seed=42,
)
```

Change the learning rate for full fine-tuning and leave the rest until something goes wrong.

Do this now

Run the same tiny training twice with learning rates ten times apart and plot both loss curves. Too-high and too-low have distinct shapes, and recognising them on sight is worth more than any table of recommended values.

BEFORE YOU MOVE ON

Training a 7B model with LoRA runs out of memory at batch size 4. Which change reduces memory without changing the effective batch size or the optimisation?

A. Halving the learning rate, which reduces the magnitude of the stored gradients.
B. Dropping to batch size 2 and doubling gradient accumulation steps.
C. Switching from bf16 to fp16, which uses less memory per activation on most hardware.
D. Reducing the number of epochs, so fewer checkpoints are held in memory during the run.

60 · 9 min

LoRA: training 0.1% of the parameters

THE ONE THING TO KEEP

LoRA freezes the base weights and learns a low-rank update, so optimiser state shrinks by the same hundredfold factor and the result is a few megabytes tied to one exact base model — which is what makes fine-tuning a 7B model on free hardware possible at all.

The observation it rests on

Full fine-tuning updates every weight. For a 7B model that means holding the weights, their gradients, and two optimiser states each — roughly four times the weight memory, so about 56 GB at bfloat16 before activations. No free GPU comes close.

LoRA starts from an empirical observation: the *change* fine-tuning makes to a weight matrix tends to be low-rank — expressible as the product of two much

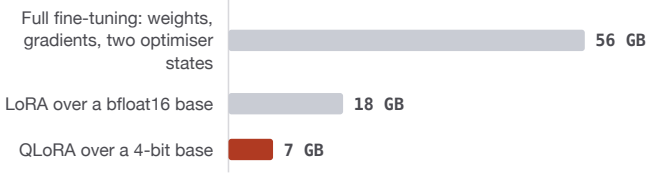
smaller matrices. So freeze the original weight `W` and learn a small update instead:

> *output = W·x + B·A·x, where A is r×d and B is d×r*

With `r=16` on a 4096×4096 matrix, A and B together hold about 131,000 numbers against the original's 16.8 million — under one per cent. Across a whole model you typically train 0.1% to 1% of the parameters.

The consequences are what make it matter:

Training a 7B model: where the memory actually goes



The weights are close to the same size in the first two rows. What collapses is the optimiser state, which shrinks by the same hundredfold factor as the trainable parameter count. The third row quantizes the frozen base as well, which is what puts a seven-billion-parameter fine-tune on a free Colab card.

- **Optimiser state shrinks by the same factor**, which is where the memory actually went.
- **The result is a few megabytes**, not gigabytes. You can publish dozens of adapters for one base model.
- **The base is untouched**, so one loaded base model can serve several adapters, swapped per request.
- **Quality is usually close to full fine-tuning** on adaptation tasks, and the published comparisons bear this out — though not universally, and not for teaching genuinely new capabilities.

In code

```
pip install peft
```

```
from peft import LoraConfig, get_peft_model, TaskType

cfg = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj",
                    "gate_proj", "up_proj", "down_proj"],
    task_type=TaskType.CAUSAL_LM,
)

model = get_peft_model(base_model, cfg)
model.print_trainable_parameters()
# trainable params: 41,943,040 || all params: 7,290,000,000 ||
trainable%: 0.575
```

Then hand it to `Trainer` exactly as before. Nothing else in the loop changes.

The three settings that matter

`r` — the rank. 8 or 16 for style and format work; 32 to 64 when teaching a genuinely new capability or a new domain. Higher `r` means more capacity and more memory, and past a point it stops helping. Start at 16.

`lora_alpha` — a scaling factor; the update is multiplied by `alpha/r`. The common convention is `alpha = 2r`, so `r=16, alpha=32`. Changing `r` without changing `alpha` silently changes the effective learning rate, which is a confusing way to get inconsistent results.

`target_modules` — which matrices get adapters. Attention projections only (`q_proj`, `v_proj`) is the minimal, cheapest setting. Including the feed-forward projections as well is the modern default and measurably better on harder tasks. You can pass `"all-linear"` to let PEFT find them, which is convenient and occasionally adapts something you did not intend.

The adapter is a separate thing

```
model.save_pretrained("my-adapter")      # a few MB
```

That directory holds `adapter_config.json` — which names the base model — and `adapter_model.safetensors`. Loading it:

```
from peft import PeftModel

base = AutoModelForCausalLM.from_pretrained(base_id,
dtype=torch.bfloat16,
                                            device_map="auto")
model = PeftModel.from_pretrained(base, "your-name/my-adapter")
```

The adapter is useless without its exact base, which is why `adapter_config.json` records it and why the module-one lesson warned about downloading an adapter and wondering why nothing loads.

For deployment you can fold it in permanently:

```
merged = model.merge_and_unload()
merged.save_pretrained("merged-model")
```

Merging removes the small inference overhead of the extra matrices and gives an ordinary model that any serving stack loads. It also gives up the ability to swap adapters, and it produces a full-size model you now have to store.

What LoRA is not good at

It adapts what is there. Teaching a model a language it has essentially never seen, or a genuinely new capability, is where full fine-tuning or continued pretraining still wins — and where a higher rank is the first thing to try before giving up on adapters.

There are also variants worth knowing exist: DoRA separates magnitude from direction and often does slightly better at the same rank; prefix and prompt tuning train even fewer parameters with less capability. LoRA remains the default because it is well understood, well supported, and good enough.

Do this now

Run `print_trainable_parameters()` on a LoRA-wrapped model and read the percentage out loud. Then compute what full fine-tuning would have needed in optimiser memory. That ratio is the entire reason this technique exists.

BEFORE YOU MOVE ON

Why does LoRA reduce training memory so much more than it reduces the number of weights involved in the forward pass?

A. Frozen base weights are offloaded to CPU during training, so only the adapter matrices stay resident.

B. Optimiser state and gradients are held only for trainable parameters, and those are a fraction of a per cent.

C. The low-rank decomposition reduces the arithmetic in each layer, which shrinks the activations that must be stored.

D. Adapters are stored in 8-bit while base weights stay in bfloat16, halving the memory for the trained part.

61 · 8 min

QLoRA, and fine-tuning a 7B model on a free GPU

THE ONE THING TO KEEP

QLoRA keeps a frozen base in 4-bit and trains precise adapters over it, de-quantizing per matrix multiply — so a 7B model fits a free GPU at the cost of speed rather than memory, and `prepare_model_for_kbit_training` is the line whose absence makes training quietly fail instead of raising.

Combining two ideas

LoRA removed the optimiser memory. The weights themselves are still there — 14 GB for a 7B model at bfloat16, which does not leave room on a 16 GB card for activations and a KV cache.

QLoRA loads the **base model in 4-bit** and trains LoRA adapters on top of it in higher precision. The frozen base is quantized because nothing updates it; the trainable part stays precise because precision is what it needs. A 7B model then trains in roughly 6–8 GB, which fits a free Colab T4 with room to spare.

```
import torch
from transformers import BitsAndBytesConfig,
AutoModelForCausalLM
from peft import LoraConfig, get_peft_model, prepare_model_-
for_kbit_training

bnb = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True,
)

base = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2.5-7B-Instruct", quantization_config=bnb, de-
vice_map="auto"
)
base = prepare_model_for_kbit_training(base, use_gradien-
t_checkpointing=True)
model = get_peft_model(base, LoraConfig(r=16, lora_alpha=32,
task_type="CAUSAL_LM",
                                        target_modules="all-
linear"))
```

What each setting is doing

`nf4` is a 4-bit data type designed for weights that are roughly normally distributed, which neural network weights are. It preserves more information than plain 4-bit integers at the same size.

`bnb_4bit_compute_dtype=torch.bfloat16` means weights are *stored* in 4 bits and *computed* in bfloat16 — dequantized on the fly for each matrix multiply. This is why QLoRA is slower than bfloat16 training per step even though it uses far less memory. You are trading time for space, deliberately.

`use_double_quant` quantizes the quantization constants themselves, saving a further few hundred megabytes on a 7B model.

`prepare_model_for_kbit_training` casts layer norms to float32, enables gradient checkpointing and makes the input embeddings require gradients.

Skip it and training is unstable or silently fails to learn — one of those lines whose absence produces a confusing result rather than an error.

What it costs in quality

The published QLoRA results, and most community replications, find results close to 16-bit LoRA on adaptation tasks. "Close" is doing work in that sentence: on tasks where the quantization damage falls — arithmetic, exact structured output, rarely seen languages — the gap is larger, for the mechanism in module three.

The practical stance: use QLoRA when the alternative is not training at all, which for most people on free hardware it is. If you have a card that fits the model at bfloat16, use bfloat16 — it is faster and marginally better.

Merging a QLoRA adapter, and the subtlety

You trained against 4-bit weights. Merging the adapter into the **full-precision** base is standard practice and is not exactly the arithmetic you trained: the adapter learned to correct a quantized base, and it is being applied to an unquantized one. In practice this works and is what almost everybody does, and it is worth evaluating the merged model rather than assuming the numbers carry over.

The cleaner path for serving is to keep the adapter separate and load it over whichever base precision you serve with. It also lets you swap adapters, which merging forfeits.

A realistic budget

On a free Colab T4, a 7B model with QLoRA, 2,000 examples of about 512 tokens, three epochs, batch size 1 with accumulation to 16: roughly two to four hours. On Kaggle's longer sessions, comfortable. Rented on a 24 GB card at community marketplace rates, a few dollars.

The binding constraint is usually the session length rather than the money, which brings back the module-three discipline: `save_steps`, `push_to_hub`, `resume_from_checkpoint`.

The honest alternative

If you want the result rather than the understanding, AutoTrain on the Hub and libraries like Unsloth and Axolotl wrap all of this behind configuration. Unsloth in particular reports substantial speed and memory improvements through custom kernels, and it is free and open source for single-GPU use. Using them is not cheating. Understanding what they are doing is what this lesson is for, because when a run fails you will need it.

Do this now

Load a 7B model with the QLoRA configuration above on a free GPU and print `torch.cuda.max_memory_allocated()` after one training step. Compare that against the 56 GB full fine-tuning would have needed. That single comparison is the reason this technique changed who can do this work.

BEFORE YOU MOVE ON

Why is QLoRA slower per training step than 16-bit LoRA despite using far less memory?

A. Gradient checkpointing is mandatory with 4-bit bases, and recomputing activations dominates the step time.

B. Quantized weights must be dequantized to the compute dtype for every matrix multiply.

C. 4-bit arithmetic is emulated in software on most GPUs, so each operation takes several native instructions.

D. Smaller memory footprints reduce the achievable batch size, so more steps are needed for the same data.

62 · 9 min

TRL, and training on conversations

THE ONE THING TO KEEP

`SFTTrainer` renders chat templates, masks the prompt, packs and truncates conversations for you, so the remaining decisions are about data — a few hundred diverse, well-written examples in the messages format beat tens of thousands of scraped ones, and the register of your responses becomes the model's register.

Why a separate library

`Trainer` trains on tensors. Instruction data is conversations, which have to be templated, masked so loss falls only on the assistant's turns, packed sensibly and truncated without cutting a turn in half. Module five did that by hand once, which is the right way to understand it and the wrong way to do it every time.

TRL's `SFTTrainer` does all of it from a dataset of messages.

```
pip install trl
```

```
from trl import SFTTrainer, SFTConfig

# each row: {"messages": [{"role": "user", ...}, {"role": "as-
sistant", ...}]}
trainer = SFTTrainer(
    model=model,
    train_dataset=ds["train"],
    eval_dataset=ds["validation"],
    peft_config=lora_cfg,
    args=SFTConfig(
        output_dir="out",
        max_length=1024,
        packing=True,
        learning_rate=2e-4,
        num_train_epochs=3,
        per_device_train_batch_size=4,
        gradient_accumulation_steps=4,
        bf16=True,
    ),
)
trainer.train()
```

`SFTConfig` extends `TrainingArguments`, so everything from the previous lesson still applies. `peft_config` wires LoRA in without a separate `get_peft_model` call.

The data format

The conversational format is a `messages` list per row, and it is the one to use because the tokenizer's own chat template renders it:

```
{"messages": [
  {"role": "system", "content": "You answer in Hindi."},
  {"role": "user", "content": "पार्सल कब आएगा?"},
  {"role": "assistant", "content": "आपका पार्सल मंगलवार तक पहुँच
जाएगा।"}
]}
```

Storing messages rather than rendered prompts is the same rule as module two, and here it pays twice: the same dataset trains any model family, because

rendering happens at training time with that model's template.

What makes instruction data good

This matters more than any hyperparameter, and it is where people underinvest.

- **Quality over quantity.** A thousand carefully written examples beat fifty thousand scraped ones. Published work on instruction tuning has repeatedly found small curated sets competitive with much larger noisy ones.
- **Diversity of instruction, not of topic.** Twenty ways of asking for the same thing teaches robustness; twenty topics with one phrasing each does not.
- **Responses in the register you want.** The model copies length, formality and structure. If your examples are three paragraphs, expect three paragraphs forever.
- **Include refusals and edge cases.** If every example has a confident answer, the model learns that confident answers are always available.
- **Check for leakage** between your training conversations and your evaluation ones. Module five's check applies unchanged.

Packing, and its caveat

`packing=True` concatenates short conversations into full-length blocks so no compute is spent on padding. On a dataset of short exchanges this is a large speed-up.

The caveat: packed examples share a sequence, so without attention masking between them one conversation can attend to the previous one. TRL handles this correctly for supported models; a hand-rolled implementation usually does not, which is why doing it yourself is a good exercise and a bad production choice.

Preference tuning, briefly

After supervised tuning, preference methods adjust a model toward chosen responses and away from rejected ones. `DPOTrainer` implements direct preference optimisation, which needs no separate reward model:

```
# each row: {"prompt": ..., "chosen": ..., "rejected": ...}
from trl import DPOTrainer, DPOConfig
```

This is what turns a model that *can* answer into one that answers the way people prefer. It needs preference pairs, which are more expensive to collect than instructions, and it is a second stage rather than a replacement for the first. TRL also implements GRPO and related methods for training against a programmatic reward — useful when correctness is checkable, such as code that must compile or arithmetic that must be right.

For most people reading this, supervised tuning is enough and preference tuning is a later problem. Knowing the sequence — pretrain, instruction tune, preference tune — explains why the models you download behave as they do.

Before you launch a long run

```
b = next(iter(trainer.get_train_dataloader()))
print(tok.decode(b["input_ids"][0]))
print(tok.decode([t for t in b["labels"][0] if t != -100]))
```

Read both. The first is what the model sees; the second is what it is graded on. If the second contains the user's words, your masking is wrong and three hours are about to be wasted.

Do this now

Build fifty instruction examples by hand for a task you care about, in the messages format, and run one epoch. Fifty is too few to work and enough to prove the pipeline. Then spend the time you saved on making two hundred good examples rather than two thousand mediocre ones.

BEFORE YOU MOVE ON

A team fine-tunes on 40,000 scraped instruction pairs and gets worse results than a colleague using 800 hand-written ones. What best explains it?

A. The larger dataset needed a lower learning rate, and the run diverged before converging properly.
B. 40,000 examples require more epochs to see each example enough times, so the run was undertrained.
C. The model copies the register and quality of its examples, so noisy scraped responses teach noisy responses.
D. Packing was enabled on the larger set, so unrelated conversations attended to each other and corrupted training.

63 · 8 min

Watching a run you cannot sit next to

THE ONE THING TO KEEP

Log training loss, validation metric, learning rate and gradient norm somewhere that outlives the machine, checkpoint with `hub_strategy="checkpoint"` so a killed session costs minutes, and resume with `resume_from_checkpoint` because reloading only the weights restarts the learning-rate schedule.

What to log, and why the default is not enough

A training run prints a loss every `logging_steps`. That one number tells you whether the run is alive and almost nothing else. Four things are worth having:

1. **Training loss** — is it falling.
2. **Validation loss or metric** — is it *generalising*, which is a different question.

3. **Learning rate** — did warmup and decay behave as configured.
4. **Gradient norm** — spikes here precede a divergence you can otherwise only see after it has happened.

All four are logged automatically with `report_to` set:

```
TrainingArguments(
    ...,
    logging_steps=10,
    eval_strategy="steps", eval_steps=100,
    report_to="tensorboard",      # or "trackio", "wandb",
"none"
)
```

Where to put the logs

TensorBoard writes event files into `output_dir`. If `push_to_hub=True` is set, those files go to the model repository and the Hub renders them on a Training metrics tab — so a stranger reading your model card can see the loss curve. Free, no account, no service.

Trackio is a lightweight open-source tracker that runs as a Space, storing runs in a dataset repository. It is free, self-hosted and the API mirrors the commercial trackers closely enough that switching is a one-line change.

Weights & Biases and similar services are free for personal use and excellent, with the ordinary caveat that your run metadata goes to a third party. On a work project that may need asking about first.

The important thing is that **the record survives the machine**. A loss curve that existed only in a Colab output cell is gone when the session ends, and with it the ability to compare this run to the next one.

Reading a loss curve

Four shapes, and what each means:

- **Both losses fall and level off.** Working. Stop when validation stops improving.

- **Training falls, validation rises.** Overfitting. Fewer epochs, more data, more dropout, or a lower LoRA rank. `load_best_model_at_end` keeps the checkpoint from before the turn.
- **Both flat.** Learning rate too low, or the labels are not reaching the loss — check the masking. A completely flat loss from step one is nearly always a data bug, not a learning-rate problem.
- **Loss spikes to NaN.** Learning rate too high, or float16 overflow. Lower the rate; switch to bf16.

A jagged curve is normal at small batch sizes. A curve that is jagged *and* trending flat usually means the effective batch is too small — raise gradient accumulation before you change anything else.

Checkpoints, and how not to lose a night

```
save_strategy="steps", save_steps=100,
save_total_limit=2,
load_best_model_at_end=True,
push_to_hub=True, hub_strategy="checkpoint",
```

`hub_strategy="checkpoint"` pushes the checkpoint to the Hub as it saves, so the run survives the machine dying. `save_total_limit=2` keeps the disk from filling — each checkpoint is a full copy of the trainable state, and on a free instance three of them can be the thing that kills the session.

Resuming is one argument:

```
trainer.train(resume_from_checkpoint=True)
```

This restores optimiser state and the scheduler position as well as weights. Restarting from a checkpoint by loading only the weights and beginning again resets the learning-rate schedule, which produces a visible jump in the loss and a worse model. Use the argument.

Seeds and what reproducibility means here

```
seed=42, data_seed=42
```

Those make the run repeatable on the same hardware with the same library versions. Across different GPUs or versions, floating-point accumulation order differs and results drift slightly. That is expected; claiming bit-identical reproducibility across machines is a claim nobody can keep.

What you *can* keep: log the versions, the seed, the base model revision and the dataset revision into the run. Then a result is reconstructible even when it is not bit-identical, which is what anybody actually needs.

Evaluate the model, not the loss

Validation loss going down is necessary and not sufficient. Before declaring a run successful, generate output for twenty held-out inputs and read them. Loss can improve while the model develops a habit you would immediately reject — always answering in English, always producing three bullet points, losing the ability to say it does not know.

This is twenty minutes and it is the only check that catches that class of problem.

Do this now

Set `report_to="tensorboard"` and `push_to_hub=True` on your next run, then open the Training metrics tab on the resulting repository. A loss curve attached to the model it produced is a small thing that makes every later comparison possible.

BEFORE YOU MOVE ON

A run is interrupted and restarted by loading the last checkpoint's weights and calling `train()` again. The loss jumps upward and the final model is worse than an uninterrupted run. Why?

A. Optimiser state and the learning-rate schedule were not restored, so training restarted with warmup at full rate.
B. The checkpoint stored weights in a lower precision, so reloading introduced rounding errors that the loss reflects.
C. Restarting reshuffles the dataset with a new seed, so the model sees examples it has already learned from in a different order.
D. Gradient accumulation counters reset mid-cycle, so the first few updates were computed on partial batches.

64 · 9 min

Loss is NaN, memory is gone, nothing is learning

THE ONE THING TO KEEP

Training failures sort into memory, NaN, flat loss, good loss with a useless model, and a training–inference mismatch — and the test that separates a pipeline bug from a data problem is deliberately overfitting ten examples, because a model that cannot memorise ten rows is not learning at all.

An ordered list, because guessing is expensive

Training failures are slower to diagnose than inference failures, because each attempt costs an hour. Work down this list rather than trying things.

Out of memory

The most common failure and the most mechanical to fix. In order, because each step costs progressively more:

1. **Reduce batch size, raise gradient accumulation.** Free.
2. `gradient_checkpointing=True`. Costs 20–30% speed, saves a lot.
3. `optim="adamw_bnb_8bit"`. Saves about three quarters of the optimiser state.
4. **Shorten** `max_length`. Check your actual length distribution first — often the 95th percentile is far below what you set.
5. **LoRA, then QLoRA.** The large wins.

One detail that catches people: **out of memory during evaluation, not training.** Evaluation batches default to the training batch size and generation builds a KV cache, so the peak can be higher than in training. Set `per_device_eval_batch_size` explicitly, lower.

And on a shared or notebook environment, memory from a previous failed run is often still held:

The curve that tells you to stop

Training loss keeps falling and validation turns upward at about step 350. Everything after that point is memorisation, and it is visible in the logged numbers long before it is visible in the output. load_best_model_at_end keeps the checkpoint from before the turn, which is not the last one and is usually better.

```
import gc, torch
del trainer, model
gc.collect(); torch.cuda.empty_cache()
```

If that does not free it, restart the kernel. Half an hour spent tuning batch size against a leak is half an hour wasted.

Loss is NaN

Three causes, in order of likelihood:

1. **Learning rate too high.** Drop it tenfold and see.
2. **fp16 overflow.** Switch to bf16 if the hardware allows. This is the T5-family failure from module three, appearing during training instead of inference.
3. **Bad data.** An empty string, a row whose labels are all `-100` so the loss is dividing by zero, or a label index outside the model's range. Check:

```
bad = [i for i, r in enumerate(ds) if all(l == -100 for l in
r["labels"])]
print("rows with no supervised tokens:", len(bad))
```

A row with no unmasked labels is a real and common bug in instruction data — it happens when truncation cuts off the whole assistant turn.

Loss is completely flat from step one

This is almost never the learning rate. It is the labels not reaching the loss:

- The column is named `label` rather than `labels`.
- Every label position is masked to `-100`.
- With PEFT, no parameters are actually trainable — `print_trainable_parameters()` reporting zero means the config's `target_modules` matched nothing, usually because the architecture's module names differ from the ones you copied.

- The model was loaded with `torch.no_grad()` somewhere, or inputs do not require gradients under a quantized base — which is what `prepare_model_for_kbit_training` fixes.

Training loss falls beautifully, the model is useless

The worst case, and it has three usual causes:

The model learned the prompt. Prompt tokens were not masked, so it is being graded on reproducing the questions. Decode the unmasked labels and look.

Overfitting. Validation loss will show it. Fewer epochs, more data, lower rank.

The task was learned trivially. If every example in one class begins with the same phrase, the model learns the phrase. This is the data-quality lesson arriving late and expensive.

It works in training and not at inference

Almost always a mismatch between how examples were assembled for training and how prompts are assembled at inference:

- A different chat template, or none.
- `add_generation_prompt=True` missing at inference.
- A system message present during training and absent afterwards.
- Different truncation, so the model sees a shape it never trained on.

The test: render one training example and one inference prompt as strings and diff them. They should be identical up to the assistant's turn. This finds the problem in two minutes and people spend days not doing it.

Catastrophic forgetting

The model gets better at your task and worse at everything else. Expected when full fine-tuning at a high learning rate. Mitigations, roughly in order:

use LoRA rather than full fine-tuning, lower the learning rate, fewer epochs, and mix in some general instruction data alongside your task data.

Worth measuring rather than assuming: keep twenty general prompts unrelated to your task and run them before and after. If the answers have degraded, you have quantified the trade rather than discovered it in production.

The cheapest habit in this module

Overfit ten examples deliberately. Train on ten rows for many epochs and check that the model reproduces them almost exactly. If it cannot memorise ten examples, your pipeline is broken — labels, masking, trainable parameters, something — and no amount of data will help. Ten minutes, and it separates a pipeline bug from a data problem before you spend three hours on the wrong one.

Do this now

Run the ten-example overfit test on whatever you are about to train. If it works, your plumbing is sound and every later failure is about data or hyperparameters. That is a genuinely useful thing to know before hour three.

BEFORE YOU MOVE ON

A LoRA fine-tune shows a loss that is flat from the very first step. What should be checked before the learning rate?

A. Whether the learning-rate warmup is too long, so the effective rate is near zero for the logged steps.

B. Whether gradient accumulation is set so high that no optimiser step has occurred within the logged window.

C. Whether the batch size is too small for the loss to be stable enough to show a trend.

D. Whether any parameters are trainable, since `target_modules` may have matched nothing.

65 · 8 min

Shipping the thing you trained

THE ONE THING TO KEEP

Publish the adapter rather than a merged republication of somebody else's weights, record the base model, training configuration and a measured result against a baseline in the card — and keep the dataset and evaluation set, because the adapter expires when its base is superseded and the data does not.

Push the adapter, not the merged model

```
model.push_to_hub("your-name/support-hi-lora")
tok.push_to_hub("your-name/support-hi-lora")
```

A LoRA adapter is a few megabytes. The merged model is several gigabytes of mostly somebody else's weights, republished. Publish the adapter:

- It is honest about what you made.
- It is fast to download and cheap to store.
- It stays usable when the base is updated, and it makes clear which base it needs.
- Several adapters can share one loaded base at serving time.

Publish a merged version as well only when there is a reason — a serving stack that cannot load adapters, or a GGUF conversion for people running on CPUs. Say in the card that it is a merge of your adapter and name the base.

The adapter card

`adapter_config.json` records `base_model_name_or_path`, and the Hub uses it to link your repository to its parent. Add the metadata that makes it findable and usable:

```
---
base_model: Qwen/Qwen2.5-7B-Instruct
library_name: peft
tags:
  - lora
  - sft
language:
  - hi
  - en
license: apache-2.0
datasets:
  - your-name/support-conversations
---
```

The `license` field needs a moment's thought and the next module covers it properly. The short version: **your adapter inherits obligations from the base model's licence.** A LoRA trained on a model with a restrictive licence does not become permissive because you wrote Apache 2.0 in your own front matter. State the base's terms as well as your own.

What the prose must contain

Six things, and a card without them is not usable by a stranger:

1. **A runnable snippet**, with the base model named and the revision pinned:

```python from peft import PeftModel from transformers import AutoModelForCausalLM, AutoTokenizer

base = AutoModelForCausalLM.from_pretrained("Qwen/Qwen2.5-7B-Instruct", dtype="bfloat16", device_map="auto") model = PeftModel.from_pretrained(base, "your-name/support-hi-lora") tok = AutoTokenizer.from_pretrained("your-name/support-hi-lora") ```

1. **What it was trained on** — how many examples, from where, in what languages, and the dataset repository if you can publish it.
2. **The training configuration** — rank, alpha, target modules, learning rate, epochs, base precision. Six lines, and without them nobody can re-
```

produce or extend your work.

3. **A measured result.** Even "macro-F1 0.78 on 300 held-out tickets, against 0.61 for the base model" is infinitely better than "works well". Say what the baseline was.
4. **A named limitation.** The thing it does badly, from your own testing. Every model has one and a card without one reads as untested.
5. **Whether the training data contained personal information**, and what you did about it.

Before you push

- **Check for secrets in the output directory.** A `.env`, a notebook with a token in a cell, a config with a connection string. `git status` in `output_dir` before uploading, and use `ignore_patterns` on `upload_folder`.
- **Check for training data you cannot publish.** Checkpoint directories sometimes contain a copy of the tokenised dataset.
- **Start private.** Make it public when you have read your own card back.

Serving several adapters on one base

```
model.load_adapter("your-name/support-hi-lora",
adapter_name="hi")
model.load_adapter("your-name/support-ta-lora",
adapter_name="ta")
model.set_adapter("hi")
```

One base in memory, several behaviours, switched per request. For a product serving several languages or several customers, this is the architecture that makes fine-tuning affordable — one 14 GB base and a directory of 20 MB adapters, rather than one 14 GB model per variant.

The maintenance you are signing up for

An adapter is pinned to its base. When a better base arrives in a few months, your adapter does not transfer — the weight shapes may match and the behaviour will not, because the thing it learned to correct has changed. You retrain.

This is why the dataset matters more than the adapter. Keep the training data, the configuration and the evaluation set in a repository, and retraining against a new base is an afternoon. Keep only the adapter and you are starting again.

Do this now

Publish an adapter privately, then open it in a fresh Colab session and load it from the card's snippet alone. If it does not work, your card is wrong — and that is exactly the test every model card should have to pass before it goes public.

BEFORE YOU MOVE ON

Why is publishing a LoRA adapter generally preferable to publishing the merged full model?

A. Merged models lose the improvement, because merging approximates the adapter's effect rather than reproducing it.

B. Adapters can be applied to any base model of the same size, which merged weights cannot.

C. It is a few megabytes, states which base it needs, and lets one loaded base serve several adapters.

D. The Hub charges for storage above a threshold, so large merged repositories incur costs adapters avoid.

CASE STUDY · MODULE 7

The fine-tune that was two different problems

A two-person company in Bengaluru building a Kannada-language support assistant for a chain of white-goods service centres.

Harsha, the founder, wanted a fine-tune. The base model answered in Kannada, but in a register that read like a textbook, and it did not know the chain's warranty rules, which are rewritten twice a year.

The chain runs forty-one service centres across two states. A customer writes in Kannada, usually from a phone, usually annoyed, and usually about a machine that is under warranty or has just fallen out of it. The assistant was meant to draft the first reply, which an agent would then send or rewrite.

Divya, the engineer, asked the question the module opens with: is the gap knowledge or behaviour. The honest answer was both, and the two halves needed opposite treatments.

The warranty rules are facts. Facts trained into weights come back fluent, confident and subtly wrong, because the model learns the shape of a policy document rather than its contents, and they change in April and October. That half became retrieval: sixty pages of policy, chunked on headings, a small multilingual embedding model, and thirty questions taken from the call log.

The register is behaviour: how a service agent speaks to a customer whose washing machine has been broken for four days. No prompt got it reliably, and few-shot examples got it for two turns and then drifted. That half was a fine-tune.

The decision

The decision with a cost on both sides was about the training data, and it was not a technical one.

Forty thousand historical chat transcripts existed. They were free, they were already labelled by who wrote what, and they were full of exactly the register they wanted. They were also full of customer names, addresses, telephone

numbers, and — in a few hundred cases — card details pasted by customers who had been told not to paste card details.

The alternative was six hundred examples written by hand with two of the chain's best agents, which meant eight working days of two people who were needed on the service floor, in a month the chain was short-staffed.

Harsha wanted the forty thousand. Divya ran the check rather than the argument: an automatic redaction pass over a sample of five hundred transcripts, then reading them. The pass caught the telephone numbers and the obvious identifiers. It missed a full address written across three lines of chat, and it missed the sentence where a customer named their employer, which in a small town names the person.

Automatic redaction misses context. Reading five hundred rows is what told them so.

Before any of this Divya spent a morning on the cheaper options, in order, because the module says to. A larger model with a better prompt closed perhaps half the register gap and none of the warranty gap. Five worked examples in the prompt closed most of the register gap for the first two turns of a conversation and then drifted, which is a known shape and not a fixable one. Only after both of those had been tried and written down did fine-tuning become the answer rather than the instinct.

The six hundred examples cost more than the model did, and that is the ordinary case rather than a surprise. Three hours of a free GPU session against nine days of two experienced agents is not a close comparison. It is also why the data went into a repository and the adapter did not particularly matter: when the base model is superseded in a few months, the adapter expires and the six hundred conversations do not. Retraining against a new base is then an afternoon rather than another nine days on the service floor.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They wrote six hundred examples by hand, over nine days rather than eight, with two agents and a rule that every example had to be a conversation one of them had actually had.

Before the real run Divya trained on ten of those examples for many epochs to see whether the model could memorise them. It could not, and the reason was that her prompt masking was wrong: loss was falling on the customer's messages as well as the agent's, so the model was being graded on writing complaints. Ten minutes found what three hours of training would have produced as a confusing result.

The run itself was a rank-sixteen adapter over a four-bit base on a free session, about three hours. Training and validation loss both fell and levelled, with no divergence. Then the check that is not a number: twenty held-out inputs, generated and read aloud. The model had learned the register, and it had also learned to promise a technician within twenty-four hours, because the agents' best conversations tended to end that way. Twelve examples were rewritten to remove the promise and the run was repeated.

The adapter is thirty-four megabytes. The warranty rules stayed in the retrieval index and were updated in April by replacing six files. The six hundred examples and the thirty questions went into a private repository, which is the part that will survive the next base model.

Why did the warranty rules belong in retrieval and the register belong in the weights?

Fine-tuning teaches behaviour; retrieval supplies knowledge. A policy is knowledge: it must be exact, it must be citable, and it changes twice a year, so training it in produces confident paraphrases that cannot be updated without retraining. A register is behaviour: it is a way of writing rather than a fact, it cannot be looked up, and it is precisely what a few hundred well-chosen examples teach. Splitting the problem this way let each half be fixed by the cheaper method.

The ten-example overfit test cost ten minutes. What did it find, and what was the alternative cost?

It found that the prompt positions were not masked, so loss was computed on the customer's turns as well as the agent's. A model that cannot memorise ten rows is not learning, so the test separates a pipeline bug from a data problem before any real run. Without it the three-hour run would have completed with a falling loss curve and produced a model that writes complaints rather than answers them — a failure that looks like success in every logged number.

The model promised a technician within twenty-four hours. What caused it, and why does only reading the output catch it?

The examples were the agents' best conversations, and their best conversations tended to end in that promise, so the habit was in the data. A model copies the register it is shown, including commitments. Loss cannot see it: a promise is fluent, in the right language and in the right style, so validation loss improves while the model acquires a liability. Generating twenty held-out answers and reading them is the only check that surfaces a learned habit you would have rejected.

MODULE 7 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Your assistant does not know this year's refund policy. Which is the right tool, and why?

A. Fine-tune on the policy until the model reproduces it word for word.
B. Retrieval, because fine-tuning teaches behaviour rather than supplying facts.
C. A higher LoRA rank, so that the policy has enough capacity to fit.
D. Both, since a fine-tune will eventually make the retrieval step unnecessary anyway.

2 LoRA cuts training memory for a 7B model by far more than the trainable parameter count alone suggests. What is the main reason?

A. The base weights are discarded from memory once the first epoch has completed.
B. No gradients are computed at all when adapters are in use.
C. Optimiser state is held per trainable parameter, so it shrinks with them.
D. The adapter lives on disk rather than in memory during the run.

3 Which learning rate suits a LoRA adapter, and why does it differ from full fine-tuning?

A. 2e-5, because a pretrained model must only ever be nudged gently.
B. The same as full fine-tuning; LoRA changes memory, not optimisation.
C. 1e-3, because an adapter behaves exactly like a newly initialised classification head.
D. 2e-4, because you are training fresh parameters rather than nudging trained ones.

4 You load a 4-bit base, attach LoRA, start training, and the loss barely moves. Which omission fits?

A. prepare_model_for_kbit_training was never called on the base model.
B. bnb_4bit_use_double_quant was left at its default of false.
C. gradient_checkpointing was enabled, which prevents gradients flowing.
D. The base was loaded with device_map auto rather than a single device.

5 You train on ten examples for many epochs and the model cannot reproduce them. What does that tell you?

A. The dataset is too small overall and needs augmenting before the real run.
B. The learning rate is a little low for a model of this size.
C. The model is too weak for the task and a larger base is required.
D. The pipeline is broken: masking, label names, or trainable parameters.

6 Training loss keeps falling while validation loss turns upward from about step 350. What should you do?

A. Train for longer, because validation recovers once the schedule decays.
B. Stop, and keep the checkpoint from before the turn.
C. Raise the learning rate so the run escapes the local minimum it is in.
D. Switch from bfloat16 to float16, which stabilises a diverging run.

MODULE 8

Depending on it, and publishing so others can

The last block is about the parts that decide whether your work survives contact with other people: what a licence actually permits and where the law is genuinely unsettled, how to judge a model with your own examples rather than a leaderboard, where inference should run and what it costs, what a card must contain to be auditable, how to contribute back, and how to pin everything so the thing you shipped still works next year.

> BY THE END OF THIS MODULE YOU CAN
>
> State what a model's licence permits for your intended use and where the question is unresolved rather than answered, choose between a provider, a dedicated endpoint and your own hardware on stated cost and privacy grounds, judge a model with thirty examples of your own, and publish work a stranger can audit and reproduce

66 · 8 min

A token in a public Space is a token you have given away

THE ONE THING TO KEEP

Scope tokens as narrowly as the job allows, keep them in environment variables and Space secrets rather than in code, and revoke on suspicion rather than reasoning about whether a leak was seen.

Why you need a token at all

Most of the Hub is readable without an account. You will still need a token for four things: private repositories, gated models, uploading anything, and any hosted inference you are billed for. A token is a string beginning `hf_` that stands in for you. Anything holding it *is* you, as far as the Hub is concerned.

Make one that can do as little as possible

Access tokens live under your account settings. There are three kinds, and the choice matters more than the two seconds it takes.

- **Fine-grained** — you tick exactly which permissions it has, and can scope it to named repositories or a single organisation. This should be your default.
- **Read** — can read everything you can read. Convenient, broad.
- **Write** — can create, modify and delete your repositories. A leaked write token is not an inconvenience.

Give each token a name that says where it lives — `laptop-read`, `render-space`, `ci-upload` — because in six months you will want to revoke one of them without breaking the others, and an untitled list of five tokens gives you no way to choose.

Where it lives on your machine

```
hf auth login
```

This prompts for the token and writes it to a file under your Hugging Face home directory, typically `~/.cache/huggingface/token`. Every library on that machine can read that file. That is the trade for never typing it again, and it is usually the right trade on your own laptop and the wrong one on a shared server.

In code, never write the token as a literal. Read it from the environment:

```
import os
from huggingface_hub import login

login(token=os.environ["HF_TOKEN"])
```

Most of the libraries pick up an `HF_TOKEN` environment variable on their own, so often you set the variable and write no authentication code at all.

Gated repositories

Some repositories — Llama, Gemma, a number of speech models and datasets — sit behind a gate. The page shows the licence and a button. Sometimes access is granted instantly. Sometimes there is a short form asking for your name, affiliation and country, and a wait of hours or days for a human. Sometimes you are refused.

That form is a condition of the licence, not a formality, and the name you give is meant to be your real one.

The symptoms are worth memorising, because they look alike and mean different things:

- **401** — there is no valid token in the request at all.
- **403**, or a `GatedRepoError` — the token is valid but that *account* has not been granted access to that repository.

The second one catches people out constantly. You accepted the terms in a browser while logged in as yourself; the code is running with a token belonging to a service account, a colleague, or a Space. Access is per account, and accepting the terms once does not propagate.

The rule, stated once

Anywhere a token is visible to someone else, it is theirs. A public Space's files are public. A notebook pushed to GitHub is public. A screenshot of your terminal posted in a support thread is public, and the token is legible in it. Automated scanners crawl for the `hf_` pattern within minutes of publication.

Hugging Face and GitHub both scan for leaked tokens and will revoke them, which is a safety net rather than a plan — it fires after publication, not before. If you think a token might have been seen, revoke and reissue. That takes thirty seconds and requires no judgement, whereas reasoning about whether anyone actually saw it requires perfect information you do not have.

In a Space, the token goes in Settings under secrets, with the name `HF_TOKEN`, and never in `app.py`.

One organisational point

A token is a person, not a service. When someone leaves a team, their tokens keep working until somebody revokes them, and any pipeline built on a departed colleague's token dies the day you do. For anything shared, create an organisation, add a dedicated account or a fine-grained token owned by the organisation, and scope it to the repositories it needs.

Do this now

Open your access tokens page. If there is a write token you cannot account for, revoke it. Then create one fine-grained read token, named after the machine you are sitting at, and log in with that.

BEFORE YOU MOVE ON

A script that downloads a gated model runs fine on a developer's laptop. Deployed to a server, it fails with a 403 and a gated-repository error. The token is definitely present and definitely valid. What is the most likely cause?

A. The gate rate-limits downloads by IP address, and the server's address has not been approved.
B. The server's token belongs to a different account, and access to a gated repository is granted per account rather than carried by any valid token.
C. The token is a read token where a gated repository requires write scope to accept the terms programmatically.
D. The acceptance expired, and gated licences must be re-accepted periodically.

67 · 10 min

What the licence tag does and does not settle

THE ONE THING TO KEEP

A fine-tune inherits its base model's terms however its own tag reads, so licence checking means following `base_model` to the root — and the questions the licence does not answer, about training data and output ownership, are genuinely unresolved and differ by country rather than merely being undocumented.

Four families, and what separates them

Permissive open-source — Apache 2.0, MIT, BSD. Commercial use, modification and redistribution allowed. Apache 2.0 adds an express patent grant and requires you to keep the notice; MIT is shorter and does neither. For a model you want to build a business on, this is the clean case. Qwen, Mistral's open releases, Whisper, most BERT-family models and a large share of the Hub sit here.

Copyleft — GPL, AGPL, and in the model world occasionally applied to code rather than weights. These require derivative works to carry the same licence. AGPL extends that to software offered over a network, which matters for a hosted service. Uncommon for weights, common for the tooling around them, and worth reading when a serving stack you depend on carries it.

Open weights with conditions — the Llama community licence, Gemma's terms, and several others. The weights are downloadable and the terms are not open-source by the usual definition. Typical conditions: acceptable-use policies that forbid certain applications, naming requirements for derivatives, and in Llama's case a threshold on monthly active users above which you must ask permission. Most people never reach the threshold; you should still know it exists before you build on it.

Non-commercial and research-only — `cc-by-nc-4.0`, bespoke research licences. Fine for learning and for a demo, not for anything that earns money, including internally at a company. This is the trap people fall into, because the model is downloadable and works perfectly.

Four licence families, and what separates them

Permissive: Apache-2.0, MIT, BSD	Commercial use, modification, redistribution. Keep the notice.
Copyleft: GPL, AGPL	Derivatives carry the same terms; AGPL reaches a hosted service.
Open weights with conditions	Use policies, naming rules for derivatives, a monthly user threshold.
Non-commercial and research-only	Downloadable, excellent, and may not earn money, including internally.

And the rule that overrides all four: a fine-tune inherits its base model's terms whatever its own tag reads, so the check is to follow base_model until you reach a repository with no parent. Nothing here is legal advice, and the questions about training data and output ownership are genuinely unresolved.

The inheritance rule

The one thing to carry from this lesson:

> ***A fine-tune inherits its base model's terms, whatever its own card says.***

An adapter trained on a non-commercially licensed model is a derivative work of that model. Somebody tagging their repository `apache-2.0` does not change what the base permits, and they frequently do tag it that way — sometimes carelessly, sometimes because they genuinely believe a LoRA is a separate artefact.

So the check is always two-step: read the model's licence, then follow `base_model` and read that one, and keep following until you reach something with no parent. The Hub's model tree makes this a few clicks.

This applies to data too. A model fine-tuned on outputs generated by a commercial API may be restricted by that API's terms of service, which several providers have written to forbid using their output to train competing models. Whether such a clause is enforceable, and against whom, is not settled — which is exactly why a card should disclose it rather than stay silent.

What the licence does not cover

The training data. A permissively licensed model may have been trained on material whose status is contested. The model's licence is a grant from whoever trained it about the weights they hold. It is not a warranty about what went into them, and almost no card claims otherwise.

The outputs. Whether text or images a model generates can be copyrighted, by whom, and whether they can infringe something in the training data, is unresolved and differs by jurisdiction. Some offices have said works without human authorship are not protected; others differ; litigation is ongoing in several countries. Anyone who tells you this is settled is describing one jurisdiction's current position as though it were universal.

Your use case. An acceptable-use policy can forbid an application the licence would otherwise permit.

Where the disagreement actually sits

It is worth naming the shape of it rather than pretending there is an answer.

One side holds that training on lawfully accessed material is a transformative, non-expressive use, closer to reading than to copying, and that requiring permission for every work would make the technology impossible. The other holds that copying at scale for commercial gain is not excused by the output being different, and that creators whose work made the model possible are entitled to consent and compensation. Several countries have text-and-data-mining exceptions, usually with conditions such as lawful access and a right for rightsholders to opt out. Others have no such provision. Courts in more than one jurisdiction are actively deciding cases as this is written.

This course cannot resolve that and will not pretend to. **Nothing here is legal advice.** What it can tell you is what a practitioner should do: record what you used and under what terms, read the base model's licence rather than the fine-tune's tag, and ask a lawyer in your own country before you stake a business on an interpretation.

The practical checklist

Before building on a model commercially:

1. Read the licence file in the repository, not the tag on the page.
2. Follow `base_model` to the root and read that licence too.
3. Look for an acceptable-use policy — often a separate file or a linked page.
4. Check for user or revenue thresholds.
5. Check any datasets you fine-tune on, with the same two-step rule.
6. Write down what you found, with dates and links. Licences change between versions, and the version you downloaded is the one you agreed to.
7. For anything material, ask a lawyer. The cost of an hour of advice is small against the cost of rebuilding.

And the free path

If licence uncertainty is blocking you, there is almost always a permissively licensed alternative within a few points of quality. Apache 2.0 models exist at every size, for almost every task. Choosing one is often cheaper than resolving the question.

Do this now

Take the model you are most likely to build on, open its actual `LICENSE` file, and follow `base_model` until you reach a repository with no parent. Note every condition you find. This takes five minutes and it is the difference between a decision and an assumption.

BEFORE YOU MOVE ON

A repository tagged `apache-2.0` declares `base_model` as a model released under a non-commercial research licence. What follows?

A. The derivative is bound by the base model's restrictions regardless of its own tag.

B. The adapter's own weights are original work, so the permissive tag governs everything the author created.

C. Whichever licence is more recent takes precedence, so the fine-tune's terms supersede the base model's.

D. The tags conflict, so no licence applies and the work falls into the public domain by default.

---

68 · 9 min

The leaderboard is not measuring your task

> THE ONE THING TO KEEP
>
> A leaderboard narrows the field and cannot pick the winner, because contamination, selective reporting and style effects all sit between a public score and your task — thirty of your own examples decide it.

The number that moved you

A model appears at the top of a table. You pick it. Two weeks later it is worse at your actual work than the boring model you were already using, and you

cannot say why.

Leaderboards on the Hub come and go — the well-known Open LLM Leaderboard ran for years and was archived in 2025 — and there are focused ones for embeddings, speech, code, vision and individual languages. Their failure modes outlive any particular table, so learn the failure modes.

Contamination

Benchmarks are published so that people can use them. That means the test questions and their answers are on the public web. Models are pretrained on scrapes of the public web. So a benchmark's answers can end up in a model's training data, and a model that has memorised the answers scores like a model that can reason.

This is usually not cheating. It is a structural consequence of training on the internet the benchmark lives on, and it gets worse the longer a benchmark has existed. Two signs to look for:

- A large gap between a famous public benchmark and a private or freshly written one on the same skill.
- A high score on an old benchmark next to an ordinary score on a benchmark released after that model's training data cutoff.

This is why benchmarks with hidden test sets and rolling questions exist, and why a score on a benchmark from three years ago tells you less every year.

Selective reporting

A card shows the benchmarks the model won. Nobody publishes the six they lost. When you see five results, ask which of the standard set for that task is absent. The gaps are information — often better information than the numbers, because the numbers are chosen and the gaps are not.

Preference arenas and the style problem

Human preference arenas show two anonymous answers and count votes. That measures something real and something no automatic benchmark cap-

tures. It also measures formatting, length, confidence and agreeableness.

A model that answers in tidy headed bullet points, hedges nothing and writes four hundred words tends to beat a model that is correct in two sentences. A model tuned to agree with the user beats one that says the premise is wrong. So an arena rank is a measurement of likeability under one style of interaction, which correlates with usefulness and is not the same thing. Read it as such.

The gap that actually matters

The famous general benchmarks are, overwhelmingly, English multiple-choice questions about general knowledge. Consider what your work is:

- Triaging complaints written in Hindi with English words mixed in.
- Transcribing Nigerian-accented English over a noisy phone line.
- Pulling five fields out of a scanned Bengali invoice.
- Rewriting product descriptions in a house style, without inventing features.

For none of these does the leaderboard rank predict much. Multilingual scores are usually an average across many languages, and an average is exactly the statistic that hides the fact that your language is one of the ones that fails. Worse, a model heavily tuned for benchmark-style answers can be measurably *worse* at following a plain instruction in your format, because that tuning taught it to produce essays and lettered options.

The evaluation you can build this afternoon

The fix is small and unglamorous, and it beats every leaderboard for your decision because it measures the thing you are actually buying.

1. Collect **thirty real examples** from your own work. Not invented ones — real ones, including the awkward ones.
2. Write down, for each, the output you would accept. Where several outputs are acceptable, write the rule.

3. Run each candidate model over all thirty. Free Spaces and provider free credits are enough for this.
4. Score by hand, in a spreadsheet, one row per example and one column per model.
5. Read the failures rather than the total. The pattern in what breaks is the finding.

That is half a day, and you can rerun it every time you consider switching. /learn/evaluating-ai sets out the method in full, including how to keep a set like this honest over time.

The rule that follows

Use the leaderboard to get from four hundred candidates to five. Use your thirty examples to get from five to one. Filtering the Hub by task and sorting by downloads is a reasonable third filter — popularity is not quality, but a heavily used model is one whose bugs somebody else has already hit and written up in the Community tab.

Do this now

Open the eval table of a model you are considering and name one standard benchmark for its task that is missing from the table. Then write down five real examples from your own work. Twenty-five to go.

BEFORE YOU MOVE ON

A model tops a well-known benchmark, then scores unremarkably on a similar test written after its training data cutoff. What is the most likely explanation?

A. The new test is harder, so the drop reflects difficulty rather than anything about the model.
B. The model was fine-tuned after the benchmark run and has lost some general capability since.
C. Test items from the older public benchmark were present in the training scrape, so part of that score reflects memorisation rather than capability.
D. The newer test uses a different prompt format, so the gap is a formatting mismatch that better prompting would close.

69 · 9 min

The evaluation that decides it

THE ONE THING TO KEEP

A leaderboard narrows the field and thirty examples of your own decide it, because only your set contains your languages, formats and hard cases — and at that size it reliably detects large differences and must never be used to justify small ones.

Why thirty, and why yours

A leaderboard narrows the field to three candidates. Nothing public can pick the winner, because no public benchmark contains your text, your languages, your formats or your users' phrasing. The thing that decides it is a small set of your own examples, and it takes an afternoon to build.

Thirty is chosen deliberately. It is few enough to read every failure, which is where the information is, and enough to detect the differences worth acting on — a model that cannot handle your language, a configuration that broke struc-

tured output, a prompt change that lost a capability. It will **not** detect a two-point difference, and treating a 26-out-of-30 against a 27-out-of-30 as a result is how people convince themselves of noise.

Building it

```
{"id": "t001", "input": "मेरा ऑर्डर कैंसिल करना है", "expect_label":
"cancellation", "note": "hindi"}
{"id": "t002", "input": "where's my stuff??", "expect_label":
"delivery", "note": "informal"}
{"id": "t003", "input": "invoice INV-2024-8823 wrong amount",
"expect_label": "billing", "note": "has id"}
```

Four rules for choosing the thirty:

1. **Take them from real inputs**, not invented ones. Invented examples are always cleaner than reality.
2. **Cover the shape of your traffic**, including the rare classes you care about. This is stratified sampling, and it means you cannot quote a single headline rate as if it were production performance.
3. **Include the hard cases deliberately** — the other language, the misspelling, the empty input, the extremely long input, the one with an identifier in it, the ambiguous one.
4. **Include three where the right answer is "I do not know"**, so you can see whether the model invents something.

Redact personal data before this file exists anywhere. It is a copy of real user text, and it will end up in a repository.

Scoring what can be scored

```
import evaluate, json

rows = [json.loads(l) for l in open("eval.jsonl")]
acc = evaluate.load("accuracy")

def run(model_fn):
    preds, refs, fails = [], [], []
    for r in rows:
        got = model_fn(r["input"])
        preds.append(got); refs.append(r["expect_label"])
        if got != r["expect_label"]:
            fails.append((r["id"], r["note"], r["input"][:60],
got))
    return acc.compute(predictions=preds, references=refs),
fails
```

More of your task is exactly scorable than you expect: is the JSON valid, is the label right, is the number correct, is the date in the right format, is the answer in the right language, is it under the length limit. Score all of those automatically and cheaply.

For genuinely open-ended output, read it. Thirty is a deliberate size because thirty is readable. A model scoring another model is a real technique with real biases — position effects, a preference for its own style, a preference for longer answers — and at thirty items it buys you nothing you cannot get by reading.

Comparing two models honestly

Run both on the identical set with identical prompts and identical generation settings. Differences of ten points or more at thirty items are worth acting on. Smaller ones are not.

Two confounders that ruin more comparisons than anything else:

- **Different chat templates or prompts.** You are then comparing prompts, not models.

- **Inherited generation defaults.** Module two's point: two models can appear to differ in quality when their `generation_config.json` files differ in temperature.

And always include a baseline that is not a large model: a keyword rule, a small fine-tuned classifier, the current system. A great many projects discover that a 40-line rule scores within a few points of the model, which is a finding worth having in week one rather than month three.

The tools, when you want more than a script

`evaluate` loads standard metric implementations so everybody's F1 is the same F1. `lighteval` runs standard benchmark suites against a model, locally or through a provider, if you want to reproduce a leaderboard number yourself rather than trust it — which is occasionally illuminating, because reproducing a published score is harder than it sounds and the gap is informative.

Neither replaces the thirty. They are for the questions that are about the field; the thirty is for the question that is about you.

Keep it, and grow it

Store the file in your repository, versioned. Add a row every time production surprises you: an input that produced a wrong answer becomes a permanent test. Over a year the set becomes a record of everything you have learned about your own problem, and it is the single most valuable artefact a small team builds.

When you change model, prompt, quantization or library version, run it. That is the whole discipline.

Do this now

Open a file and write ten rows from real inputs in the next fifteen minutes. Not thirty — ten. Run your current system against them and read every failure. You will find something in the first ten, and that is the point.

BEFORE YOU MOVE ON

Two candidate models score 26/30 and 27/30 on a team's own evaluation set. What is the correct conclusion?

A. The one-point difference is within noise at this sample size, so the choice should rest on other grounds.

B. The second model is better and should be adopted, since the comparison used identical inputs and settings.

C. The set is too easy, since both models pass most items, and harder examples should replace it entirely.

D. Neither result is usable because thirty items cannot support any conclusion about model quality.

70 · 8 min

Renting the compute instead of owning it

THE ONE THING TO KEEP

The Hub routes inference to third-party providers behind one OpenAI-compatible interface, and the choice between serverless, a dedicated endpoint, your own server and the user's device turns on token volume against GPU utilisation — with the crossover much higher than teams assume and privacy deciding it outright for sensitive data.

The Hub as a router

Many model pages carry an inference widget and a code snippet that calls a hosted API. Behind it sits a router: Hugging Face forwards your request to one of several third-party providers — Together, Fireworks, Replicate, SambaNova, Cerebras and others — under a single token and a single bill.

```
from huggingface_hub import InferenceClient

client = InferenceClient(api_key=os.environ["HF_TOKEN"])

out = client.chat_completion(
    model="Qwen/Qwen2.5-7B-Instruct",
    messages=[{"role": "user", "content": "Summarise this in
two lines: ..."}],
    max_tokens=200,
)
print(out.choices[0].message.content)
```

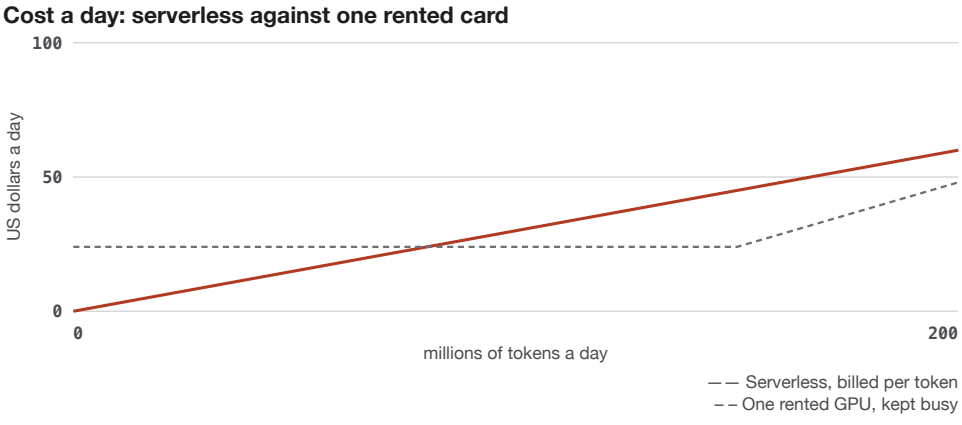
The interface is OpenAI-compatible, so the `openai` package pointed at the router's base URL works unchanged — which matters, because it means switching between a hosted model, a local llama.cpp server and a provider is a change of base URL rather than a rewrite.

There is a small monthly credit on free accounts and a larger one on paid; beyond that you are billed at the provider's rates with no markup, which is stated on the pricing page and worth re-reading, since these arrangements change.

The four options, and the honest comparison

Serverless through a provider. No infrastructure, per-token billing, cold starts on less popular models, and your input goes to a third party. Right for prototypes, spiky traffic and anything where you would rather not run a GPU.

A dedicated endpoint. Hugging Face Inference Endpoints, or the equivalent on a cloud, gives you a container running your model on hardware you rent by the hour, with autoscaling and scale-to-zero available. Predictable latency, private, and billed for uptime rather than requests — the same trap as paid Spaces, with the same fix: configure scale-to-zero before you configure anything else.



Order-of-magnitude figures rather than quotes, and the rates move. The shape is what transfers: per-token billing is a line through the origin, a rented card is flat until you need a second one, and the crossover sits much further right than teams assume. It moves right again by however idle the card is, because idle costs the same as busy.

Your own server. vLLM or text-generation-inference on a rented or owned GPU. Cheapest per token at volume, most work, complete control of data.

On the user's device. transformers.js in a browser, GGUF on a laptop, a small model on a phone. Zero marginal cost, complete privacy, limited to small models. Module three covered it, and it remains the most underused option.

The arithmetic that decides it

The crossover is not where people guess. Take a rough figure of a few tens of US cents per million tokens for a small open model through a provider, and around a dollar an hour for a modest GPU.

At 100,000 tokens a day, serverless costs a few cents a day and a dedicated GPU costs twenty-four dollars. Serverless wins by a very wide margin.

At 100 million tokens a day, serverless is in the tens of dollars a day and one busy GPU is still twenty-four. Now owning wins, provided you can keep it busy — and utilisation is the variable nobody models. A GPU idle 80% of the time costs the same as a GPU at full load, so your effective rate is five times the headline one.

Run the arithmetic with your own volume before choosing. Most teams self-host too early, and a smaller number stay on per-token billing long past the point where a single card would be cheaper.

Privacy is not a footnote here

Sending data to a provider means it leaves your machine. What is logged, for how long, and whether it may be used for training differs by provider and by plan, and the answer lives in their terms rather than in the Hub's interface. For health, financial, legal or children's data — or anything a user would not have pasted into a public website — that question has to be answered before the integration is written, not after.

The local path is genuinely available and genuinely free. A 3B model at `Q4_K_M` on a CPU handles a great deal of routine work, and nothing leaves the building.

Reliability, stated plainly

A provider can deprecate a model, change its quantization, rate limit you, or go down. All four have happened to somebody this year. Three mitigations cost little:

1. **Keep the provider behind one function** in your code, so swapping is a small change.
2. **Have a fallback**, even a smaller local model, for when the primary is unavailable.
3. **Cache aggressively.** A large share of production requests are repeats, and a hash-keyed cache is often the single biggest cost saving available.

Do this now

Estimate your monthly token volume, then compute what it costs serverless and what one rented GPU costs for the same month. Write both numbers down. Most people have never done this and are surprised by which side they are on.

BEFORE YOU MOVE ON

A service handles about 200,000 tokens a day and moves from per-token billing to a dedicated GPU endpoint to save money. The bill rises. Why?

A. Dedicated endpoints add a management fee on top of the hourly hardware rate, which per-token pricing avoids.

B. Endpoint hardware is billed per hour whether or not requests arrive, and this volume leaves it mostly idle.

C. Cold starts on a scale-to-zero endpoint trigger repeated billed initialisations that exceed the steady-state cost.

D. Dedicated endpoints cannot batch requests, so each one occupies the GPU exclusively and throughput falls.

71 · 8 min

When transformers stops being the right server

THE ONE THING TO KEEP

Purpose-built servers beat a transformers loop structurally rather than by tuning — continuous batching keeps the GPU full, paged attention stops the KV cache fragmenting, and prefix caching reuses a shared system prompt — and unlike at training time, quantization under these stacks buys speed as well as memory.

The wall you hit

A Flask endpoint wrapping `model.generate` works, and it works for one user at a time. Requests queue, the GPU sits idle between them, and at ten concurrent users latency becomes unacceptable while utilisation stays under a third.

The cause is structural rather than a tuning problem. A naive server processes one batch to completion before starting the next, so a request that arrives one

token into a 500-token generation waits for all 500. And the KV cache is allocated as a contiguous block sized for the worst case, so most of the memory reserved for it is never used.

What purpose-built servers do differently

Continuous batching. New requests join the running batch at the next token step, and finished ones leave immediately. The GPU stays full. This alone is typically a several-fold throughput gain at the same latency.

Paged attention. The KV cache is allocated in fixed-size blocks, like operating-system memory pages, rather than one contiguous reservation per sequence. Fragmentation drops sharply and far more concurrent sequences fit in the same memory.

Prefix caching. When many requests share an opening — a long system prompt, a retrieved document — the computation for that shared prefix is done once and reused. For a retrieval-augmented application where every request carries the same instructions, this is a large and easy saving.

```
pip install vllm
vllm serve Qwen/Qwen2.5-7B-Instruct --max-model-len 8192
```

That gives an OpenAI-compatible server on port 8000. Text-generation-inference is Hugging Face's equivalent, distributed as a container:

```
docker run --gpus all -p 8080:80 \
  ghcr.io/huggingface/text-generation-inference:latest \
  --model-id Qwen/Qwen2.5-7B-Instruct
```

Both are free and open source. Both expose the same API shape as the providers in the last lesson, so your client code does not change.

The CPU path is the same idea

```
llama-server -hf unsloth/Qwen2.5-7B-Instruct-GGUF:Q4_K_M --port
8080 -c 8192
```

llama.cpp's server also does continuous batching and prefix caching, on a CPU or Apple silicon, with no GPU at all. For a small team serving modest internal traffic this is frequently the whole answer — one machine, one binary, no cloud account.

What you give up

- **Flexibility.** These servers run generation. A custom head, an intermediate layer, a bespoke loss — none of that fits, and transformers remains the right tool for it.
- **Model coverage.** Support is per-architecture and a brand-new model may not be supported for weeks.
- **Startup time.** Loading and warming a large model takes minutes, which matters if you were hoping to scale to zero.
- **Operational surface.** Now you own a service: monitoring, restarts, upgrades, capacity.

Quantization and serving, together

Serving stacks load AWQ, GPTQ and native low-precision formats with kernels written for them, so unlike the bitsandbytes path from module three, quantization here buys **speed as well as memory**. A 4-bit AWQ model under vLLM is genuinely faster than bfloat16, not merely smaller. That is the reverse of the training-time picture and worth holding separately in your head.

What to measure before and after

The same four numbers as module three, plus two that only appear under load:

- **Throughput** in tokens per second, aggregated across concurrent requests.
- **Time to first token** at the 95th percentile, not the mean.
- **Concurrency at which latency degrades** — the number that tells you when to add a second machine.
- **GPU utilisation.** If it sits at 30% under load, the server is the bottleneck rather than the hardware, and that is the finding that justifies this whole lesson.

Load-test with something that issues concurrent requests rather than a loop. A sequential loop measures latency and tells you nothing about the problem these servers exist to solve.

Do this now

Take a model you serve and run the same twenty concurrent requests against a plain transformers loop and against vLLM or llama-server. Record aggregate throughput for both. The ratio is usually large enough that the decision makes itself.

BEFORE YOU MOVE ON

A GPU shows 30% utilisation while users complain about latency under concurrent load. What does this most likely indicate?

A. The model is too large for the card, so layers are being swapped between GPU and CPU memory each step.

B. Requests run batch-by-batch to completion, so the GPU idles while new ones wait.

C. The KV cache has exhausted available memory, forcing the server to recompute attention for earlier tokens.

D. Tokenization on the CPU is the bottleneck, since it cannot keep the GPU supplied at this request rate.

72 · 10 min

Publish it so a stranger can use it without writing to you

THE ONE THING TO KEEP

A card with a runnable snippet, a named limitation and a deliberate licence makes your work usable by strangers, and everything you would regret publishing must be decided before the first push, because copies survive deletion.

You have something worth uploading sooner than you think

People assume publishing on the Hub means releasing a foundation model. Almost nothing on it is that. Things that are genuinely useful and need no GPU to produce:

- Four hundred labelled examples in a language the Hub barely covers.
- An evaluation set for a task in your industry — thirty real invoices with the correct fields, say.
- A Space that chains two existing models into one tool that did not exist.
- A converted format: a GGUF or ONNX build of a model that only ships PyTorch weights.
- A LoRA adapter, which is a few tens of megabytes rather than a few gigabytes.

Each of these is somebody's afternoon and somebody else's month.

Three ways up

The website. Create a new model, dataset or Space and drag files into the browser. This works from a phone for small files, and it is the right route the first time, because you see what the interface expects.

The command line, for anything larger or repeated:

```
hf auth login
hf upload my-name/marathi-support-tickets ./data --repo-
type=dataset
```

From the library you are already in. Most Hugging Face libraries have a `push_to_hub()` that creates the repository and uploads in one call at the end of a training script.

Create it **private** first. Get it working, write the card, then flip it to public in Settings when you are ready. There is no penalty for this and it removes the pressure that makes people publish half-finished things.

The front matter that makes it findable

The YAML block at the top of `README.md` is what search, filters and the Hub's own linking read. Without it your work exists and nobody finds it:

```
---
license: apache-2.0
language:
  - mr
  - en
base_model: openai/whisper-small
pipeline_tag: automatic-speech-recognition
tags:
  - speech
  - marathi
---
```

`base_model` is the one people forget, and it works in both directions — your repository appears on the parent model's page under its fine-tunes, which is how strangers find you.

The card a stranger can act on

Five things, and a card with all five is better than most of what is on the Hub:

1. **What it does**, in one sentence, at the top.

2. **A code block they can paste** that runs as written. Test it in a clean environment. A snippet with an undeclared import is the commonest defect on the Hub.
3. **What it was built from** — the base model, the data, roughly how much and from where.
4. **What it is bad at**, named specifically. Not *may produce inaccurate output*, but *word error rate roughly doubles on speakers under twenty*, or *only tested on invoices from two vendors*. You will be right, it costs you nothing, and it saves the next person a week.
5. **The licence and any obligation attached to it.**

Choosing a licence on purpose

- **Apache-2.0 or MIT** for maximum reuse. Apache adds an explicit patent grant, which is why organisations often prefer it.
- **CC-BY-4.0** for data where you want credit.
- **CC-BY-SA** if you want derivatives to stay open, accepting that this makes some commercial adoption impossible. That is a choice, not an accident — make it knowingly.
- **OpenRAIL** variants add use restrictions, and are common on generative models.

And name things honestly. **Open weights is not the same as open source.** Downloadable weights under a licence with use restrictions is a real and valuable thing, and it is not open source under the Open Source Initiative's definition — whose 2024 definition for AI also asks for meaningful information about the training data, which most released models do not provide. Both categories are worth having. Calling one by the other's name is how the word stops meaning anything. /learn/open-models goes further.

What you cannot take back

Once something is public, people clone it. Deleting the repository does not delete the copies, and mirrors of the Hub exist. So the decisions that matter happen **before** the first push, not after:

- Is there personal data in this, and did those people agree.
- Is any of it a client's, or under an agreement.
- Is there a face or a voice in it belonging to someone who did not consent to being in a dataset.
- If you promise a takedown route in the card, is it a promise you can actually keep.

If a dataset contains people, say in the card how someone asks to be removed, and mean it.

Do this now

Publish one small real thing this week. Two hundred rows, a card with all five sections, a licence you chose deliberately, a limitation stated plainly. It takes about an hour, and it is the whole difference between having used the Hub and being on it.

BEFORE YOU MOVE ON

A teacher uploads a public dataset of student essays with names removed, then realises two essays still contain identifying details and deletes the repository the same day. What is the accurate view of the situation?

A. Deletion removes the repository from the Hub but not any copies already cloned or mirrored, so the disclosure has to be handled as one that already happened.

B. Because it was public for less than a day, no copies will exist and deleting resolves it.

C. Rewriting the git history to drop the two files removes them everywhere, since the Hub is a git host.

D. Making the repository private instead of deleting it retracts the data while preserving the work.

73 · 9 min

Writing a card somebody can audit

THE ONE THING TO KEEP

A card written for an auditor — documented data, stated intended and out-of-scope use, measured evaluation against a baseline with per-group results, a specific named limitation and a contact — is also the card a careful user needs, and it is the practice that survives whatever the specific regulation turns out to be.

Two audiences, one document

A model card is read by somebody deciding whether to use your model, and — increasingly — by somebody asking how a system in production was built. The second reader is new, and they ask harder questions: what was it trained on, how was it tested, who is accountable, what does it do to people it was not designed for.

Writing for the second reader improves the document for the first, because the questions are the same ones a careful user has and does not ask.

The structure

Front matter that is correct. `license`, `base_model`, `language`, `datasets`, `pipeline_tag`, `library_name`. These are indexed, they make your work findable, and a wrong `pipeline_tag` means nobody filtering for your task ever sees it.

One paragraph: what it is and what it is for. No marketing. "A LoRA adapter for Qwen2.5-7B-Instruct that classifies Hindi and English customer support messages into nine intents."

Two limitation sections

True of everything	Specific, and free to write
• May produce inaccurate output	• Accuracy on Hindi is eight points below English
• Use with caution	• Trained on one company's support queue in 2024
• Performance may vary by input	• Untested on Hinglish code-switching, common here
• Not suitable for all use cases	• Word error rate roughly doubles on phone audio
	• Not for deciding anything about a person unreviewed

The left column is true of every model ever released and therefore informs nobody. The right column costs an afternoon of your own testing, which you have already done, and it is the section most likely to stop somebody misusing your work.

A runnable snippet. With the revision pinned. Test it in a fresh environment before publishing — the commonest defect in cards is a snippet that never ran.

Intended use, and out-of-scope use. The second is the one people skip and the one that protects everybody. "Not for medical, legal or financial advice. Not evaluated on languages other than Hindi and English. Not suitable for deciding anything about a person without human review."

Training data. Source, size, date range, languages, how it was collected, and whether it contained personal data. If you cannot publish the dataset, describe it. A card that says nothing about data is a card that cannot be assessed.

Training procedure. Base model and revision, method, hyperparameters, hardware, duration. Six lines, and they are what makes your work reproducible rather than anecdotal.

Evaluation. What you measured, on what, against what baseline. A number without a baseline means nothing; a number without a described set means less.

Limitations and bias. This is the section that separates a card from a README, and it has to be specific. Not "may produce biased output" — which is true of everything and informs nobody — but "accuracy on Hindi messages is 8 points below English; the training data came from one company's support queue in 2024 and skews toward delivery complaints; the model has not been tested on Hinglish code-switching, which is common in this domain."

Environmental cost, if you can compute it. Hardware, hours, and where it ran. One line.

Contact. A discussion tab is enough, but say so.

Per-group results, where they apply

If your model touches people, one overall number hides what matters. Report the metric separately by whatever groups your data supports — language, region, input length, customer type. A model at 0.89 overall may be at 0.93 on English and 0.71 on Hindi, and the second pair of numbers is the whole story for a service whose users are mostly Hindi speakers.

You can only report groups your data records. That is itself a finding worth stating: "we could not measure performance by region because region is not recorded" is honest and useful.

What regulation is starting to ask for

Rules are arriving at different speeds in different places. The European Union's AI Act introduces obligations that vary by risk category and by whether you are a provider or a deployer, with documentation and transparency duties among them. Other jurisdictions are drafting, consulting or doing nothing. Sectoral rules — medical devices, financial advice, employment decisions — often bite before any AI-specific law does.

The honest summary is that the obligations differ by where you are, what the system does, and who uses it, and that this is moving. **Nothing here is legal advice.** What generalises is the practice: a documented dataset, a stated intended use, a measured evaluation with per-group results, a named limitation and a way to contact somebody. A team that has those can answer whatever the specific requirement turns out to be. A team that has none of them is starting from nothing whenever somebody asks.

The test

Give the card to somebody who has not seen the project. Ask them: what does it do, what will it do badly, can they run it in ten minutes, and would they be

comfortable if it were used on them. Those four questions catch nearly everything.

Do this now

Open your best existing card and write the out-of-scope section, specifically, from your own testing. It is the section you can write today with information you already have, and it is the one most likely to stop somebody misusing your work.

BEFORE YOU MOVE ON

A card's limitations section reads "this model may reflect biases present in its training data". Why is this close to useless?

A. It admits a flaw without proposing a mitigation, which readers interpret as an unresolved defect.

B. Bias claims require statistical evidence, so an unquantified statement is not defensible under most documentation standards.

C. It is true of every model, so it transfers no information about this one.

D. It shifts responsibility onto the dataset's authors, which the model's publisher cannot do.

74 · 8 min

Pull requests, discussions and being useful

THE ONE THING TO KEEP

Any Hub repository accepts pull requests and proposals can be loaded with `revision="refs/pr/N"` before merging, so the highest-value contributions are small — a missing licence or chat template, a safetensors conversion, a card translation, a named limitation — and the discussions tab usually holds the documentation the card does not.

You can open a pull request on a model

Because a repository is git, anyone can propose a change to anyone's model. On the Hub these live as refs like `refs/pr/4`, so a proposal can be downloaded and tested before it is merged:

```
model = AutoModel.from_pretrained("some-org/some-model",
revision="refs/pr/4")
```

The contributions that are genuinely welcome and take under an hour:

- **A missing licence or `pipeline_tag`.** Without the tag the model is invisible to every filtered search, which the author usually does not realise.
- **A safetensors conversion**, so nobody has to unpickle a stranger's file.
- **A missing `chat_template`**, which is the difference between the model working and not.
- **A working code snippet** in the card, tested.
- **A translation of the card**, which for Hindi, Bengali, Tamil or Marathi genuinely widens who can use the model.
- **A named limitation you found**, with the inputs that produced it.

That last one is the most valuable and the least common. A card that says "fails on inputs over 400 tokens, see discussion #12" is worth more than a

benchmark table.

```
hf upload some-org/some-model ./README.md --create-pr
```

Discussions are where the real documentation lives

The Community tab on a model or dataset frequently contains the information the card does not: the actual prompt format, the version that broke, the workaround for a tokenizer bug, the licence clarification from the author. **Read the discussions before concluding a model does not work.** A large fraction of the problems in this course have a thread about them.

When you open one, make it answerable. A good report contains the model id with revision, library versions, the exact rendered prompt, the generation parameters, what you expected and what happened. Most unanswered threads on the Hub are missing the rendered prompt, and the rendered prompt is the answer surprisingly often.

The norms

The Hub is a public square with an unusually mixed population — researchers, hobbyists, students, companies. A few things hold:

- **Assume the maintainer is unpaid.** Most are. A model released for free carries no support obligation.
- **Report security issues privately first.** A malicious file or a leaked secret goes to the platform and the author, not to a public thread.
- **Do not reupload somebody's model under your own name.** Fine-tune it, quantize it, convert it — and say what it came from, with a `base_model` field and a link. Republishing weights as your own is both a licence problem and the thing that damages the commons everybody here depends on.
- **Credit the dataset** as carefully as you credit the model.

Publishing is contributing

The most useful things a learner can publish are not state-of-the-art models:

- **A quantized conversion** of a model that has none, done carefully and documented.
- **A Space** that demonstrates a technique with a readable `app.py`. People learn from working code far more than from documentation.
- **A dataset** in a language or domain that is thin on the Hub. Indic-language datasets are genuinely scarce and genuinely valuable, and a few thousand well-documented rows is a real contribution.
- **An evaluation** of existing models on a task nobody has measured, with the set published.
- **A card translation.**

Each of these is achievable in a weekend and each is more useful than another fine-tune of a popular model on a public instruction set.

The reciprocity that makes this work

Everything in this course exists because people published things they could have kept. The libraries are free, the models are free, the hosting is free, the compute for small things is free. That arrangement is not guaranteed and it is not charity — it is sustained by people putting back more than they take out.

You do not owe anyone a model. Opening one pull request that adds a missing licence tag takes ten minutes and makes a repository usable by everyone who finds it afterwards. That is a reasonable rate of exchange.

Do this now

Find one model you have used whose card is missing something you know — a licence, a template, a limitation, a working snippet. Open a pull request. It is the fastest way to stop being only a consumer of this ecosystem.

BEFORE YOU MOVE ON

Why does a model missing its `pipeline_tag` become effectively invisible on the Hub, even with many downloads?

A. Untagged repositories are excluded from the download counter, so their apparent popularity understates real use.

B. The Hub's full-text search indexes only repositories with complete metadata, so the card's prose is never matched.

C. Without a task tag it cannot appear in the controlled-vocabulary filters most people use to browse.

D. Models without a task tag are ranked below tagged ones in every sort order as a quality signal.

75 · 9 min

Still working next year

THE ONE THING TO KEEP

Four things move independently under a working system — the model's `main` branch, the libraries, the platform defaults and your own corpus — so pin every `from_pretrained` to a revision, pin dependency versions, read deprecation warnings as dated outage notices, and mirror any model you genuinely cannot lose.

Four things move under you

A system built on the Hub has four independent moving parts, and each has broken somebody's production:

1. **The model.** `main` gets a new commit. Weights change, a tokenizer is fixed, a template is corrected.
2. **The libraries.** transformers, torch, gradio and datasets all ship breaking changes between major versions.

3. **The platform.** Space SDK defaults, hardware tiers, provider availability.
4. **Your own data.** The corpus your vectors were built from drifts away from the vectors.

Pinning the first two is nearly free. The other two need a habit.

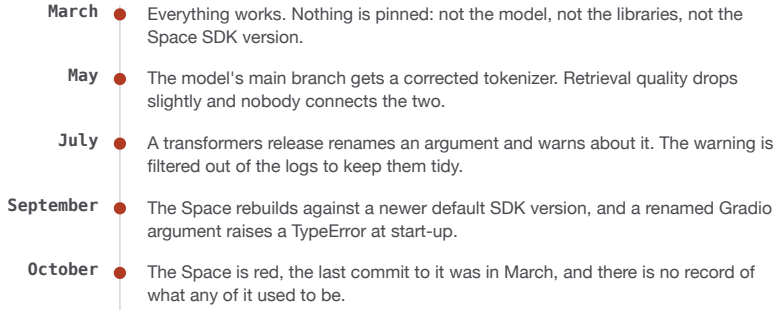
Pin the model

```
REV = "06f233fe06e710322aca913c1bc4249a0d71fce1"

model = AutoModelForCausalLM.from_pretrained(MODEL,
revision=REV)
tok = AutoTokenizer.from_pretrained(MODEL, revision=REV)
```

Copy the hash from the History tab. Pin **every** `from_pretrained` in the codebase, including the tokenizer and the processor — a tokenizer that updates independently of the weights is exactly the mismatch module two warned about.

A system that worked in March and failed in October



Four things move independently and none of them is your code. Pinning the model revision and the library versions is nearly free; the five-line record of what you pinned is what turns the next failure into debugging rather than archaeology.

Write the revision into your model card, your evaluation results and your logs. "Trained on Llama-3.1-8B-Instruct" without a hash is not reproducible, because that name has meant several different sets of bytes.

Pin the libraries

```
transformers==4.46.2
torch==2.5.1
datasets==3.1.0
gradio==5.9.1
accelerate==1.1.1
```

And pin `sdk_version` in a Space's front matter, and the Python version where it matters. Unpinned dependencies are the second-largest cause of a thing that worked in March failing in October with no commit in between.

Upgrade deliberately: read the release notes, upgrade one thing, run your thirty examples, commit. Upgrading five things at once and finding a regression means bisecting five variables.

The migration that will come

Transformers deprecates with warnings before it removes. `torch_dtype` became `dtype`; `evaluation_strategy` became `eval_strategy`; `huggingface-cli` became `hf`. Each was warned about for a release or more before the old form stopped working.

So: **read your warnings.** A deprecation warning is a dated notice that your code will break. Filtering them away, which many projects do to clean up logs, converts a warning into an outage later.

When the model you depend on disappears

Repositories get deleted, made private, or gated retrospectively. If your system cannot function without a specific model:

- **Mirror it** into your own private repository. This is permitted for models whose licence allows redistribution, and it is the reason licence checking matters operationally as well as legally.
- **Cache the weights** somewhere you control, not only in an ephemeral container.

- **Record the revision**, so you can identify exactly what you had.

For anything that matters, a model you did not mirror is a dependency on somebody else's continued goodwill.

Keeping up without drowning

The field moves fast enough that trying to follow all of it is a full-time job that produces no work. A sustainable version:

- **Check the trending tab monthly**, not daily. Anything that matters is still there a month later.
- **Follow the organisations** whose models you actually use, so you see their releases.
- **Read release notes** for the libraries you pin, on upgrade.
- **Re-run your thirty examples** against one new candidate model a quarter. This is the only method that tells you whether a new release matters *to you*, and it takes an afternoon.

What does not change: the Hub is git; a tokenizer must match its model; memory is parameters times bytes; padding side matters for generation; similarity is not relevance; fine-tuning teaches behaviour and retrieval supplies knowledge; a number without a baseline means nothing. Every specific tool in this course will be superseded and none of those will.

Re-embedding, the data-side drift

A vector index built in January against documents that changed in June is quietly wrong. Store the source document's hash alongside each vector, and re-embed the chunks whose hash changed. That is a scheduled job, not a rebuild, and it is the difference between a search that decays and one that does not.

The record that makes all of this possible

One file in your repository, updated when anything changes:

```
model      Qwen/Qwen2.5-7B-Instruct @ 06f233f
adapter    your-name/support-lora @ a91c4d2
embeddings BAAI/bge-small-en-v1.5 @ e8f0c1b, 384 dims, nor-
malised
libs       transformers 4.46.2, torch 2.5.1, peft 0.13.2
eval       eval/thirty.jsonl, last run 2026-09-14, 27/30
```

Five lines. It takes two minutes to write and it is the difference between debugging and archaeology.

Do this now

Go through your current project and pin every `from_pretrained` to a revision. Then write the five-line record above. You have just made everything you built this course reproducible, which is the thing that makes it worth having built.

BEFORE YOU MOVE ON

A team pins their model weights to a commit hash but loads the tokenizer with no revision. What can go wrong?

A. Nothing, since a tokenizer cannot change in ways that affect a pinned model's behaviour.

B. The tokenizer download will fail, because the Hub requires matching revisions across components of one repository.

C. Loading will raise a vocabulary-size mismatch, which makes the problem obvious rather than silent.

D. A later commit can change the chat template or special tokens, so prompts differ while the weights stay fixed.

CASE STUDY · MODULE 8

Three weeks before launch, the base was not theirs to sell

A four-person design studio in Ahmedabad, three weeks from launching a paid tool that cuts out and relights product photographs for small online sellers.

The model at the centre of the product was a fine-tune published by an individual. Four hundred thousand downloads, a readable card, a demo Space that worked on a phone, and a licence tag reading apache-2.0. The studio had built on it for four months.

The tool itself was small and good. A seller photographs a kurta on a bedsheet in a room with one window, and gets back a clean cut-out on a white background with the light corrected. Two hundred rupees a month, aimed at people selling forty items rather than four thousand.

The check happened by accident. A prospective customer's procurement form asked for a statement of every third-party component and its licence, with a signature underneath.

Ritu, the partner filling it in, opened the model page and followed the front matter. The fine-tune named a parent. The parent named another parent. The third repository, with no parent of its own, was under a non-commercial research licence.

The apache-2.0 tag at the top of that chain was either a careless tick or a sincere belief that an adapter is a separate artefact. It made no difference which. A fine-tune inherits its base model's terms whatever its own card says, and the studio was three weeks from charging money for a derivative of a model that may not earn money.

The decision

Swapping meant a permissively licensed alternative. Two existed. On the studio's thirty test photographs — textiles, brass and steel, shot on phones in the

back rooms of Gujarati sellers, which is the case no public benchmark contains — the better of the two scored four points worse, and the loss was concentrated in the hard cases: fine fabric edges, and hair on a model's forearm.

Launching anyway meant shipping. Nobody would notice on day one. The uploader probably did not know either. The risk is not a letter in the first week; it is a customer's own procurement form asking the same question in a year, when the tool sits inside forty shops' workflows and the answer has to be no.

They did write to the uploader to ask. Eight days, no reply, which is what an unpaid maintainer of a popular repository usually has time for.

So the choice was four quality points and eight days of schedule, against a dependency they could not answer a question about, in a product other people were about to put in their shop window.

The studio had done a version of this check at the start, which is why the mistake is worth studying rather than dismissing. Four months earlier somebody had opened the model page, seen apache-2.0 in the sidebar, and written it in a notebook. That is the check most people perform and it is the wrong one, because the tag describes an intention and the chain describes an entitlement. Nothing about the page announces that the badge at the top is only the first link.

Ritu also asked what the honest fallback would have been if no permissive alternative had existed at any quality. The answer they arrived at was to charge for the part they had made rather than the part they had borrowed — the preprocessing, the batching, the seller-facing workflow — and to let the customer bring their own model or run the non-commercial one themselves for their own use. It was not needed. Writing it down took ten minutes and it turned an argument about whether to ship into an argument about which of two lawful products to ship, which is a much easier argument to have three weeks before a launch.

STOP HERE

Before turning the page: what would you have done, and what would it have cost if you were wrong? Write it down. The point of the next section is to compare.

WHAT HAPPENED

They swapped, and launched eight days late.

The four-point gap turned out to be two once they spent a day on the pre-processing step. The permissive model was more sensitive to input resolution, which is a fixable property rather than a capability gap, and they only found that because thirty photographs is few enough to sit down and look at all eleven failures one at a time.

Three practices came out of that fortnight and stayed. A two-line inventory per model, recording repository, revision hash, licence, the date it was read and where it runs. A rule that licence checking means following the base model field until a repository has no parent, rather than reading the badge on the page. And a card on the small pre-processing Space they published themselves, saying what it was built from, what it does badly, and that the licence of their code is not the licence of the model inside it.

The launch slipped eight days. The inventory takes about a minute per model and has been consulted twice since, both times by somebody else's procurement form.

What does an apache-2.0 tag on a fine-tune actually settle, and what does it not?

It states what the uploader intends to grant over their own contribution, and it is indexed by the Hub so people can filter on it. It cannot grant rights the uploader never held. A fine-tune is a derivative of its base, so the base's terms travel with it however the child is tagged, and the check is to follow the base model field until there is no parent. It also says nothing about the training data or about who owns the outputs, both of which are genuinely unresolved and differ by country.

Why could thirty photographs of their own decide the swap when a leaderboard could not?

Because the thirty contained the case the studio is paid for: phone photographs of textiles and brass in poor light, which no public benchmark includes. A leaderboard rank is measured on somebody else's distribution and narrows four hundred candidates to a shortlist; it cannot pick between two shortlisted models for a specific job. It also gave them failures to read, and reading the eleven failures is what found the resolution problem.

The four-point gap became two after a day of work. What does that say about deciding between models on a single number?

That a headline score bundles the model with everything around it — pre-processing, prompt, generation settings, input resolution — and a gap is not evidence of which part is responsible until you look. Reading the failures separated a fixable input problem from a capability difference. At thirty items a four-point gap is also close to the noise floor, so acting on it as a capability judgement, in either direction, would have been unsound.

MODULE 8 · TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record: answers and the reason behind each are at the back. If two questions from the same module go wrong, re-read that module before the exam rather than after.

1 Your Space gets a 403 on a gated model whose terms you accepted in a browser. What is wrong?

A. The token has expired and a new one has to be issued from settings.

B. Gated repositories can never be loaded from a Space under any circumstances at all.

C. Access is granted per account, and the Space's token belongs to another one.

D. The repository was made private again after you accepted its terms.

2 A model is published under Apache 2.0. Which question does that licence actually settle?

A. Whether the material it was trained on was lawfully obtained.

B. What the publisher grants you over the weights they hold.

C. Who owns the text or images that the model generates for you.

D. Whether your particular application complies with an acceptable-use policy.

3 At 100,000 tokens a day for a small open model, which is cheaper, and why?

A. A rented GPU, because per-token billing carries a high monthly floor.

B. Neither; providers price their tokens directly against GPU-hour costs.

C. A rented GPU, because providers apply a minimum charge per account.

D. Serverless, because a rented card bills for every idle hour as well.

4 A transformers loop behind a web framework sits at 30 per cent GPU utilisation with ten concurrent users. Which change addresses the cause?

A. A server with continuous batching, so new requests join a running batch.
B. Raising the batch size inside the loop until the utilisation figure climbs.
C. Quantizing to four bits with bitsandbytes for extra throughput.
D. Moving to a larger card with more memory bandwidth available.

5 Model A scores 27 of 30 and model B scores 26 of 30 on your own evaluation set. What may you conclude?

A. A is better on your task and should be adopted over B.
B. The set is unreliable and ought to be rewritten from scratch.
C. Nothing: one item is well inside the noise at this sample size.
D. B is better, because it was run second and had the harder conditions.

6 Which pair, pinned, removes most of the failures where something worked in March and broke in October?

A. The hardware tier and the Space's configured sleep timer.
B. The tokenizer choice and the model's published generation configuration.
C. The dataset revision and the random seed used during training.
D. The model revision and the versions of the libraries you install.

EXAMINATION

Hugging Face, End to End — final examination

This paper covers the whole course: reading a repository and its licence, tracing text through the transformers library, fitting a model to the hardware in front of you, building and debugging a Space, working with datasets far larger than your RAM, building and measuring retrieval, adapting a model with LoRA or QLoRA, and depending on all of it without being surprised later. Twenty-five questions are drawn each time from a larger pool, so a retake is a different paper. The pass mark is 70 per cent. There is no timer, because a clock measures panic as well as understanding, and many people here are reading in a second or third language. If you do not pass, you may retake it after thirty minutes with fresh questions, as many times as you need. A teacher can open the next course for you regardless of this paper.

On the website a sitting is 25 questions drawn from this pool and the pass mark is 70%. There is no time limit, there and here. Passing opens the next course in the track.

1 `1. The Hub, read properly`

You want to know what a large repository will cost in disk before you fetch anything. Which answer is reliable?

A. Add up every one of the file sizes shown in the Files and versions listing on the page.

B. Multiply the parameter count in the repository's name by two bytes each.

C. Read total_size in the safetensors index, or call model_info with files_metadata.

D. Check the download counter, which scales with how large the weights are.

2 1. The Hub, read properly

A repository ships both .bin and .safetensors copies of the same weights. What does a naive snapshot_download do?

A. Fetches both, doubling the download; allow_patterns takes only what you need.
B. Fetches the safetensors copy only, because that format is always preferred by default.
C. Fetches whichever copy is newer according to the repository's commit history.
D. Fails with an error, because two formats of one model cannot both be cached.

3 1. The Hub, read properly

Why can Hub search fail to return the most-used model for a task even when that model plainly exists?

A. Search only indexes repositories updated within the last twelve months.
B. Search matches repository names and card text rather than what a model does.
C. Search ranks by likes, and heavily used models attract relatively few likes.
D. Search excludes any repository whose card omits a pipeline_tag in front matter.

4 1. The Hub, read properly

You click Agree and access repository on a gated model. What has that settled?

A. That the model is licensed permissively, because access was granted at all.
B. That your organisation's accounts now have access to the same repository too.
C. That you may download the files; the licence still governs what you do next.
D. That the terms have been accepted for every account you use on the platform.

5 1. The Hub, read properly

Which facet of the dataset filters is computed automatically rather than claimed by the uploader?

A. The language tags, which the viewer detects from a sample of rows.
B. The licence, which the Hub verifies against the LICENSE file in the repository itself.
C. The task category, drawn from a controlled vocabulary the Hub maintains.
D. The size band, which is populated from the dataset viewer rather than typed.

6 1. The Hub, read properly

A repository holds adapter_config.json and a forty-megabyte adapter_-model.safetensors, and nothing you try will load it. Why?

A. The weights are sharded and the index file is missing from the repository.
B. It is a LoRA adapter and needs the base model named inside adapter_config.json.
C. Forty megabytes is a truncated upload, so the repository is simply incomplete.
D. Adapters can only be loaded through the Hub's inference API, never through transformers.

7 1. The Hub, read properly

One model has 3,000 likes and 400 downloads. Another has 40 likes and 900,000 downloads. What distinguishes them?

A. The first is newer, because likes accumulate before downloads begin.
B. The second is better, because downloads cannot be produced by one person.
C. The first is famous and the second is load-bearing; they measure different things.
D. The first was promoted on the trending tab, and the second one never appeared there at all.

8 2. Transformers, from characters to output

You want one vector per token to feed into a model of your own. Which class gives you that?

A. AutoModelForFeatureExtraction, which is built for exactly this purpose.
B. AutoModelForSequenceClassification, reading logits before the softmax step.
C. AutoModel, which loads the body and returns last_hidden_state.
D. AutoConfig, which exposes the hidden states declared in the configuration.

9 2. Transformers, from characters to output

A classifier returns LABEL_0 and LABEL_1. What is the risk, and where does the meaning live?

A. The head is randomly initialised, so the labels have no meaning at all yet.
B. The mapping was never filled in; guessing polarity can invert a whole dataset.
C. The model is multilingual, so labels are numbered rather than named by design.
D. The tokenizer and the model disagree, which is why the labels lost their names.

10 2. Transformers, from characters to output

You render a chat template with tokenize=False and then encode the resulting string with default settings. What can go wrong?

A. The template's Jinja is re-evaluated, so the roles are rendered a second time.
B. The tokenizer adds a second beginning-of-sequence token, shifting every offset.
C. The special tokens are escaped as text, so the model sees literal angle brackets.
D. Nothing, because encoding a rendered template is the recommended path.

11 2. Transformers, from characters to output

A classifier reads three-thousand-word complaints and performs badly. Which single change is most likely to help?

A. Raise the batch size, so more of each complaint is processed at once.
B. Turn truncation off, so the model receives the whole complaint every time.
C. Set truncation_side to left, so the end of the complaint survives the cut.
D. Lower the temperature, which makes a classifier more decisive on long text.

12 2. Transformers, from characters to output

Which change most improves throughput on a batch of mixed-length inputs, at no cost in quality?

A. Sort by length before batching, then restore the original order afterwards.
B. Raise the batch size until the card's memory is fully occupied by the work.
C. Set padding to max_length, so every batch has exactly the same tensor shape.
D. Disable the attention mask, so padded positions no longer have to be computed.

13 2. Transformers, from characters to output

How does TextIteratorStreamer deliver tokens to a web application as they arrive?

A. generate yields tokens directly, and the streamer simply formats each one.
B. The model is run once per token, so each call returns a single new piece.
C. The streamer polls the model's internal cache at a fixed interval and reads it.
D. generate runs on a second thread and fills a queue the main thread consumes.

14 2. Transformers, from characters to output

A nine-hundred-token prompt with max_length=1024 returns an answer cut off mid-sentence. What is the fix?

A. Raise max_length repeatedly until the model stops cutting the answer short each time.

B. Use max_new_tokens, which counts only the output rather than prompt plus output.

C. Set min_new_tokens as well, which forces the model to finish its sentence.

D. Add a stop string, so generation ends cleanly at a punctuation boundary.

15 3. Running it on the hardware you actually have

A model loads happily and then runs out of memory partway through a long generation. What is the usual cause?

A. The weights are reloaded for each token when device_map auto is in use.

B. Framework overhead grows steadily as more requests pass through the process.

C. The KV cache, which grows with sequence length and with batch size.

D. Activation memory, which is fixed by the architecture and cannot be reduced.

16 3. Running it on the hardware you actually have

What does a single GGUF file contain that a transformers repository spreads across several files?

A. Weights, tokenizer, architecture metadata and the chat template, together.

B. Weights and a training log, so the conversion can be reproduced by anyone.

C. Weights only, with everything else fetched from the Hub when it is first run.

D. Weights and the ONNX graph, which is why it runs without PyTorch installed.

17 3. Running it on the hardware you actually have

You load a model in four bits with bitsandbytes on a card that had room for bfloat16, and inference is slower. Why?

A. Four-bit kernels are only implemented for the newest generation of hardware.

B. The quantized weights are dequantized on the fly for each matrix multiply.

C. Four-bit models have more layers, because each one holds fewer parameters.

D. The model is being partly offloaded to the CPU, which the config does silently.

18 3. Running it on the hardware you actually have

How can a static Space with no server process be a complete, working AI application?

A. The Hub runs the model for it and returns results to the page behind the scenes.

B. The page calls a free inference provider, which is why no server is needed.

C. An ONNX model downloads into the browser and runs in WebAssembly or WebGPU.

D. Static Spaces are given a hidden CPU container that the platform manages.

19 3. Running it on the hardware you actually have

Every free GPU session ends. Which discipline actually makes them usable?

A. Keep the browser tab active, so the session is not judged to be idle.

B. Work on the largest model that fits, so that fewer sessions are needed in total overall.

C. Install everything from source, so the environment is identical each time.

D. Push artefacts to the Hub, checkpoint often, and debug on a tiny model first.

20 3. Running it on the hardware you actually have

A distilled speech model is roughly six times faster with a word error rate within about a point. What is the caveat?

A. Distilled models cannot be fine-tuned, because the teacher is unavailable.
B. That figure holds for the data it was distilled on; check on your own examples.
C. The speed-up applies on a GPU only, and disappears entirely on a processor.
D. The published comparison used a different decoding strategy from the original model's.

21 3. Running it on the hardware you actually have

An ONNX export completes with warnings. What must you do before relying on it?

A. Re-export on a machine with more memory, which removes the warnings.
B. Nothing; the exporter refuses to write a file when the graph is not equivalent.
C. Compare the exported model's logits against PyTorch within a stated tolerance.
D. Run the ONNX model twice and check that both runs produce the same output.

22 4. Spaces: putting something in front of people

Which single gr.Interface argument most changes how many strangers actually try your Space?

A. examples, because a visitor faced with an empty box usually leaves.
B. title, because it is what appears on the gallery card in search results.
C. flagging_mode, because it decides whether visitors can report bad output.
D. description, because it is where the model's limitations are explained clearly.

23 4. Spaces: putting something in front of people

What happens when the duration on a ZeroGPU-decorated function is set lower than the work needs?

A. The function queues until a longer allocation becomes free on the shared pool.
B. The allocation is extended automatically and the extra time is billed to quota.
C. The call is killed partway through, so the generation never finishes.
D. The function falls back to the CPU and completes, much more slowly than usual.

24 4. Spaces: putting something in front of people

A Space sleeps often and takes three minutes to wake. What is the best use of a small paid persistent volume?

A. Store the visitors' uploads, so a restart does not lose work in progress.
B. Store the model cache, so waking loads weights from the volume, not the network.
C. Store the build image, so the container does not have to be rebuilt each time.
D. Store the application logs, so that failures from before the sleep are still readable.

25 4. Spaces: putting something in front of people

A Docker Space builds, starts, and shows a blank page forever. Which two mistakes cause that?

A. A missing app_file field, or a Dockerfile that never copies the source in.
B. Running as root, or forgetting to expose a port in the Dockerfile at all.
C. Listening on a port other than 7860, or binding to localhost instead of 0.0.0.0.
D. An image that is far too large, or a base image the platform does not recognise at all.

26 4. Spaces: putting something in front of people

Somebody duplicates your Space, which reads an API token from its environment. What happens to your secret?

A. It is copied, so you should rotate the token whenever anyone duplicates.
B. It is copied but masked in the interface, and readable from the duplicate's code.
C. The duplicate fails to build, because a missing secret stops the container.
D. It is not carried over; the duplicate is prompted to supply its own value.

27 4. Spaces: putting something in front of people

Why does a CommitScheduler example take a lock before appending a line to its file?

A. Because the scheduler rewrites the file on each push and would drop the line.
B. Because concurrent requests interleave partial lines and corrupt the JSON.
C. Because the Hub rejects a commit made while another commit is in flight.
D. Because the filesystem is read-only except during the scheduler's own window.

28 4. Spaces: putting something in front of people

What does naming an event handler with api_name buy you, and what does api_name=False do?

A. It publishes the endpoint; False keeps it internal and out of the API listing.
B. It enables the endpoint, since without a name no handler is callable from outside at all.
C. It documents the endpoint; False only hides it from the Use via API page.
D. It caches the endpoint's results; False disables caching for that handler.

29 5. Datasets, and the work that happens before any model

In the datasets library, what do train[0] and train[:3] return?

A. A row object and a list of row objects, both carrying the feature types.

B. A dict for the row, and a dict of lists for the slice.

C. A tuple of values and a list of tuples, in the order of the features.

D. A dataframe row and a dataframe, because slicing converts to pandas.

30 5. Datasets, and the work that happens before any model

Your positive class is two per cent of rows. Why does train_test_split need stratify_by_column?

A. Because the trainer will refuse a split whose class proportions are noticeably unbalanced.

B. Because stratification oversamples the rare class, which fixes the imbalance.

C. Because an unstratified split can leave almost no positives, so recall measures nothing.

D. Because without it the split is not reproducible between runs of the script.

31 5. Datasets, and the work that happens before any model

Your map function reads a threshold from a module-level global. You change the threshold and rerun. What happens?

A. The cache is invalidated correctly, because the library hashes the whole module file.

B. The map raises, because functions that close over globals cannot be hashed.

C. The map reruns, but the old cache file remains and fills the disk silently.

D. You get the previous cached results, because the hash covers bytecode, not globals.

32 5. Datasets, and the work that happens before any model

Why is set_transform rather than map the right tool for image augmentation?

A. map cannot operate on Image columns at all, because they decode lazily on access.

B. map would freeze one random draw into a cache and train on it every epoch.

C. map runs on a single process, so augmentation would become the bottleneck.

D. map writes uncompressed arrays, which would be too large to keep on disk.

33 5. Datasets, and the work that happens before any model

A telephone corpus is at 8,000 Hz and your speech model expects 16,000. What does cast_column to Audio at 16,000 achieve?

A. It resamples lazily on access, making the input well formed for the model.

B. It reconstructs the missing high frequencies, so accuracy matches wide-band audio.

C. It rewrites the stored files at the new rate, which doubles the dataset on disk.

D. It tells the model to expect 8,000 Hz input, so no resampling is needed at all.

34 5. Datasets, and the work that happens before any model

You tokenise a dataset and then call push_to_hub. What have you published?

A. The original text, because the library stores transformations separately.

B. The text and the token ids, since map adds columns rather than replacing them.

C. Token ids for one tokenizer, which nobody with a different model can use.

D. Nothing usable, because tokenised datasets cannot be uploaded to the Hub.

35 5. Datasets, and the work that happens before any model

A dataset card reports inter-annotator agreement of 78 per cent. What does that figure tell you before you train?

A. That the dataset needs at least a third annotator before it can be used.
B. That roughly 78 per cent of the rows are usable and the remainder should be dropped.
C. That a model will reach about 78 per cent accuracy after enough epochs.
D. That measured accuracy has a ceiling near there, because the labels disagree.

36 6. Embeddings, and search that understands

Why should you not build sentence embeddings yourself from a base model's token vectors?

A. Because base models produce token vectors in a different numeric range.
B. Because the pooling has to match what the model was trained with, or it misleads.
C. Because token vectors are not normalised and cannot be averaged meaningfully.
D. Because a base model's vocabulary differs from that of any dedicated embedding model.

37 6. Embeddings, and search that understands

Why do 'the battery lasts all day' and 'the battery does not last all day' sit close together in most embedding spaces?

A. Because negation is handled by the reranker rather than by the embedding model.
B. Because both sentences are short, and short texts cluster near the space's centre.
C. Because the texts are nearly identical and one small token carries the meaning.
D. Because the model normalises the vectors, which removes the sign of the difference.

38 6. Embeddings, and search that understands

A tutorial says to treat anything above 0.8 cosine similarity as a match. Why is that unsafe advice?

A. Because cosine similarity is undefined for vectors that have been normalised.

B. Because thresholds must always be set higher for multilingual models than for English ones.

C. Because similarity should be computed with dot product once vectors are normalised.

D. Because each model spreads its space differently, so a threshold does not transfer.

39 6. Embeddings, and search that understands

You truncate 768-dimensional vectors to 256 to save memory. When is that safe?

A. Only for models trained so the first dimensions are usable on their own.

B. Whenever the vectors are normalised, because truncation preserves the angle.

C. For any model, as long as you truncate the queries and documents alike.

D. Only when the index is exact, since approximate indexes need full vectors.

40 6. Embeddings, and search that understands

Top-k retrieval keeps returning five nearly identical chunks. Which technique addresses that, and why does it matter?

A. A larger k, so that more distinct material appears further down the list.

B. Maximal marginal relevance, which trades relevance against redundancy.

C. A cross-encoder reranker, which scores each candidate against the query.

D. Deduplicating the query, since repeated terms attract repeated passages.

41 6. Embeddings, and search that understands

Two models both produce 512-dimensional vectors. What happens if you put both sets in one index?

A. Search degrades gradually, in proportion to how different the two models really are.

B. The index refuses the second set, because the model name is stored with it.

C. The arithmetic runs and returns confident nonsense, because it is not one space.

D. Results stay correct for the majority model and become random for the other.

42 6. Embeddings, and search that understands

Why does changing the embedding model cost more than changing almost anything else in a retrieval system?

A. Because every stored vector becomes worthless and the corpus must be re-encoded.

B. Because the chunk boundaries all have to be recomputed for the new tokenizer first.

C. Because the reranker has to be retrained to match the new vector geometry.

D. Because the evaluation questions are model-specific and must be rewritten.

43 7. Training and adapting, with the libraries the Hub is built on

per_device_train_batch_size is 4 and gradient_accumulation_steps is 8, on one device. What does that buy and what does it cost?

A. An effective batch of 32 with the memory of 4, at the cost of wall-clock time.

B. An effective batch of 32 with the memory of 32, spread across eight steps.

C. An effective batch of 12, at the cost of a slightly noisier gradient estimate.

D. An effective batch of 4 repeated eight times, which is equivalent to more epochs.

44 7. Training and adapting, with the libraries the Hub is built on

What does gradient_checkpointing=True trade, and roughly how much?

A. It halves the model's precision to save memory, at some cost in quality.
B. It stores activations on the CPU instead of the GPU, at the cost of transfers.
C. It recomputes activations in the backward pass, costing roughly a fifth to a third of speed.
D. It skips writing checkpoints during training, saving disk space and about a tenth of the time.

45 7. Training and adapting, with the libraries the Hub is built on

Why does changing LoRA's r without changing lora_alpha produce inconsistent results?

A. Because alpha sets the dropout rate, which must scale with the rank.
B. Because the update is scaled by alpha over r, so the effective step size moves.
C. Because alpha names the target modules, which differ by architecture.
D. Because r and alpha have to be equal for the low-rank decomposition to remain valid.

46 7. Training and adapting, with the libraries the Hub is built on

merge_and_unload folds an adapter into the base weights. What do you give up?

A. The ability to serve several adapters over one loaded base, and a small file.
B. The ability to continue training, because merged weights are frozen afterwards.
C. Compatibility with transformers, since merged models need a separate loader.
D. The record of the base model, which is only stored in adapter_config.json.

47 7. Training and adapting, with the libraries the Hub is built on

packing=True concatenates short conversations into full-length blocks. What has to be handled for that to be correct?

A. The padding token must be redefined, because packed blocks have no padding.
B. Attention masking between packed examples, so one cannot attend to another.
C. The learning rate must be lowered, because each step now sees more tokens.
D. The labels must be shifted by one, because block boundaries break the offset.

48 7. Training and adapting, with the libraries the Hub is built on

Your session died mid-run. Why is resume_from_checkpoint better than loading the saved weights and starting again?

A. Because loading the weights alone drops the tokenizer, which must match the run exactly.
B. Because the checkpoint is compressed and only the resume path can read it.
C. Because a restart reshuffles the data, so some examples are seen twice.
D. Because it restores optimiser state and the scheduler position, not only weights.

49 7. Training and adapting, with the libraries the Hub is built on

Why publish a LoRA adapter rather than a merged republication of the base model's weights?

A. Because a merged model cannot be licensed, while an adapter always can be.
B. Because merged weights lose the base's tokenizer and are therefore unusable.
C. Because it is honest about what you made, small to store, and names its base.
D. Because the Hub rejects uploads of weights that another account already hosts.

50 8. Depending on it, and publishing so others can

Why should a fine-grained token be the default rather than a read or write token?

A. Because fine-grained tokens are the only kind that work from inside a Space.
B. Because it can be scoped to named repositories, so a leak is bounded.
C. Because read and write tokens expire, while fine-grained tokens do not.
D. Because the Hub rate limits read and write tokens more aggressively.

51 8. Depending on it, and publishing so others can

You think a token might have appeared in a screenshot. What is the correct response?

A. Check whether the screenshot was deleted before anyone is likely to have seen it.
B. Rely on the platform's leak scanners, which revoke exposed tokens automatically.
C. Rotate it at the next convenient point, since a screenshot is low-risk exposure.
D. Revoke and reissue it, which takes thirty seconds and needs no judgement.

52 8. Depending on it, and publishing so others can

What is benchmark contamination, and which sign points to it?

A. Test answers reaching pretraining data; a gap between a famous and a fresh benchmark.
B. A benchmark whose questions were written by the same lab that built the model.
C. A test set with duplicated items, which inflates a score by repetition alone.
D. A leaderboard that ranks models trained on very different amounts of compute alongside each other.

53 8. Depending on it, and publishing so others can

A model ranks highly in a human preference arena. What has been measured?

A. Accuracy on the tasks that arena users most commonly bring to a model.

B. Likeability under one style of interaction, including formatting and length.

C. Agreement with expert judgements, since arena voters are screened for expertise.

D. Instruction-following, since voters compare answers to the same instruction.

54 8. Depending on it, and publishing so others can

Every request in your application carries the same long system prompt. Which serving feature helps most?

A. Continuous batching, which lets new requests join a batch already running.

B. Paged attention, which stops the key-value cache fragmenting its memory.

C. Prefix caching, which computes the shared opening once and reuses it.

D. Speculative decoding, which drafts tokens with a smaller model first.

55 8. Depending on it, and publishing so others can

How does quantization behave differently under a purpose-built serving stack compared with training-time bitsandbytes?

A. It loses less quality, because serving stacks calibrate on your own traffic.

B. It behaves identically; the difference is only in how the weights are stored.

C. It applies to the activations as well as the weights, which is why it is faster there.

D. It buys speed as well as memory, because the kernels are written for the format.

56 8. Depending on it, and publishing so others can

Somebody has opened a pull request against a model you depend on. How do you evaluate it before it is merged?

A. Clone the fork that the contributor pushed from, and then build the model locally.

B. Load it with revision set to refs/pr/N, which resolves the proposal as a ref.

C. Wait for the maintainer to merge, since proposals are not downloadable.

D. Copy the changed files out of the diff view and apply them to your own copy.

Answers

Each entry gives the correct option, then what each wrong one reveals about how somebody was thinking. The second part is worth more than the first: a wrong answer here is a named misreading, not a mistake.

Lesson checks

1. The Hub is git, and that explains everything else — C

A — Assumes an access control would hand back a stub file rather than refuse the request outright.

B — Reads a consistent, exactly-sized text file as a partial binary download.

D — Blames repository versioning for a checkout that reproduced exactly what was committed.

2. The search box is the worst way in — B

A — Relies on a naming habit rather than indexed metadata; many conversions use different suffixes, and name search will silently miss them.

C — Finds popular quantized models in general, not conversions of the specific model already chosen, and excludes AWQ and GPTQ formats entirely.

D — Depends on the uploader writing prose rather than on the structured base_model field the Hub actually indexes.

3. Read the licence before you read the demo — B

A — Treats training as an act that resets the terms attached to what was trained on.

C — Confuses the access gate with the licence, when the gate only controls who gets the files.

D — Assumes an unclear licence resolves in the user's favour rather than removing permission.

4. A model's name is a specification, if you can read it — C

A — Invents a restriction; MoE weights quantize with the same schemes as dense weights.

B — Multiplies the name out literally; the experts share layers, so the true total is about 47B rather than 56B.

D — Imagines instruction tuning adds weights; it changes the existing ones and adds only a template.

5. What the files in a model repository are for — B

A — Assumes safetensors files are compressed; they store raw tensor bytes with a JSON header and no compression.

C — Treats a routine duplicate-format layout as corruption, when a stale index would cause load failures rather than a size discrepancy.

D — Plausible-sounding but rare in published inference repositories, and optimiser state would be named as such rather than as model weights.

6. Loading a model can run someone else's code — C

A — Overstates the scanner, which targets pickle opcode streams and is heuristic rather than a guarantee for arbitrary Python.

B — Misreads the format; the header is parsed as data, and there is no evaluation step for it to exploit.

D — Invents an interaction between two independent options.

7. Four shelves, and picking the wrong one costs a weekend — D

A — Weighs convenience and hardware limits while ignoring that uploading confidential files is itself a disclosure.

B — Solves for cost and still sends the confidential images to a third party.

C — Treats a dedicated rented container as equivalent to the files never leaving her machine.

8. Ask the Hub questions instead of browsing it — B

A — Confuses repository storage with the current revision's file listing, which is what siblings describes.

C — Deduplication affects upload and storage, not the bytes a fresh client must receive.

D — Assumes a repository-wide aggregate when the call is scoped to one revision.

9. What a download count is counting — C

A — Inverts the inflation problem: downloads are the counter most easily produced by automation, while Spaces and fine-tunes require effort.

B — Treats download volume as a proxy for reliability when 260 running Spaces already demonstrate B loads in many configurations.

D — Declines a reading the evidence supports, and defers to leaderboards that measure neither adoption nor fit.

10. One line that hides five steps — A

B — Invents language routing; the pipeline applies one fixed model to whatever it is given.

C — Reaches for a plausible-sounding low-level failure when the cause is simply a model applied outside its training distribution.

D — Correctly notes that scores are not calibrated probabilities, but wrongly concludes the English result carried no information.

11. AutoModel gives you numbers, not answers — C

A — Blames the pretraining objective; those representations are exactly what fine-tuning uses successfully.

B — An inverted mapping would produce accuracy near zero rather than near chance, and would not explain the warning.

D — Mismatches head to task; masked LM predicts vocabulary tokens, not class labels.

12. Your text is integers, and the count is not the word count — B

A — Imagines an internal pivot-translation stage that these models do not perform.

C — Confuses an encoding detail with vocabulary construction; tokenizers work over bytes and learned merges, not fixed-width characters.

D — Attributes to user behaviour a gap that persists when message lengths are held constant, as they are here.

13. Padding on the wrong side ruins generation quietly — D

A — A real bug, but it would also degrade batched encoder classification, and here the model generates from the wrong final position.

B — Treats a batch as one long sequence; items in a batch each get their own context window.

C — Sampling variance would produce different good answers, not systematically vague ones for short prompts.

14. Logits, and the four knobs between them and words — B

A — Would explain identical output, but a fixed seed is an unusual default and does not explain output that looks deterministic and safe.

C — Invents a clamp; values above 1 flatten the distribution and increase variety rather than removing it.

D — Assumes a dependency between parameters; temperature alone still defines a full distribution to sample from.

15. The chat template is not optional — A

B — A system message shapes style, but its absence does not make a model impersonate the user.

C — Would produce an answer followed by extra turns, rather than no answer at all.

D — A genuine bug with a similar smell, but it produces stray markers in output rather than a clean user-style turn on a first message.

16. Throughput and latency are different problems — C

A — Invents silent splitting; exceeding memory raises an out-of-memory error rather than degrading quietly.

B — Describes a scheduling effect that does not change total compute, which is what a throughput shortfall measures.

D — Fast tokenizers do parallelise, and tokenization is rarely dominant against model forward passes.

17. Showing tokens as they arrive, and making them stop — B

A — Confuses a cap with a stop condition; the limit truncates rambling but does not cause it.

C — Treats continuation past the answer as a repetition problem, which sampling would address.

D — Blames display buffering for content the model actually generated.

18. It loaded, it ran, the output is nonsense — B

A — Invents a kernel restriction; half precision is well supported for these architectures.

C — Names the right precision pair but the wrong mechanism: the problem is exponent range and overflow to NaN, not underflow of small weights.

D — Tokenization is device-independent and happens before anything reaches the accelerator.

19. Your disk fills up before your patience does — C

A — Reaches for a general desktop behaviour instead of the cache's own layout.

B — Assumes an active repair process rather than a passive cache that only fetches when asked.

D — Invents weight sharing between separate model files instead of the deduplication the cache actually does at the blob level.

20. The memory arithmetic you can do in your head — A

B — Invents a promotion step; weight dtype is fixed at load and does not change with traffic.

C — Names a real consumer, but activations are transient per forward pass and do not accumulate across a conversation.

D — Fragmentation is real but secondary here, and the shapes are not identical — conversations are getting longer.

21. Spreading a model across whatever you have — C

A — Describes multi-GPU behaviour; with a single GPU there is no pipeline to serialise.

B — Would slow things by a factor of a few, not thousands, and dtype comes from the load call rather than the map.

D — Plausible in shape, but the cache stays with its layer's device rather than being offloaded independently by default.

22. Quantization: what you buy and what you pay — D

A — Quantization touches weights only; the tokenizer is unaffected by precision.

B — Templates live in the tokenizer config and are usually carried over; this would also degrade chat quality, which held up.

C — Assumes quantization shortens output, and misplaces the fault in the parser rather than in generation fidelity.

23. GGUF, and the laptop that has no GPU — C

A — Names a check that would catch the error rather than prevent it, and is not what makes the format robust.

B — Anyone can convert and upload GGUF files, which is exactly why conversion provenance matters.

D — Invents a byte-level runtime; GGUF models tokenize exactly as their originals do.

24. ONNX, Optimum, and models that run in a browser tab — B

A — WebGPU improves speed but WebAssembly runs the same models, more slowly.

C — Export availability matters and can be produced with Optimum; the ownership restriction is invented.

D — Batching helps throughput but is not the gate on feasibility for short interactive inputs.

25. Where the free GPUs actually are — B

A — Trades away the work rather than protecting it, and no free session length is guaranteed in the first place.

C — Helps with files but does not by itself produce checkpoints at intervals or a resume path.

D — Longer sessions reduce the frequency of the problem without addressing the loss when it happens.

26. What actually runs on eight gigabytes — C

A — Sampling can be disabled, so determinism is not what separates the two here.

B — Overextends a real point about quantization; short label outputs are not where the damage concentrates.

D — Unused context costs nothing; compute scales with the tokens actually present.

27. Proving the change helped — A

B — Describes a real distortion that pushes timings the wrong way — slower, not faster — and is a separate issue from synchronisation.

C — Invents implicit batching across independent calls.

D — Conflates memory reporting with wall-clock timing; the counter and the clock are unrelated.

28. A Space is someone else's computer, already configured — A

B — Attributes both symptoms to a hardware downgrade when a start-up delay and a queue position have separate causes.

C — Reads a shared serving queue as a per-user limit applied to you.

D — Assumes weights are fetched per request rather than loaded once when the container starts.

29. Forty lines from nothing to a public URL — B

A — Queueing on shared hardware is real but would not multiply a single request's time by more than a hundred.

C — A cold start happens once on wake, not on every request while traffic is steady.

D — The queue does serialise, which raises waiting time under load but does not lengthen the work itself.

30. Blocks, when Interface is no longer enough — D

A — Blames the queue for what is ordinary Python scope; the queue schedules calls without sharing local data.

B — Invents a response cache that Gradio does not apply by default.

C — Attributes to cookie scope a leak that exists entirely in server-side process state.

31. Duplicate it, and the app becomes yours to change — D

A — Imagines automatic rebuilds happening on an idle Space rather than only on a commit.

B — Assumes billing follows requests served rather than time the hardware is held.

C — Ties the container's lifetime to the owner's browser session.

32. The README front matter is the deployment config — A

B — Invents an architecture mismatch; wheels are resolved at build time for the target platform.

C — Unpinned model revisions do cause real failures, but they surface in loading or inference rather than in constructing UI components.

D — Attributes the failure to a storage policy rather than to the traceback's actual location in the UI library.

33. Hardware, sleep, and the bill that runs while you are asleep — B

A — Describes ZeroGPU-style per-call accounting, which is not how reserved hardware is billed.

C — Invents a bandwidth charge and assumes frequent cold starts on hardware that in fact never slept.

D — Imagines elastic reservation; the hardware tier is fixed once selected.

34. The filesystem disappears, and what to do about it — C

A — Moves the file within the same ephemeral container filesystem.

B — Would persist the data but makes a git commit per request and publishes user text into the Space's own repository.

D — Shares data across users and assumes a graceful shutdown that container restarts do not provide.

35. Streamlit, Docker and a Space with no server at all — A

B — A wrong SDK would change the build entirely rather than produce a running container with no reachable page.

C — Permissions matter for cache writes and surface as import errors, not as a silent blank page.

D — Invents a truncation behaviour; an oversized image fails the build outright.

36. Every Space is an API you can call — C

A — A real fragility, and one that `api_name` is designed to prevent from the Space owner's side.

B — Throttling is likely, but it is a reliability constraint of the same kind already set aside in the question.

D — API calls enter the same queue as web users rather than bypassing it.

37. It built, it started, it is red — B

A — An install failure appears in build logs and prevents the container from starting at all.

C — Handler exceptions surface in the UI and the logs, and do not terminate the container.

D — Sleep stops a Space cleanly after an idle period rather than seconds after boot.

38. Do not download the dataset — B

A — Trades an exact answer for an estimate when the exact answer was cheaply available, and assumes shards are randomly composed.

C — Believes `.filter()` on a normal dataset is applied during download rather than after the full copy is cached.

D — Assumes large-scale statistics require the owner's infrastructure, when the columnar format makes them cheap for anyone.

39. A dataset that behaves like a list and is not one — A

B — Describes leakage, which random splitting does not cause; each row lands in exactly one split.

C — Accuracy is stable here precisely because it is uninformative at 2% prevalence.

D — Training order does introduce variance, but it would not produce swings of the size an almost-empty positive class does.

40. Why a 300 GB dataset opens instantly — C

A — Confuses page caching, which serves bytes from an existing file, with the fingerprint that decides whether to recompute.

B — Immutability is real but each map writes a new file rather than being ignored.

D — Describes streaming behaviour in a situation where the caching path implies a downloaded dataset.

41. Transforming a million rows without waiting an hour — B

A — Parallelism is unrelated to column length alignment.

C — Invents a divisibility rule; batches may produce any number of rows.

D — Inverts the batched contract, which is always a dict of lists regardless of length changes.

42. Query a dataset with SQL before downloading it — B

A — The viewer does show statistics, but an arbitrary SQL query is computed from data rather than from a precomputed summary.

C — Invents sampling; the counts returned are exact.

D — Assumes server-side execution; DuckDB runs locally and pulls the bytes it needs.

43. Turning your files into a dataset — C

A — Encoding efficiency is a side effect and not the reason to reach for it.

B — Validation is a real benefit but not the one that survives contact with other people.

D — Stratification benefits from it but can be done on any column with discrete values.

44. The card is where a dataset stops being a file — C

A — Treats a declaration as a transfer of risk to the publisher, which does not protect a downstream user.

B — Assumes public accessibility settles the question, which is exactly what is contested across jurisdictions.

D — Confuses who is likely to complain with what permissions actually exist.

45. Media columns decode when you touch them — A

B — Resizing changes geometry but does not reverse a crop or a flip.

C — Accessing the column decodes it; a function operating on bytes would raise rather than pass through.

D — Attributes the failure to storage fidelity rather than to a single frozen random draw.

46. Duplicates, leakage, balance, and label noise — C

A — Chunk length affects difficulty but would degrade offline and online scores together.

B — A real hazard, but it would shift the score in either direction rather than systematically flatter it.

D — A genuine deployment mismatch, but it does not explain an offline score that is too high.

47. From a dataset to batches a trainer can eat — D

A — Wasteful and slow, but padded positions are excluded from the loss by the collator.

B — Would prevent learning entirely rather than teaching the model a different task, and the loss would not fall.

C — Produces degraded formatting rather than a systematic role reversal.

48. A list of numbers where near means similar — A

B — Normalisation matters for the arithmetic but would not systematically confuse two near-identical sentences.

C — Truncation affects long inputs; both of these are short enough to encode fully.

D — A small index does produce poor nearest neighbours, but here a semantically opposite document is winning on genuine similarity.

49. The library, and the four calls you need — C

A — Normalisation rescales but cannot recover information discarded with the dropped dimensions.

B — Assumes the loss scales with the proportion cut rather than with how the model was trained.

D — Matryoshka truncation is applied to existing vectors; re-encoding is not what makes it work.

50. Picking the model, and reading MTEB without being fooled — D

A — Misdescribes the aggregation; the average is across tasks for each model.

B — Invents a size weighting that the benchmark does not apply.

C — Contamination affects every column equally rather than singling out retrieval.

51. Cosine, dot product, and the normalisation that connects them — B

A — Would produce matches on everything rather than almost nothing, and the metric is chosen by the caller.

C — Cosine ignores length by construction, so normalisation does not cap it.

D — Invokes a general geometric intuition that does not describe how trained text embeddings distribute.

52. Chunking decides what your search can find — A

B — Plausible in general, but it does not explain why the same model performs well in one language under the same settings.

C — Python strings count characters rather than bytes, so splitting does not break codepoints.

D — More chunks per document changes recall slightly but does not degrade matching the way truncation does.

53. NumPy first, FAISS second, a database rarely — C

A — Index type and normalisation are independent choices made by the caller.

B — Tuning improves recall but an approximate index does not become exact at any setting.

D — Invents a rebuild requirement; queries do not modify the graph.

54. The second pass that fixes the first — B

A — Training data differs, but the architectural ability to compare the two texts jointly is what enables the distinction.

C — Score range affects readability of the output, not the model's ability to tell the pair apart.

D — Correctly notes there is no precomputation, but attributes the benefit to conditioning on surface features rather than joint attention.

55. One space for pictures and words — A

B — Normalisation equalises length without aligning the underlying dimensions' meanings.

C — Shared training data does not produce aligned coordinate systems; alignment must be trained for explicitly.

D — Dimensionality is the only thing checked, which is precisely why the incompatibility passes silently.

56. Thirty questions, and the numbers they give you — C

A — Reranking addresses the gap between recall and precision, which here is small; the binding constraint is the recall ceiling.

B — May raise recall slightly, but a ceiling this low usually means relevant chunks are not findable at any k.

D — Diversity fixes crowding within retrieved results and cannot surface a chunk that never appeared.

57. Three cheaper things to try first — B

A — More training on the same data increases memorisation of phrasing rather than reliable factual recall.

C — Scaling the same approach makes the model more confident without making updates possible or recall exact.

D — Interference is real but secondary to the fact that weights are a poor store for facts that must be exact and updatable.

58. Trainer, and what it saves you writing — B

A — Leakage produces inflated scores, but it would inflate any evaluation method rather than only the training-time one.

C — ROUGE has real weaknesses, but they would not create a gap between training-time and inference-time behaviour.

D — Worth checking and a real source of variance, but a smaller effect than scoring a different task entirely.

59. The six arguments worth understanding — B

A — Learning rate scales updates at application time; gradient tensors are the same size regardless.

C — Both are two bytes per value, so the memory is identical and only the numerical range differs.

D — Checkpoints are written to disk and epochs do not affect peak memory during a step.

60. LoRA: training 0.1% of the parameters — B

A — Base weights remain on the device because the forward pass needs them every step.

C — Activations are largely unchanged, since the full base computation still runs alongside the adapter.

D — Adapter dtype is a minor detail against a hundredfold reduction in the number of optimiser states.

61. QLoRA, and fine-tuning a 7B model on a free GPU — B

A — Checkpointing does cost time and is commonly enabled, but it is a separate choice rather than what quantization itself imposes.

C — The arithmetic is not performed in 4 bits at all; the weights are unpacked before computation.

D — Inverts the effect: less memory allows larger batches, and the question is about per-step cost anyway.

62. TRL, and training on conversations — C

A — A tuning issue would show in the loss curve rather than producing a consistently weaker model.

B — Inverts the relationship; more data means more updates per epoch, not fewer.

D — A real hazard in hand-rolled packing, but TRL masks between packed examples for supported models.

63. Watching a run you cannot sit next to — A

B — Checkpoints save in the training dtype, and rounding would not produce a schedule-shaped jump.

C — Order changes cause small variance rather than a systematic upward jump and a worse final model.

D — A real but tiny effect confined to one or two steps.

64. Loss is NaN, memory is gone, nothing is learning — D

A — Warmup would show a rate ramping up in the logs and a loss that begins moving after it.

B — Plausible arithmetic, but accumulation of that size would be visible in the configuration and still eventually step.

C — Small batches make a curve noisy rather than flat.

65. Shipping the thing you trained — C

A — Merging is exact arithmetic for the trained precision; the approximation concern applies narrowly to QLoRA.

B — Adapters are tied to their exact base; shape compatibility is not behavioural compatibility.

D — Public repository storage is not the constraint that drives this choice.

66. A token in a public Space is a token you have given away — B

A — Invents a network-level restriction where the control is actually per account.

C — Reads the failure as a missing permission level rather than a missing grant on that account.

D — Treats a persistent grant as a session that lapses, which would fail on the laptop too.

67. What the licence tag does and does not settle — A

B — Treats an adapter as separable when it is a derivative that cannot function without the restricted base.

C — Invents a recency rule; licensing is not chronological.

D — Absence of a valid grant means fewer permissions, never more.

68. The leaderboard is not measuring your task — C

A — Explains away a specific pattern with a general one, without asking why the timing lines up with the data cutoff.

B — Assumes the two scores came from different versions when both describe the same released weights.

D — Treats a systematic gap tied to the data cutoff as a surface prompting issue.

69. The evaluation that decides it — A

B — Controlling the conditions removes confounders without making a one-item difference statistically meaningful.

C — High scores may reflect a well-matched task rather than an inadequate set, and replacing it loses comparability.

D — Overcorrects: thirty items detect large differences reliably, which is what the set is for.

70. Renting the compute instead of owning it — B

A — Focuses on a pricing component rather than the structural difference between paying for time and paying for use.

C — Scale-to-zero reduces the bill in this scenario, and initialisation is not separately metered.

D — Serving stacks on endpoints batch continuously, and throughput is not the binding cost factor at this volume.

71. When transformers stops being the right server — B

A — Offloading would show high utilisation punctuated by transfer stalls, and would be slow even for a single request.

C — Cache exhaustion produces errors or evictions rather than low utilisation.

D — Tokenization is negligible against generation and would not hold a GPU at 30%.

72. Publish it so a stranger can use it without writing to you — A

B — Assumes exposure scales with elapsed time rather than with automated copying that happens immediately.

C — Treats history rewriting on the origin as reaching clones made from the previous history.

D — Confuses removing future access with recalling what has already been distributed.

73. Writing a card somebody can audit — C

A — Mitigations are welcome but a card's job is first to describe accurately.

B — Appeals to a standard rather than to what the sentence fails to tell a reader.

D — Reads it as an attribution of blame when the problem is that it says nothing specific at all.

74. Pull requests, discussions and being useful — C

A — Downloads are counted regardless of metadata.

B — Full-text search covers card content independently of the task tag.

D — Invents a ranking penalty rather than describing exclusion from a filter.

75. Still working next year — D

A — Assumes tokenizer updates are cosmetic when they can change ids, special tokens and the chat template.

B — Invents a consistency requirement; components load independently.

C — Only a size change raises; a corrected chat template or altered special tokens passes silently.

Module test — 1. The Hub, read properly

1. B

A repository on the Hub is a git repository, and main moves. Every loading function takes a revision argument that accepts a branch, a tag or a commit hash. Without one, the code means whatever is on main at the moment it runs, so the model can change without the code changing. Pinning is one argument and removes the whole category.

2. C

Git handles source code, not five-gigabyte tensors, so the Hub keeps pointers in git and bytes in a separate large-file store. Without the large-file client installed, git faithfully clones what was committed: a few lines of text naming an object hash. Nothing failed, and hf download or hf_hub_download resolves the real object.

3. A

The count is file resolutions over a rolling thirty days, dominated by continuous integration, library defaults and containers with cold caches. That makes it a floor on how well exercised a model is: the missing tokenizer file and the unparseable config have already been found by somebody. It is not a ranking, because it measures neither quality nor fit.

4. C

Cards declare their parent with a base_model field and the Hub indexes the relationship in both directions, so a base model's page carries links reading Finetunes, Quantizations and Adapters. Following Quantizations gives you every GGUF, AWQ and GPTQ conversion anyone has published, already filtered. Search matches names and card text, not meaning, so it would miss most of them.

5. D

A fine-tune is a derivative of the model it was built on, so the base's terms travel with it however the child is tagged. Mis-tagging is common, sometimes careless and sometimes a sincere belief that an adapter is a separate artefact. The check is to follow base_model until you reach a repository with no parent, and read that licence.

6. B

Safetensors removes one hole: a JSON header plus raw bytes has no code path. The flag opens a different one, by importing modelling code from the repository into your process. The weights format and the loading code are two separate decisions, so read the .py files, pin the revision beside the flag, and run it where no tokens live.

Module test — 2. Transformers, from characters to output

1. B

The pipeline hides five steps, and the last but one is a softmax over the head's unnormalised scores. A softmax always has a maximum, so it always looks confident, including on text nothing in the model has seen. Nothing in the output can tell you the model is out of its depth, and on an English-only default model a Hinglish sentence is exactly that.

2. C

The checkpoint has no classification head, so transformers creates one, fills it with random numbers, and warns rather than failing, because that is the normal first step before fine-tuning. The model then loads, runs and returns confident-looking scores that are noise. Treat that warning as an error, not a log line.

3. D

A tokenizer is a fixed vocabulary plus rules for splitting anything into pieces from it. Common English words get one token; a script the vocabulary barely covers falls back to multi-byte fragments. It is a property of the vocabulary rather than of the language, which is why a multilingual model narrows the gap, and it costs you three times over: price, context window and steps to get right.

4. A

A decoder-only model generates from the last position of the sequence. Pad on the right and the sequence ends in padding, so the model is asked to continue from a pad token with your prompt some distance back. Output degrades, nothing raises, and the same prompt alone still looks fine. Set padding_side to left for any causal model you batch.

5. C

Published generation defaults are inherited from generation_config.json unless you override them. With do_sample false the run is greedy and every sampling parameter you passed is ignored; older versions did not even warn. Print model.generation_config before benchmarking, because two models that seem to differ in quality often differ only here.

6. B

Generation ends on the end-of-sequence token, on max_new_tokens, or on a stopping criterion. Conversions and fine-tunes frequently ship a template whose end marker is not the token the generation config is watching for, so nothing ever fires. Print tok.eos_token_id and model.generation_config.eos_token_id and compare them; several models legitimately need a list.

Module test — 3. Running it on the hardware you actually have

1. A

Each cached repository keeps its real bytes in blobs/ and exposes readable filenames as links under snapshots/<commit>. Deleting the links removes nothing, and two revisions of one model share every blob that did not change. Use hf cache scan and hf cache delete, which also shows you the four copies of the same model from four tutorials.

2. C

Both take two bytes and spend them differently. Float16 gives more bits to precision and caps out around 65,504; an activation above that becomes infinity, infinity times zero is NaN, and the NaN spreads within one layer. Bfloat16 trades precision for float32's exponent range, and for neural networks the range matters more.

3. D

Nothing is broken. device_map auto places what does not fit on the CPU and then on disk, and a disk layer is read from an offload folder for every token of every request. Offloading suits a one-off overnight job; for anything interactive, quantization is the answer, because a heavily quantized model that fits beats a full-precision one that does not.

4. A

This is counter-intuitive and well replicated: at the same memory, the bigger model quantized harder generally wins. Quantization removes redundancy, and a large model has more of it, which is also why small models degrade faster at four bits. It is worth confirming on your own task, because the damage is not spread evenly.

5. B

The loss is not uniform. It lands in multi-step arithmetic, precise JSON, rarely seen languages and long-range consistency, which are exactly the capabilities a quick chat check cannot see. If your application depends on structure, measure it: thirty examples with exact-match scoring at both precisions takes twenty minutes.

6. A

CUDA work is queued asynchronously, so a call returns before the arithmetic has happened. Without a synchronise around the timed region you are measuring how fast Python can issue work. The two companion habits: exclude the warm-up call, which carries one-off allocation and kernel compilation and is two to ten times slower, and report the tail rather than the mean, because a median of 300 milliseconds beside a 95th percentile of four seconds describes a system some people experience as broken.

Module test — 4. Spaces: putting something in front of people

1. B

Loading at import means the model is constructed once when the Space starts. Move that call inside the function and every request downloads or reloads weights, turning a fast response into a slow one and, on a busy Space, exhausting memory. It is invisible while you are the only visitor, which is why it is the commonest performance bug here.

2. C

A Space is one process serving everybody, so a list declared at module level is one list for the whole internet. gr.State is per session: it goes into the handler as an input and comes back as an output, and dies with the browser session. The model is the one thing correctly shared, because it is read-only.

3. D

The 48-hour sleep is a cost control for free hardware and does not apply to a GPU you are renting. Paid hardware meters time, not requests, so a Space with three visitors a day costs what one with three thousand costs. Set the sleep timer first and change the hardware second, because afterwards is when you forget.

4. B

The deployment configuration is YAML at the top of README.md, and sdk_version is the field people leave off. Without it the Space builds against whatever the platform currently defaults to, and a major version of Gradio renamed component arguments. Pin it, pin requirements.txt, and the class of failure disappears.

5. C

An out-of-memory kill comes from the kernel, not from Python, so there is no exception to print. On free CPU hardware with sixteen gigabytes, loading a 7B model at float32 does this reliably. Fix it with a smaller model, a lower dtype, quantization, or larger hardware, in that order of preference.

6. C

Spaces are for demonstration and light use, and a loop over ten thousand items turns somebody's free demo hardware into your batch processor. It is also fragile: their Space can sleep, change or vanish, and you cannot see its logs. Duplicating gives you the hardware, the sleep policy and the data path, which is what a dependency needs.

Module test — 5. Datasets, and the work that happens before any model

1. B

The rule is how many passes you will make. Streaming fetches row groups over HTTP as you consume them, so a second epoch fetches them again; downloading writes Arrow once and memory-maps it, which is limited by disk rather than RAM. Stream when you will read the data roughly once.

2. C

select, shuffle and sort build a list of row positions over a memory-mapped Arrow table. Nothing is rewritten, so reordering a dataset far larger than RAM is instant and free. filter is the exception, because it has to read every row to decide, which is why num_proc matters there and not here.

3. A

A batched map receives a dict of lists and may return lists of a different length, which makes it the tool for splitting and for dropping. The old columns then no longer match, so remove_columns is required or the library refuses the mismatch. This is also why batched mapping is ten to fifty times faster: the Arrow-to-Python crossing is paid once per batch.

4. D

Public datasets are converted to parquet automatically. Parquet is columnar with its metadata in the footer, so a reader can fetch the footer, work out which row groups it needs, and request only those byte ranges over HTTP. A query on one column never touches the others, which is why class balance is answerable on a phone.

5. A

Grouped leakage. Chunks from one document are not independent, so a random split over chunks puts the neighbouring paragraph in training. The general rule is to split along the axis deployment must generalise across: by document, by customer, by speaker, or by date when you are predicting forward.

6. C

Flat from step one is almost never the learning rate; it is the labels not reaching the loss. The forward signature expects a column literally named labels, so label becomes an unused column that is silently dropped. The same shape comes from every label position being masked to -100, or from a PEFT config whose target modules matched nothing.

Module test — 6. Embeddings, and search that understands

1. B

Two texts about the same topic are close whether one answers the other or not, and cancel and renew are near-identical in form and topic. Embeddings retrieve the neighbourhood; deciding within it is what a cross-encoder reranker is for, because it reads the query and the document together.

2. B

Dot product is cosine multiplied by both lengths, so unnormalised search has a bias toward documents whose vectors happen to be long, and nobody can see it in the results. On unit vectors, cosine, dot product and Euclidean distance rank identically, so normalising at encode time removes the choice entirely.

3. C

Several families were trained with a prefix on the query and not on the passages. Omit it and you lose accuracy for no visible reason: nothing errors and the results are merely a bit worse. The card says which, and a large share of reports that a model underperformed its advertisement are this.

4. D

A 1,500-character limit is roughly 375 English tokens and roughly 900 Hindi tokens on an English-centric tokenizer, so the Hindi chunks cross the model's window while the English ones sit comfortably inside it. Count with the model's own tokenizer, or the multilingual system is worst in the language it was added for.

5. A

A reranker can only reorder what it is given, so retrieve at the k where recall plateaus. Rerank cost is linear in candidates, and it already dominates the request. Recall measures the retriever's ceiling; the reranker's job is precision within it, which is where ten to twenty points of hit rate at one come from.

6. D

One matrix multiply searches a million vectors in tens of milliseconds, and 31,000 is nothing. Below a few million vectors, exact search is not merely sufficient, it is better, because every approximate index trades recall for speed. Databases earn their place through filtering, updates and sharing, none of which is search quality.

Module test — 7. Training and adapting, with the libraries the Hub is built on

1. B

Training facts into weights is inefficient, unreliable and impossible to update: the model learns the shape of your documents rather than their contents and produces fluent, confident, subtly wrong versions. Retrieval puts the actual text in front of the model at answer time, can be updated by replacing a file, and can cite its source.

2. C

Under Adam each trainable parameter carries a gradient and two optimiser states, which is where full fine-tuning's roughly fourfold overhead comes from. Freezing the base and training a low-rank update cuts the trainable count by a hundredfold, and the optimiser state falls by the same factor. The weights themselves are still there, which is what QLoRA then attacks.

3. D

Full fine-tuning sits around 2e-5 to 5e-5, because higher rates destroy what pretraining learned. LoRA sits about ten times higher, at 1e-4 to 3e-4, because the adapter matrices start from scratch. The symptoms are unambiguous either way: NaN means too high, a loss that barely moves after an epoch means too low.

4. A

That call casts layer norms to float32, enables gradient checkpointing and makes the input embeddings require gradients. Without it, training over a quantized base is unstable or silently fails to learn, and nothing raises. It is one of the lines whose absence produces a confusing result rather than an error.

5. D

A model that cannot memorise ten rows is not learning at all, so the fault is mechanical rather than statistical. It separates a pipeline bug from a data problem in ten minutes, before you spend three hours discovering the same thing expensively. Check the label column name, the -100 masking, and print_trainable_parameters.

6. B

Diverging curves are overfitting: the model is memorising the training set while getting worse at anything else. load_best_model_at_end with a validation metric keeps the checkpoint from before the turn, which is not the last one and is usually better. The longer-term fixes are fewer epochs, more data, or a lower adapter rank.

Module test — 8. Depending on it, and publishing so others can

1. C

401 means there was no valid token at all; 403 or a GatedRepoError means the token is valid and that account has not been granted access. You accepted the terms as yourself and the code is running as a Space, a service account or a colleague. Acceptance does not propagate, so grant it for the account that actually makes the request.

2. B

A licence is a grant from whoever holds the thing being licensed, about that thing. It is not a warranty about the training data, it does not resolve output ownership, and it cannot override an acceptable-use policy attached separately. The first and third of those are genuinely unresolved and differ by country, which is why a card should disclose rather than reassure.

3. D

At that volume serverless costs a few cents a day and a modest card costs about twenty-four dollars a day whether or not anybody uses it. The crossover sits much further right than teams assume, and it moves further right the more idle the card is, because a GPU idle four fifths of the time has an effective rate five times its headline one.

4. A

The cause is structural. A naive loop finishes one batch before starting the next, so a request arriving one token into a 500-token generation waits for all 500, and the contiguous KV cache reservation wastes most of the memory it holds. Continuous batching and paged attention fix both, and the gain is several times rather than a few per cent.

5. C

Thirty examples reliably detect large differences — a model that cannot handle your language, a configuration that broke structured output — and cannot detect one or two points. Acting on a one-item gap is superstition. If a small difference genuinely matters, you need several hundred items, which is a different project with a different budget.

6. D

Four things move independently: the model's main branch, the libraries, the platform defaults and your own corpus. Pinning the first two is nearly free — a revision argument on every from_pretrained, and exact versions in requirements.txt, including sdk_version for a Space. The other two need habits rather than a pin.

Hugging Face, End to End — final examination

1. C ***1. The Hub, read properly***

The listing includes formats you will not download — many repositories carry both .bin and .safetensors copies of the same weights — so adding it up overstates the bill. The index file's total_size covers exactly the shards of one format, and model_info with files_metadata gives per-file sizes before a byte moves.

2. A ***1. The Hub, read properly***

snapshot_download takes the repository, so duplicated formats are downloaded twice over. allow_patterns and ignore_patterns are the difference between a six-gigabyte download and a twelve-gigabyte one, and hf download takes the same include filters on the command line.

3. B ***1. The Hub, read properly***

It is largely a text match over names and card content, not a semantic search over capability. A model downloaded four million times will not appear if its uploader never wrote your phrase. The filter sidebar is the real interface: task tag, library, language and licence compose into a URL you can bookmark.

4. C ***1. The Hub, read properly***

Gating and licensing are separate. The gate controls who may fetch the bytes; the licence controls what may then be done with them, and the two often disagree in strictness. Access is also per account, which is why a Space with its own token still gets a 403 after you accepted the terms as yourself.

5. D ***1. The Hub, read properly***

Most facets on the Hub are metadata somebody typed, which makes them claims. The dataset size band is derived from the automatic conversion and the viewer, so it is one of the more trustworthy facets on the site. Language tags in particular are a claim, not a measurement.

6. B ***1. The Hub, read properly***

An adapter is a small set of extra matrices with no network of its own. adapter_config.json records base_model_name_or_path precisely because the adapter is useless without its exact base, and PeftModel.from_pretrained applies it over a base you load first.

7. C ***1. The Hub, read properly***

A like is one click by one signed-in person, so it measures attention and skews hard toward an announcement. Downloads are dominated by machines, so they measure how much other software depends on the thing. You usually want the second, and neither is a quality signal.

8. C ***2. Transformers, from characters to output***

AutoModel loads the network body and nothing else, returning one vector per position. Naming a head with AutoModelForX attaches a task layer, and on a checkpoint that has no such head the layer is created at random. Use AutoModel when you want representations, and a head class when you want a prediction.

9. B ***2. Transformers, from characters to output***

config.id2label carries the names, and an uploader who left it as defaults published a model whose label meanings live only in the card, or in a discussion thread. Assuming index 1 means positive is a silent fifty per cent chance of inverting every result you produce. Check, do not assume.

10. B ***2. Transformers, from characters to output***

The template has already inserted the special tokens. Encoding it again with add_special_tokens left at its default prepends another marker, which puts the model slightly outside the format it was trained on and, in a masking pipeline, shifts every index by one. Pass add_special_tokens=False, or tokenize through the template itself.

11. C ***2. Transformers, from characters to output***

Truncation is silent and defaults to cutting from the right, so a 512-token window keeps the polite introduction and discards the accusation at the end. Keeping the end costs one setting. The alternatives are chunking deliberately or choosing a model with a longer window.

12. A ***2. Transformers, from characters to output***

A batch runs at the length of its longest member, so one two-thousand-token item among thirty-one short ones wastes most of the compute. Sorting makes each batch internally uniform and routinely doubles throughput on support tickets. Restoring the original order afterwards is the step people forget, and forgetting it mislabels every row.

13. D ***2. Transformers, from characters to output***

generate blocks until it finishes, so it is started on its own thread with the streamer passed in, and the main thread iterates the streamer as a queue. Chunks are not tokens and not words — the decoder buffers until it can emit valid text — so never treat a chunk boundary as a word boundary.

14. B ***2. Transformers, from characters to output***

max_length counts the prompt and the continuation together, so a long prompt leaves a small remainder for the answer. max_new_tokens counts only what is generated and is almost always what people mean. Always set one of them: a model with no limit and a broken stop token will generate to the end of its context window.

15. C ***3. Running it on the hardware you actually have***

Attention keys and values for every token so far are kept rather than recomputed, which is what makes each step cheap. It is also why a batch that fits at 512 tokens fails at 4,096. Failing at load is a weights problem; failing partway through generation is nearly always the cache.

16. A ***3. Running it on the hardware you actually have***

Self-containment is the point: there is no config to match, no tokenizer to mismatch and no virtual environment to break. You download one file and run it, on a CPU, on Apple silicon or on a phone, which is what makes this the most important free path for anyone without a graphics card.

17. B ***3. Running it on the hardware you actually have***

bitsandbytes buys memory, not throughput: weights are stored small and computed in a wider type, so there is work on every multiply. Pre-quantized AWQ and GPTQ repositories use kernels written for the packed format and are faster. If it fits at bfloat16, use bfloat16 — quantization is a constraint, not an improvement.

18. C ***3. Running it on the hardware you actually have***

transformers.js fetches an ONNX build into the page and executes it on the visitor's device. Hosting is then a static file: no queue, no sleep, no per-request compute, and the user's text never leaves their machine. The ceiling is the download — tens to low hundreds of megabytes is practical, a chat model is not.

19. D ***3. Running it on the hardware you actually have***

The machine is temporary, so write code that assumes it: save to a private Hub repository rather than to the session, set save_steps you can afford to lose, and resume from a checkpoint on restart. Getting the script right on a small model on a CPU first means the switch to a large model on a GPU usually runs unchanged.

20. B ***3. Running it on the hardware you actually have***

Distillation trains a small model to imitate a large one on a distribution. Outside that distribution the small model can be relatively worse, and accented speech, telephone audio or a language the distillation set barely covered are exactly where the gap opens. The trade is measured and published, and it is measured on somebody else's data.

21. C ***3. Running it on the hardware you actually have***

Unusual control flow, custom operators and remote-code implementations can export with warnings and then produce subtly different numbers. Two runs of the same broken graph agree perfectly. An allclose check against the PyTorch logits is the only test that catches it, and the alternative is an unexplained accuracy drop later.

22. A ***4. Spaces: putting something in front of people***

Clickable examples turn a blank form into something a person can try in one click, without knowing what to type. The description matters for honesty and flagging matters for privacy, but neither decides whether somebody engages at all. A limitation in the description costs nothing and prevents misuse; examples decide whether anyone gets that far.

23. C ***4. Spaces: putting something in front of people***

ZeroGPU attaches a slice of a GPU for up to the duration you asked for and then releases it. Asking for too little kills the call mid-generation; asking for far too much wastes quota. There is overhead on every allocation as well, so many tiny GPU calls are worse than one batched call.

24. B ***4. Spaces: putting something in front of people***

Most of a cold start on a large model is downloading the weights again. Pointing HF_HOME at the volume before importing transformers turns a three-minute wake into something closer to fifteen seconds. The volume is not a database — one volume, one Space, no backups by default and no concurrency control beyond what you write.

25. C ***4. Spaces: putting something in front of people***

The platform routes to port 7860, and a process bound to localhost is unreachable from outside its own container. Both produce a Space that builds and starts and serves nothing. The third classic Docker failure is different: a root-owned cache directory causes permission errors at import, which do appear in the container logs.

26. D ***4. Spaces: putting something in front of people***

Secrets are stored encrypted, handed to your process as environment variables, never shown on the page and never carried into a duplicate. That is correct behaviour and occasionally confusing. What must never happen is a token in app.py, because a public Space's files are public, permanently and to everyone.

27. B ***4. Spaces: putting something in front of people***

A Space is one process serving many visitors, so two handlers can be writing at the same moment. Without the lock their lines interleave and the file stops being parseable. The scheduler itself is there so you are not making a git commit per request; the lock is there so what it pushes is valid.

28. A ***4. Spaces: putting something in front of people***

Gradio exposes every event handler as an endpoint automatically, so naming one gives it a stable route that survives layout changes rather than creating it. api_name=False removes a handler from the API entirely, which is what you want for anything that is a user-interface detail rather than a capability.

29. B ***5. Datasets, and the work that happens before any model***

Indexing a row gives a dict; slicing gives a dict of lists, because the storage is columnar. Half the confusing errors in a first hour with the library come from expecting a list of dicts. It is also why a batched map receives lists and must return lists of a matching length.

30. C ***5. Datasets, and the work that happens before any model***

A random ten per cent split of a two per cent class can easily land with a handful of positives or none, and a recall figure computed on four items is noise. Stratification keeps the proportions equal across both halves. Reproducibility is a separate argument, and that is what seed is for.

31. D ***5. Datasets, and the work that happens before any model***

Every transformation writes a new Arrow file keyed by a hash of the operation, and the hash covers the function's bytecode. Editing the function invalidates it correctly; changing a value the function merely reads does not. Pass parameters as arguments rather than closing over globals, or pass load_from_cache_file=False.

32. B ***5. Datasets, and the work that happens before any model***

Augmentation has to differ between epochs, which is the whole point of it. map materialises a new Arrow file, so the randomness is drawn once and then reused, producing a model that overfits despite augmentation. set_transform applies on access and re-runs every time. The rule: map for deterministic preprocessing, set_transform for anything random.

33. A ***5. Datasets, and the work that happens before any model***

Audio and Image columns decode lazily, so casting costs nothing until a row is read and then resamples on the way out. It fixes the commonest silent failure in speech work, where a mismatched rate produces a much worse transcript and no error. It cannot restore information that was never recorded, so telephone audio still has a higher error rate.

34. C ***5. Datasets, and the work that happens before any model***

push_to_hub writes the dataset as it currently exists, including every transformation applied in the session. Publishing token ids bakes in your model choice and makes the dataset useless to anyone else. Publish the raw form and keep the preprocessing in a script beside it: publish the thing, not your view of the thing.

35. D ***5. Datasets, and the work that happens before any model***

Where two trained labellers disagree, no model can be right against both, so label noise caps the number you can measure. Knowing 78 per cent before you start saves weeks of chasing a ceiling. It is also the rarest figure on a dataset card, and its absence is itself worth noticing.

36. B ***6. Embeddings, and search that understands***

Getting from token vectors to one good sentence vector requires the pooling the model was trained with, and guessing produces something that works badly enough to mislead. sentence-transformers stores the right pooling beside the weights. The deeper point is that the capability is contrastively trained in, which is why a perfectly good language model makes famously poor sentence embeddings.

37. C ***6. Embeddings, and search that understands***

The vector reflects the whole text, and almost all of this text is shared. A single negating token barely moves it. This is a real failure of the mechanism rather than a tuning problem, so for any task where negation decides the answer, plan for a cross-encoder or a rule rather than hoping for a better embedding model.

38. D ***6. Embeddings, and search that understands***

One model's clearly related is 0.85 and another's is 0.55, and text embeddings occupy a narrow cone where two unrelated sentences may already score 0.35. Rank rather than threshold wherever you can. Where you must threshold, derive it by encoding a few hundred known-related and known-unrelated pairs of your own and finding where they separate.

39. A ***6. Embeddings, and search that understands***

Matryoshka training makes the early dimensions carry most of the information, so truncating costs a little accuracy and saves a third of the memory and search time. Nothing makes that true of an ordinary model: its dimensions are not ordered by importance, and cutting them destroys the geometry even if both sides are cut alike.

40. B ***6. Embeddings, and search that understands***

If one paragraph matches, its neighbours match too, so the top five can cover one part of the question five times. MMR picks each next result by balancing similarity to the query against dissimilarity to what is already chosen. A reranker will happily rank five duplicates at the top, so apply MMR after reranking rather than instead of it.

41. C ***6. Embeddings, and search that understands***

The same shape is not the same space. Nothing in an array says which model produced it, so the dot products compute and the ranking is meaningless. Store the model name and revision with every index and refuse to load one whose recorded model does not match the query encoder — four lines against a bug that is almost undiagnosable from its symptoms.

42. A ***6. Embeddings, and search that understands***

Vectors from two models are not comparable, so the whole corpus is re-encoded — hours of compute on a million chunks, or a real bill on a hosted API. Two habits follow: settle the model with your thirty questions before encoding at scale, and store the chunk text beside the vectors so re-encoding is local rather than a re-extraction from source documents.

43. A ***7. Training and adapting, with the libraries the Hub is built on***

Accumulation runs four examples, keeps the gradients, runs four more, and updates after eight rounds. Effective batch is the product of the two numbers and the number of devices, and it is what matters for stability. It costs time rather than quality, and it is how a sixteen-gigabyte free card trains at batch sizes that would otherwise need eighty.

44. C ***7. Training and adapting, with the libraries the Hub is built on***

Activations are discarded in the forward pass and recomputed when they are needed for the backward pass, which cuts activation memory substantially at a speed cost of roughly twenty to thirty per cent. It is almost always worth it, and it sits second on the memory list after lowering the batch size and raising accumulation, which are free.

45. B ***7. Training and adapting, with the libraries the Hub is built on***

The learned update is multiplied by alpha divided by r, so raising the rank alone quietly lowers the scale of every update — which reads as a learning-rate change you did not make. The usual convention is alpha equal to twice r, so r=16 with alpha=32, and keeping to it makes rank comparisons mean something.

46. A ***7. Training and adapting, with the libraries the Hub is built on***

Merging removes the small inference overhead of the extra matrices and produces an ordinary model any serving stack can load. It also produces a full-size model you must store, and forfeits the swap-per-request architecture that makes one base plus a directory of small adapters affordable. Merge for a stack that cannot load adapters, and otherwise keep them separate.

47. B ***7. Training and adapting, with the libraries the Hub is built on***

Packed examples share a sequence, so without masking between them one conversation can attend to the previous one. TRL handles this for supported models; hand-rolled packing usually does not, which is why it is a good exercise and a bad production choice. The gain is real on datasets of short exchanges, because no compute is spent on padding.

48. D ***7. Training and adapting, with the libraries the Hub is built on***

Restarting from weights alone resets the learning-rate schedule, which produces a visible jump in the loss and a worse model. The argument restores the optimiser's momentum and variance estimates and the scheduler's position, so the run continues rather than beginning again. With hub_strategy set to checkpoint, the state survives the machine as well.

49. C ***7. Training and adapting, with the libraries the Hub is built on***

The adapter is a few megabytes of your work; the merge is several gigabytes of mostly somebody else's, republished. Publishing the adapter makes clear which base it needs, stays cheap to download, and lets several adapters share one loaded base at serving time. Publish a merge only for a reason, and say in the card what it is a merge of.

50. B ***8. Depending on it, and publishing so others can***

Anything holding a token is you, as far as the Hub is concerned. A fine-grained token ticks exactly the permissions it needs and can be limited to named repositories or an organisation, so what leaks is bounded. Name it after where it lives, so that in six months you can revoke one without breaking the other four.

51. D ***8. Depending on it, and publishing so others can***

Reasoning about whether anybody saw it requires information you do not have, and automated scanners crawl for the token pattern within minutes of publication. Platform scanning is a safety net that fires after publication rather than a plan. Revoking is cheap and certain, which is why it is the rule rather than a judgement call.

52. A ***8. Depending on it, and publishing so others can***

Benchmarks are published so people can use them, so their questions and answers sit on the public web that models are pretrained on. It is usually structural rather than cheating, and it worsens the longer a benchmark exists. The tell is a strong score on an old public benchmark beside an ordinary score on one released after the training cutoff.

53. B ***8. Depending on it, and publishing so others can***

Arenas measure something real that no automatic benchmark captures, and they also measure formatting, length, confidence and agreeableness. A model that writes four hundred tidy words and hedges nothing beats one that is correct in two sentences, and a model tuned to agree beats one that says the premise is wrong. Read the rank as what it is.

54. C ***8. Depending on it, and publishing so others can***

When many requests share an opening — a long system prompt, or a retrieved document — the computation for that prefix is done once and reused. For a retrieval-augmented application where every request carries the same instructions, it is a large and easy saving. Continuous batching and paged attention help too, for different reasons.

55. D ***8. Depending on it, and publishing so others can***

At training time, four-bit bitsandbytes is often slower than bfloat16 because weights are dequantized on the fly: you are buying memory. A serving stack loading AWQ, GPTQ or a native low-precision format uses kernels written for the packed layout, so the model is genuinely faster as well as smaller. Hold the two pictures separately.

56. B ***8. Depending on it, and publishing so others can***

Because a repository is git, a proposal lives as a real ref and every loading function takes a revision. So a missing chat template, a safetensors conversion or a corrected card can be tested before anybody merges it. That is also the route by which the highest-value contributions are made, and most take under an hour.